Basic Computer Coding: C++

2nd Edition

BIBLIOTEX
Digital Library

BASIC COMPUTER CODING: C++

2nd Edition

BIBLIOTEX
Digital Library

www.bibliotex.com

BASIC COMPUTER CODING: C++

2ND EDITION

BIBLIOTEX
Digital Library

www.bibliotex.com

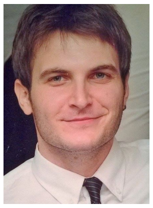
email: info@bibliotex.com

e-book Edition 2022

ISBN: 978-1-98467-608-5 (e-book)

In Collaboration with 3G E-Learning LLC. Originally P-ublished in printed book format by 3G E-Learning LLC with ISBN 978-1-98465901-9

EDITORIAL BOARD



Aleksandar Mratinković was born on May 5, 1988 in Arandjelovac, Serbia. He has graduated on Economic high school (2007), The College of Tourism in Belgrade (2013), and also has a master degree of Psychology (Faculty of Philosophy, University of Novi Sad). He has been engaged in different fields of psychology (Developmental Psychology, Clinical Psychology, Educational Psychology and Industrial Psychology) and has published several scientific works.

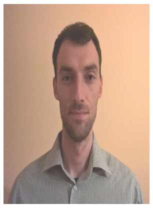
Dan Piestun (PhD) is currently a startup entrepreneur in Israel working on the interface of Agriculture and Biomedical Sciences and was formerly president-CEO of the National Institute of Agricultural Research (INIA) in Uruguay. Dan is a widely published scientist who has received many honours during his career including being a two-time recipient of the Amit Golda Meir Prize from the Hebrew University of Jerusalem, his areas of expertise includes stem cell molecular biology, plant and animal genetics and bioinformatics. Dan's passion for applied science and technological solutions did not stop him from pursuing a deep connection to the farmer, his family and nature. Among some of his interest and practices counts enjoying working as a beekeeper and onboard fishing.

Hazem Shawky Fouda has a PhD. in Agriculture Sciences, obtained his PhD. From the Faculty of Agriculture, Alexandria University in 2008, He is working in Cotton Arbitration & Testing General Organization (CATGO).

Felecia Killings is the Founder and CEO of LiyahAmore Publishing, a publishing company committed to providing technical and educational services and products to Christian Authors. She operates as the Senior Editor and Writer, the Senior Writing Coach, the Content Marketing Specialist, Editor-in-Chief to the company's quarterly magazine, the Executive and Host of an international virtual network, and the Executive Director of the company's online school for Authors. She is a former high-school English instructor and professional development professor. She possesses a Master of Arts degree in Education and a Bachelor's degree in English and African American studies.

Dr. Sandra El Hajj, Ph.D. in Health Sciences from Nova Southeastern University, Florida, USA is a health professional specialized in Preventive and Global Health. With her 12 years of education obtained from one of the most prominent universities in Beirut, in addition to two leading universities in the State of Florida (USA), Dr. Sandra made sure to incorporate interdisciplinary and multicultural approaches in her work. Her long years of studies helped her create her own miniature world of knowledge linking together the healthcare field with Medical Research, Statistics, Food Technology, Environmental & Occupational Health, Preventive Health and most noteworthy her precious last degree of Global Health. Till today, she is the first and only doctor specialized in Global Health in the Middle East area.

Fozia Parveen has a Dphil in Sustainable Water Engineering from the University of Oxford. Prior to this she has received MS in Environmental Sciences from National University of Science and Technology (NUST), Islamabad Pakistan and BS in Environmental Sciences from Fatima Jinnah Women University (FJWU), Rawalpindi.



Igor Krunic 2003-2007 in the School of Economics. After graduating in 2007, he went on to study at The College of Tourism, at the University of Belgrade where he got his bachelor degree in 2010. He was active as a third-year student representative in the student parliament.Then he went on the Faculty of science, at the University of Novi Sad where he successfully defended his master's thesis in 2013. The crown of his study was the work titled Opportunities for development of cultural tourism in Cacak". Later on, he became part of a multinational company where he got promoted to a deputy director of logistic. Nowadays he is a consultant and writer of academic subjects in the field of tourism.

Dr. Jovan Pehcevski obtained his PhD in Computer Science from RMIT University in Melbourne, Australia in 2007. His research interests include big data, business intelligence and predictive analytics, data and information science, information retrieval, XML, web services and service-oriented architectures, and relational and NoSQL database systems. He has published over 30 journal and conference papers and he also serves as a journal and conference reviewer. He is currently working as a Dean and Associate Professor at European University in Skopje, Macedonia.

Dr. Tanjina Nur finished her PhD in Civil and Environmental Engineering in 2014 from University of Technology Sydney (UTS). Now she is working as Post-Doctoral Researcher in the Centre for Technology in Water and Wastewater (CTWW) and published about eight International journal papers with 80 citations. Her research interest is wastewater treatment technology using adsorption process.

Stephen obtained his PhD from the University of North Carolina at Charlotte in 2013 where his graduate research focused on cancer immunology and the tumor microenvironment. He received postdoctoral training in regenerative and translational medicine, specifically gastrointestinal tissue engineering, at the Wake Forest Institute of Regenerative Medicine. Currently, Stephen is an instructor for anatomy and physiology and biology at Forsyth Technical Community College.

Michelle holds a Masters of Business Administration from the University of Phoenix, with a concentration in Human Resources Management. She is a professional author and has had numerous articles published in the Henry County Times and has written and revised several employee handbooks for various YMCA organizations throughout the United States.

HOW TO USE THE BOOK

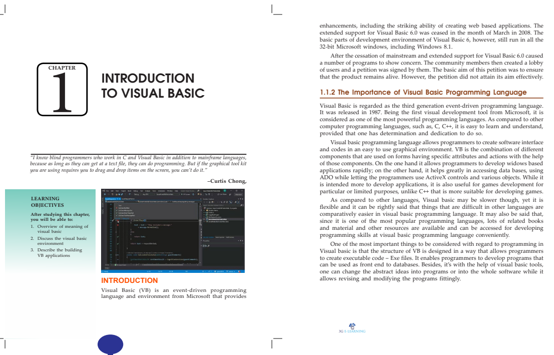
This book has been divided into many chapters. Chapter gives the motivation for this book and the use of templates. The text is presented in the simplest language. Each paragraph has been arranged under a suitable heading for easy retention of concept. Keywords are the words that academics use to reveal the internal structure of an author's reasoning. Review questions at the end of each chapter ask students to review or explain the concepts. References provides the reader an additional source through which he/she can obtain more information regarding the topic.

LEARNING OBJECTIVES

See what you are going to cover and what you should already know at the start of each chapter

ABOUT THIS CHAPTER

An introduction is a beginning of section which states the purpose and goals of the topics which are discussed in the chapter. It also starts the topics in brief.



CHAPTER 1

INTRODUCTION TO VISUAL BASIC

"I know blind programmers who work in C and Visual Basic in addition to mainframe languages, because as long as they can get at a text file, they can do programming. But if the graphical tool kit you are using requires you to drag and drop items on the screen, you can't do it."

–Curtis Chong,

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Overview of meaning of visual basic
2. Discuss the visual basic environment
3. Describe the building VB applications

INTRODUCTION

Visual Basic (VB) is an event-driven programming language and environment from Microsoft that provides

enhancements, including the striking ability of creating web based applications. The extended support for Visual Basic 6.0 was ceased in the month of March in 2008. The basic parts of development environment of Visual Basic 6, however, still run in all the 32-bit Microsoft windows, including Windows 8.1.

After the cessation of mainstream and extended support for Visual Basic 6.0 caused a number of programs to show concern. The community members then created a lobby of users and a petition was signed by them. The basic aim of this petition was to ensure that the product remains alive. However, the petition did not attain its aim effectively.

1.1.2 The Importance of Visual Basic Programming Language

Visual Basic is regarded as the third generation event-driven programming language. It was released in 1987. Being the first visual development tool from Microsoft, it is considered as one of the most powerful programming languages. As compared to other computer programming languages, such as, C, C++, it is easy to learn and understand, provided that one has determination and dedication to do so.

Visual basic programming language allows programmers to create software interface and codes in an easy to use graphical environment. VB is the combination of different components that are used on forms having specific attributes and actions with the help of those components. On the one hand it allows programmers to develop widows based applications rapidly; on the other hand, it helps greatly in accessing data bases, using ADO while letting the programmers use ActiveX controls and various objects. While it is intended more to develop applications, it is also useful for games development for particular or limited purposes, unlike C++ that is more suitable for developing games.

As compared to other languages, Visual basic may be slower though, yet it is flexible and it can be rightly said that things that are difficult in other languages are comparatively easier in visual basic programming language. It may also be said that, since it is one of the most popular programming languages, lots of related books and material and other resources are available and can be accessed for developing programming skills at visual basic programming language conveniently.

One of the most important things to be considered with regard to programming in Visual basic is that the structure of VB is designed in a way that allows programmers to create executable code – Exe files. It enables programmers to develop programs that can be used as front end to databases. Besides, it's with the help of visual basic tools, one can change the abstract ideas into programs or into the whole software while it allows revising and modifying the programs fittingly.

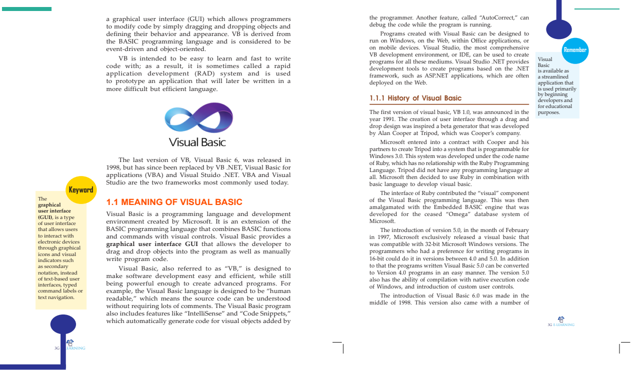
3G E-LEARNING

REMEMBER

This revitalizes a must read information of the topic.

KEYWORDS

This section contains some important definitions that are discussed in the chapter. A keyword is an index entry that identifies a specific record or document. It also gives the extra information to the reader and an easy way to remember the word definition.



a graphical user interface (GUI) which allows programmers to modify code by simply dragging and dropping objects and defining their behavior and appearance. VB is derived from the BASIC programming language and is considered to be event-driven and object-oriented.

VB is intended to be easy to learn and fast to write code with; as a result, it is sometimes called a rapid application development (RAD) system and is used to prototype an application that will later be written in a more difficult but efficient language.

Visual Basic

The last version of VB, Visual Basic 6, was released in 1998, but has since been replaced by VB .NET, Visual Basic for applications (VBA) and Visual Stuido .NET. VBA and Visual Studio are the two frameworks most commonly used today.

Keyword

The **graphical user interface (GUI)**, is a type of user interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation, instead of text-based user interfaces, typed command labels or text navigation.

1.1 MEANING OF VISUAL BASIC

Visual Basic is a programming language and development environment created by Microsoft. It is an extension of the BASIC programming language that combines BASIC functions and commands with visual controls. Visual Basic provides a **graphical user interface GUI** that allows the developer to drag and drop objects into the program as well as manually write program code.

Visual Basic, also referred to as "VB," is designed to make software development easy and efficient, while still being powerful enough to create advanced programs. For example, the Visual Basic language is designed to be "human readable," which means the source code can be understood without requiring lots of comments. The Visual Basic program also includes features like "IntelliSense" and "Code Snippets," which automatically generate code for visual objects added by

3G E-LEARNING

the programmer. Another feature, called "AutoCorrect," can debug the code while the program is running.

Programs created with Visual Basic can be designed to run on Windows, on the Web, within Office applications, or on mobile devices. Visual Studio, the most comprehensive VB development environment, or IDE, can be used to create programs for all these mediums. Visual Studio .NET provides development tools to create programs based on the .NET framework, such as ASP.NET applications, which are often deployed on the Web.

Remember

Visual Basic is available as a streamlined application that is used primarily by beginning developers and for educational purposes.

1.1.1 History of Visual Basic

The first version of visual basic, VB 1.0, was announced in the year 1991. The creation of user interface through a drag and drop design was inspired a beta generator that was developed by Alan Cooper at Tripod, which was Cooper's company.

Microsoft entered into a contract with Cooper and his partners to create Tripod into a system that is programmable for Windows 3.0. This system was developed under the code name of Ruby, which has no relationship with the Ruby Programming Language. Tripod did not have any programming language at all. Microsoft then decided to use Ruby in combination with basic language to develop visual basic.

The interface of Ruby contributed the "visual" component of the Visual Basic programming language. This was then amalgamated with the Embedded BASIC engine that was developed for the ceased "Omega" database system of Microsoft.

The introduction of version 5.0, in the month of February in 1997, Microsoft exclusively released a visual basic that was compatible with 32-bit Microsoft Windows versions. The programmers who had a preference for writing programs in 16-bit could do it in versions between 4.0 and 5.0. In addition to that the programs written Visual Basic 5.0 can be converted to Version 4.0 programs in an easy manner. The version 5.0 also has the ability of compilation with native execution code of Windows, and introduction of custom user controls.

The introduction of Visual Basic 6.0 was made in the middle of 1998. This version also came with a number of

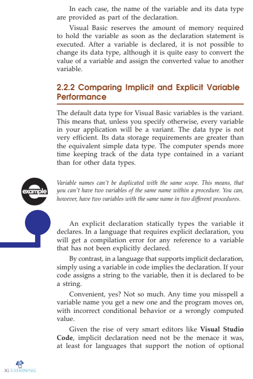
3G E-LEARNING

DID YOU KNOW?

This section equip readers the interesting facts and figures of the topic.

EXAMPLE

The book cabinets' examples to illustrate specific ideas in each chapter.



In each case, the name of the variable and its data type are provided as part of the declaration.

Visual Basic reserves the amount of memory required to hold the variable as soon as the declaration statement is executed. After a variable is declared, it is not possible to change its data type, although it is quite easy to convert the value of a variable and assign the converted value to another variable.

2.2.2 Comparing Implicit and Explicit Variable Performance

The default data type for Visual Basic variables is the variant. This means that, unless you specify otherwise, every variable in your application will be a variant. The data type is not very efficient. Its data storage requirements are greater than the equivalent simple data type. The computer spends more time keeping track of the data type contained in a variant than for other data types.

Variable names can't be duplicated with the same scope. This means, that you can't have two variables of the same name within a procedure. You can, however, have two variables with the same name in two different procedures.

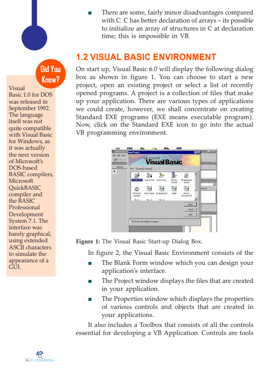
An explicit declaration statically types the variable it declares. In a language that requires explicit declaration, you will get a compilation error for any reference to a variable that has not been explicitly declared.

By contrast, in a language that supports implicit declaration, simply using a variable in code implies the declaration. If your code assigns a string to the variable, then it is declared to be a string.

Convenient, yes? Not so much. Any time you misspell a variable name you get a new one and the program moves on, with incorrect conditional behavior or a wrongly computed value.

Given the rise of very smart editors like **Visual Studio Code**, implicit declaration need not be the menace it was, at least for languages that support the notion of optional

3G E-LEARNING



- There are some, fairly minor disadvantages compared with C. C has better declaration of arrays – its possible to initialize an array of structures in C at declaration time; this is impossible in VB.

1.2 VISUAL BASIC ENVIRONMENT

Did You Know?

Visual Basic 1.0 for DOS was released in September 1992. The language itself was not quite compatible with Visual Basic for Windows, as it was actually the next version of Microsoft's DOS-based BASIC compilers, Microsoft QuickBASIC compiler and the BASIC Professional Development System 7.1. The interface was barely graphical, using extended ASCII characters to simulate the appearance of a GUI.

On start up, Visual Basic 6.0 will display the following dialog box as shown in figure 1. You can choose to start a new project, open an existing project or select a list of recently opened programs. A project is a collection of files that make up your application. There are various types of applications we could create, however, we shall concentrate on creating Standard EXE programs (EXE means executable program). Now, click on the Standard EXE icon to go into the actual VB programming environment.

Figure 1: The Visual Basic Start-up Dialog Box.

In figure 2, the Visual Basic Environment consists of the

- The Blank Form window which you can design your application's interface.
- The Project window displays the files that are created in your application.
- The Properties window which displays the properties of various controls and objects that are created in your applications.

It also includes a Toolbox that consists of all the controls essential for developing a VB Application. Controls are tools

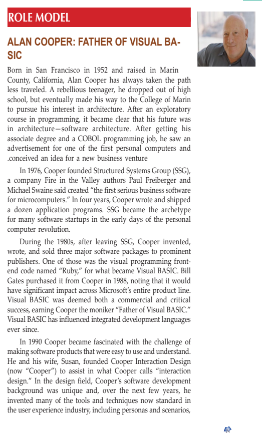
3G E-LEARNING

ROLE MODEL

A biography of someone who has/had acquired remarkable success in their respective field as Role Models are important because they give us the ability to imagine our future selves.

CASE STUDY

This reveals what students need to create and provide an opportunity for the development of key skills such as communication, group working and problem solving.



ROLE MODEL

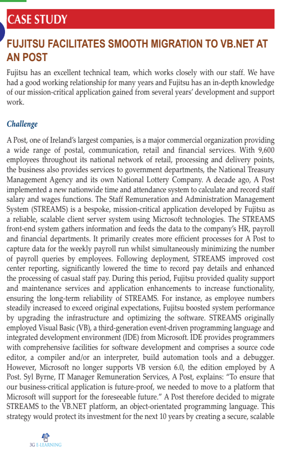
ALAN COOPER: FATHER OF VISUAL BASIC

Born in San Francisco in 1952 and raised in Marin County, California, Alan Cooper has always taken the path less traveled. A rebellious teenager, he dropped out of high school, but eventually made his way to the College of Marin to pursue his interest in architecture. After an exploratory course in programming, it became clear that his future was in architecture—software architecture. After getting his associate degree and a COBOL programming job, he saw an advertisement for one of the first personal computers and .conceived an idea for a new business venture

In 1976, Cooper founded Structured Systems Group (SSG), a company Fire in the Valley authors Paul Freiberger and Michael Swaine said created "the first serious business software for microcomputers." In four years, Cooper wrote and shipped a dozen application programs. SSG became the archetype for many software startups in the early days of the personal computer revolution.

During the 1980s, after leaving SSG, Cooper invented, wrote, and sold three major software packages to prominent publishers. One of those was the visual programming front-end code named "Ruby," for what became Visual BASIC. Bill Gates purchased it from Cooper in 1988, noting that it would have significant impact across Microsoft's entire product line. Visual BASIC was deemed both a commercial and critical success, earning Cooper the moniker "Father of Visual BASIC." Visual BASIC has influenced integrated development languages ever since.

In 1990 Cooper became fascinated with the challenge of making software products that were easy to use and understand. He and his wife, Susan, founded Cooper Interaction Design (now "Cooper") to assist in what Cooper calls "interaction design." In the design field, Cooper's software development background was unique and, over the next few years, he invented many of the tools and techniques now standard in the user experience industry, including personas and scenarios,



CASE STUDY

FUJITSU FACILITATES SMOOTH MIGRATION TO VB.NET AT AN POST

Fujitsu has an excellent technical team, which works closely with our staff. We have had a good working relationship for many years and Fujitsu has an in-depth knowledge of our mission-critical application gained from several years' development and support work.

Challenge

A Post, one of Ireland's largest companies, is a major commercial organization providing a wide range of postal, communication, retail and financial services. With 9,600 employees throughout its national network of retail, processing and delivery points, the business also provides services to government departments, the National Treasury Management Agency and its own National Lottery Company. A decade ago, A Post implemented a new nationwide time and attendance system to calculate and record staff salary and wages functions. The Staff Remuneration and Administration Management System (STREAMS) is a bespoke, mission-critical application developed by Fujitsu as a reliable, scalable client server system using Microsoft technologies. The STREAMS front-end system gathers information and feeds the data to the company's HR, payroll and financial departments. It primarily creates more efficient processes for A Post to capture data for the weekly payroll run whilst simultaneously minimizing the number of payroll queries by employees. Following deployment, STREAMS improved cost center reporting, significantly lowered the time to record pay details and enhanced the processing of casual staff pay. During this period, Fujitsu provided quality support and maintenance services and application enhancements to increase functionality, ensuring the long-term reliability of STREAMS. For instance, as employee numbers steadily increased to exceed original expectations, Fujitsu boosted system performance by upgrading the infrastructure and optimizing the software. STREAMS originally employed Visual Basic (VB), a third-generation event-driven programming language and integrated development environment (IDE) from Microsoft. IDE provides programmers with comprehensive facilities for software development and comprises a source code editor, a compiler and/or an interpreter, build automation tools and a debugger. However, Microsoft no longer supports VB version 6.0, the edition employed by A Post. Syl Byrne, IT Manager Remuneration Services, A Post, explains: "To ensure that our business-critical application is future-proof, we needed to move to a platform that Microsoft will support for the foreseeable future." A Post therefore decided to migrate STREAMS to the VB.NET platform, an object-orientated programming language. This strategy would protect its investment for the next 10 years by creating a secure, scalable

3G E-LEARNING

KNOWLEDGE CHECK

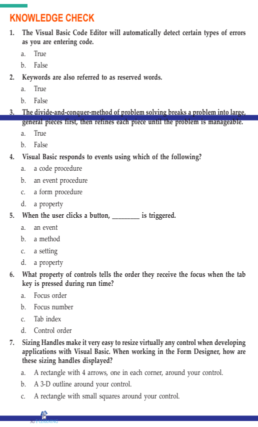
This is given to the students for progress check at the end of each chapter.

REVIEW QUESTIONS

This section is to analyze the knowledge and ability of the reader.

REFERENCES

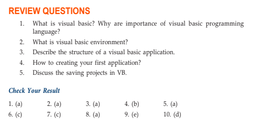
References refer those books which discuss the topics given in the chapters in almost same manner.



KNOWLEDGE CHECK

1. **The Visual Basic Code Editor will automatically detect certain types of errors as you are entering code.**
   a. True
   b. False
2. **Keywords are also referred to as reserved words.**
   a. True
   b. False
3. **The divide-and-conquer-method of problem solving breaks a problem into large, general pieces first, then refines each piece until the problem is manageable.**
   a. True
   b. False
4. **Visual Basic responds to events using which of the following?**
   a. a code procedure
   b. an event procedure
   c. a form procedure
   d. a property
5. **When the user clicks a button, ________ is triggered.**
   a. an event
   b. a method
   c. a setting
   d. a property
6. **What property of controls tells the order they receive the focus when the tab key is pressed during run time?**
   a. Focus order
   b. Focus number
   c. Tab index
   d. Control order
7. **Sizing Handles make it very easy to resize virtually any control when developing applications with Visual Basic. When working in the Form Designer, how are these sizing handles displayed?**
   a. A rectangle with 4 arrows, one in each corner, around your control.
   b. A 3-D outline around your control.
   c. A rectangle with small squares around your control.

3G E-LEARNING

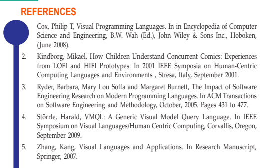


REVIEW QUESTIONS

1. What is visual basic? Why are importance of visual basic programming language?
2. What is visual basic environment?
3. Describe the structure of a visual basic application.
4. How to creating your first application?
5. Discuss the saving projects in VB.

Check Your Result

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



REFERENCES

Cox, Philip T, Visual Programming Languages. In in Encyclopedia of Computer Science and Engineering, B.W. Wah (Ed.), John Wiley & Sons Inc., Hoboken, (June 2008).
2. Kindborg, Mikael, How Children Understand Concurrent Comics: Experiences from LOFI and HIFI Prototypes. In 2001 IEEE Symposia on Human-Centric Computing Languages and Environments , Stresa, Italy, September 2001.
3. Ryder, Barbara, Mary Lou Soffa and Margaret Burnett, The Impact of Software Engineering Research on Modern Programming Languages. In ACM Transactions on Software Engineering and Methodology, October, 2005. Pages 431 to 477.
4. Störrle, Harald, VMQL: A Generic Visual Model Query Language. In IEEE Symposium on Visual Languages/Human Centric Computing, Corvallis, Oregon, September 2009.
5. Zhang, Kang, Visual Languages and Applications. In Research Manuscript, Springer, 2007.

TABLE OF CONTENTS

PREFACE

C++ is an object-oriented programming language. It is an extension to C programming. C++ is a general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming. It has imperative, object-oriented and generic programming features. C++ runs on lots of platform like Windows, Linux, Unix, Mac etc. It is also popular in communications and gaming. It is used in many other industries: health care, finances, and even defense. Any program that was developed in the original C language can easily be moved into C++ without any major modifications. Because C++ offers flexibility, programmers are able to create powerful constructs and introduce new conceptual objects and abstract applications. As a result, C++ allows programmers to directly control and manipulate hardware resources to produce high functioning programs.

Organization of the Book

This edition is systematically divided into eight chapters. This book is designed to equip with C++ programming fundamentals including objects and classes, language reference, inheritance, templates, exceptions, and static class members.

Chapter 1 introduces the basics of C++, such as OOPs Concept, syntax and structure of C++, data types and variables in C++. Operators and size of operators in C++ is also covered in this chapter.

Chapter 2 starts with the features of C++.The difference between C and C++ is also given in this chapter, including variables declaration.

Chapter 3 focuses on overloading used to avoid redundant code where the same method name is used multiple times but with a different set of parameters. Overloading provides code clarity, eliminates complexity, and enhances runtime performance.

Chapter 4 focuses on inheritance concept that allows programmers to define a class in terms of another class, which makes creating and maintaining application easier. When writing a new class, instead of writing new data member and member functions all over again, programmers can make a bonding of the new class with the old one that the new class should inherit the members of the existing class.

Chapter 5 takes a look on polymorphism in C++. It illustrates the concept and importance of polymorphism. Implementing polymorphism in c++ and other applications of polymorphism are also given.

Chapter 6 is intended to focus on exception handling in C++ Programming. In handled exceptions, execution of the program will resume at a designated block of code, called a catch block, which encloses the point of throwing in terms of program execution.

Chapter 7 focuses on I/O Streams. One of the great strengths of C++ is its I/O system, IO Streams. The second thing you may notice is that the word "stream" is used an awful lot. At its most basic, I/O in C++ is implemented with streams.

Chapter 8 is Control Flow. It explains branching or conditional structure. Moreover, it describes iterative or looping structure.

CHAPTER

1

BASICS OF C++

"C++ protects against accident, not against fraud."

–Bjarne Stroustrup,

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Concept of C++
2. Discuss about OOPs Concept Basics
3. Focus on syntax and structure of C++ program
4. Define the data types in C++
5. Variables in C++
6. Understand the Operators in C++
7. Discuss about sizeof operator in C++
8. Describe the Loop Type

```
 *
 * @var boolean
 */
define('PSI_INTERNAL_XML', false);

if (version_compare("5.2", PHP_VERSION, ">")) {
    die("PHP 5.2 or greater is required!!!");
}
if (!extension_loaded("pcre")) {
    die("phpSysInfo requires the pcre extension to php in order to w
        properly.");
}

require_once APP_ROOT.'/includes/autoloader.inc.php';

// Load configuration
require_once APP_ROOT.'/config.php';

if (!defined('PSI_CONFIG_FILE') || !defined('PSI_DEBUG')) {
    $tpl = new Template("/templates/html/error_config.html");
    echo $tpl->fetch();
    die();
```

INTRODUCTION

C++ is a general-purpose, object-oriented programming language. It was created by Bjarne Stroustrup at Bell Labs circa 1980. C++ is very similar to C (invented by Dennis

Ritchie in the early 1970s). C++ is so compatible with C that it will probably compile over 99% of C programs without changing a line of source code. Though C++ is a lot of well-structured and safer language than C as it OOPs based.

Some computer languages are written for a specific purpose. Like, Java was initially devised to control toasters and some other electronics. C was developed for programming OS. Pascal was conceptualized to teach proper programming techniques. But C++ is a general-purpose language. It well deserves the widely acknowledged nickname "Swiss Pocket Knife of Languages."

1.1 CONCEPT OF C++

C++ fully supports object-oriented programming, including the four pillars of object-oriented development −

- Encapsulation
- Data hiding
- Inheritance
- Polymorphism

Standard Libraries

Standard C++ consists of three important parts −

- The core language giving all the building blocks including variables, data types and literals, etc.
- The C++ Standard Library giving a rich set of functions manipulating files, strings, etc.
- The Standard Template Library (STL) giving a rich set of methods manipulating data structures, etc.

The ANSI Standard

The ANSI standard is an attempt to ensure that C++ is portable; that code you write for Microsoft's compiler will compile without errors, using a compiler on a Mac, UNIX, a Windows box, or an Alpha.

The ANSI standard has been stable for a while, and all the major C++ compiler manufacturers support the ANSI standard.

Learning C++

The most important thing while learning C++ is to focus on concepts.

The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones.

C++ supports a variety of programming styles. You can write in the style of Fortran, C, Smalltalk, etc., in any language. Each style can achieve its aims effectively while maintaining runtime and space efficiency.

1.1.1 Use of C++

C++ is used by programmers to create computer software. It is used to create general systems software, drivers for various computer devices, software for servers and software for specific applications and also widely used in the creation of video games.

C++ is used by many programmers of different types and coming from different fields. C++ is mostly used to write device driver programs, system software, and applications that depend on direct hardware manipulation under real-time constraints. It is also used to teach the basics of object-oriented features because it is simple and is also used in the fields of research. Also, many primary user interfaces and system files of Windows and Macintosh are written using C++. So, C++ is really a popular, strong and frequently used programming language of this modern programming era.

- C++ is used by hundreds of thousands of programmers in essentially every application domain.
- C++ is being highly used to write device drivers and other software that rely on direct manipulation of hardware under real-time constraints.
- C++ is widely used for teaching and research because it is clean enough for successful teaching of basic concepts.
- Anyone who has used either an Apple Macintosh or a PC running Windows has indirectly used C++ because the primary user **interfaces** of these systems are written in C++.

Keyword

A class can implement more than one **interface**. An interface can extends another interface or interfaces (more than one interface).

1.1.2 Feature of Object oriented C++

- The main focus remains on data rather than procedures.
- Object-oriented programs are segmented into parts called objects.
- Data structures are designed to categorize the objects.
- Data member and functions are tied together as a data structure.
- Data can be hidden and cannot be accessed by external functions using access specified.
- Objects can communicate among themselves using functions.
- New data and functions can be easily added anywhere within a program whenever required.
- Since this is an object-oriented programming language, it follows a bottom up approach, i.e. the execution of codes starts from the main which resides at the lower section and then based on the member function call the working is done from the classes.

The object-oriented approach is a recent concept among programming paradigms and has various fields of progress. Object-oriented programming is a technique that provides a way of modularizing programs by creating memory area as a partition for both data and functions that can further be used as a template to create copies of modules on demand.

1.1.3 Benefits of C++ over C Language

The major difference being OOPS concept, C++ is an object oriented language whereas C language is a procedural language. Apart from this there are many other features of C++ which gives this language an upper hand on C laguage.

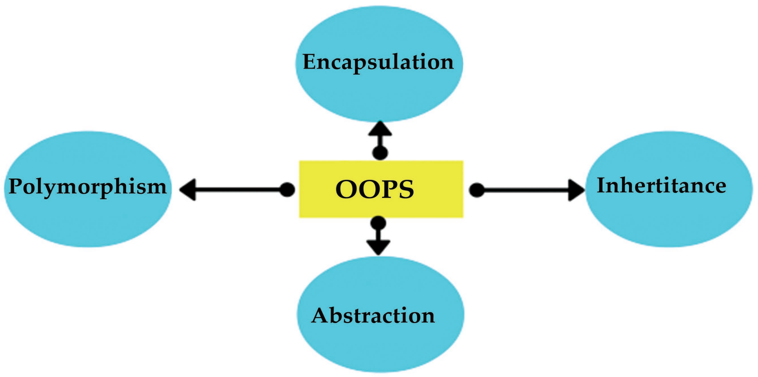
Following features of C++ makes it a stronger language than C,

- There is Stronger Type Checking in C++.
- All the OOPS features in C++ like Abstraction, Encapsulation, Inheritance etc makes it more worthy and useful for programmers.
- C++ supports and allows user defined operators (i.e Operator Overloading) and function overloading is also supported in it.
- Exception Handling is there in C++.
- The Concept of Virtual functions and also Constructors and Destructors for Objects.
- Inline Functions in C++ instead of Macros in C language. Inline functions make complete function body act like Macro, safely.

- Variables can be declared anywhere in the program in C++, but must be declared before they are used.

1.2 OOPS CONCEPT BASICS

Object Oriented programming is a programming style that is associated with the concept of Class, Objects and various other concepts revolving around these two, like Inheritance, Polymorphism, Abstraction, Encapsulation etc.



1.2.1 Access Control in Classes

Now before studying how to define class and its objects, lets first quickly learn what are access specifies. Access specifies in C++ class defines the access control rules. C++ has 3 new keywords introduced, namely,

- public
- private
- protected

These access specifies are used to set boundaries for availability of members of class be it data members or member functions

Access specifies in the program, are followed by a colon. You can use either one, two or all 3 specifies in the same class to set different boundaries for different class members. They change the boundary for all the declarations that follow them.

Public

Public, means all the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. Hence there are chances that they might change them. So the key members must not be declared public.

Private

Private keyword, means that no one can access the class members declared private outside that class. If someone tries to access the private member, they will get a compile time error. By default class variables and member functions are private.

Keyword

A **member function** of a class is a function that has its definition or its prototype within the class definition like any other variable

Protected

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

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

1.3 SYNTAX AND STRUCTURE OF C++ PROGRAM

Here we will discuss one simple and basic C++ program to print "Hello this is C++" and its structure in parts with details and uses.

1.3.1 First C++ program

```
#include <iostream.h>
using namespace std;
int main()
{
cout << "Hello this is C++";
}
```

Header files are included at the beginning just like in C program. Here iostream is a header file which provides us with input & output streams. Header files contained predeclared function libraries, which can be used by users for their ease.

Using namespace std, tells the compiler to use standard namespace. Namespace collects identifiers used for class, object and variables. NameSpace can be used by two ways in a program, either by the use of using statement at the beginning, like we did in above mentioned program or by using name of namespace as prefix before the identifier with scope resolution (::) operator.

Example : std::cout << "A";

main(), is the function which holds the executing part of program its return type is int.

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

Comments

For single line comments, use // before mentioning comment, like

```
cout<<"single line";   // This is single line comment
```

For multiple line comment, enclose the comment between /* and */

```
/*this is
  a multiple line
  comment */
```

Using Classes

Classes name must start with capital letter, and they contain data variables and member functions. This is a mere introduction to classes,

```
class Abc
{
 int i;           //data variable
 void display()        //Member Function
  {
   cout<<"Inside Member Function";
  }
}; // Class ends here
```

```
int main()
{
 Abc obj;  // Creatig Abc class's object
 obj.display();  //Calling member function using class object
}
```

This is how class is defined, its object is created and the member functions are used.

Remember

Variables can be declared anywhere in the entire program, but must be declared, before they are used. Hence, we don't need to declare variable at the start of the program.

1.4 DATA TYPES IN C++

They are used to define type of variables and contents used. Data types define the way you use storage in the programs you write. Data types can be built in or abstract.

Built in Data Types

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

User defined or Abstract data types

These are the type, that user creates as a class. In C++ these are classes where as in C it was implemented by structures.

1.4.1 Basic Built in types

char	for character storage (1 byte)
int	for integral number (2 bytes)
float	single precision floating point (4 bytes)
double	double precision floating point numbers (8 bytes)

Example :

```
char a = 'A';           // character type
int a = 1;              // integer type
```

```
float a = 3.14159;      // floating point type
double a = 6e-4;        // double type (e is for exponential)
```

Other Built in types

bool	Boolean (True or False)
void	Without any Value
wchar_t	Wide Character

1.4.2 Enum as Data type

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

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

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

Modifiers

Specifiers modify the meanings of the predefined built-in data types and expand them to a much larger set. There are four data type modifiers in C++, they are:

- long
- short
- signed
- unsigned

Below mentioned are some important points you must know about the modifiers,

- **long** and **short** modify the maximum and minimum values that a data type will hold.
- A plain int must have a minimum size of **short**.
- Size hierarchy : short int < int < long int

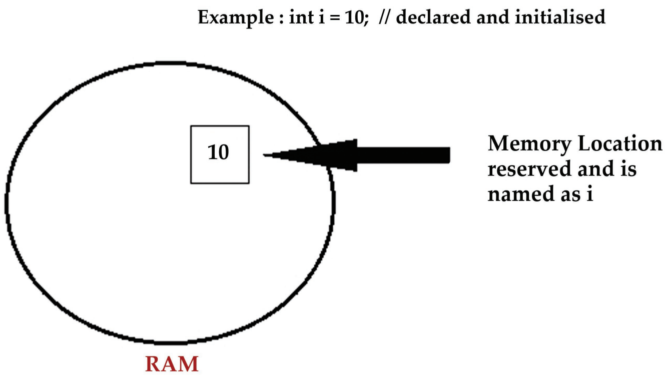
Keyword

Compilers are a type of translator that support digital devices, primarily computers.

- Size hierarchy for floating point numbers is : float < double < long double
- **long float** is not a legal type and there are no **short floating point** numbers.
- **Signed** types includes both positive and negative numbers and is the default type.
- **Unsigned**, numbers are always without any sign, that is always positive.

1.5 VARIABLES IN C++

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


Example : int i = 10; // declared and initialised
10
Memory Location reserved and is named as i
RAM

1.5.1 Basic types of Variables

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

bool	For variable to store boolean values(True or False)
char	For variables to store character types.
int	for variable with integral values

float and double are also types for variables with large and floating point values	

1.5.2 Declaration and Initialization

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

Example :

```
int i;      // declared but not initialised
char c;
int i, j, k;  // Multiple declaration
```

Initialization means assigning value to an already declared variable,

```
int i;   // declaration
i = 10;  // initialization
```

Initialization and declaration can be done in one single step also,

```
int i=10;         //initialization and declaration in same step
int i=10, j=11;
```

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

```
int i,j;
i=10;
j=20;
int j=i+j;   //compile time error, cannot redeclare a variable in same scope
```

1.5.3 Scope of Variables

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

- Global Variables
- Local variables

Global variables

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

Example : Only declared, not initialized

```
include <iostream>
using namespace std;
int x;                  // Global variable declared
int main()
{
 x=10;                  // Initialized once
 cout <<"first value of x = "<< x;
 x=20;                  // Initialized again
 cout <<"Initialized again with value = "<< x;
}
```

Local Variables

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

Example :

```
include <iostream>
using namespace std;
```

```
int main()
{
 int i=10;
 if(i<20)        // if condition scope starts
  {
    int n=100;   // Local variable declared and initialized
  }              // if condition scope ends
 cout << n;      // Compile time error, n not available here
}
```

Some Special types of Variable

There are also some special keywords, to impart unique characteristics to the variables in the program. Following two are mostly used,

- **Final** - Once initialized, its value can't be changed.
- **Static** - These variables holds their value between function calls.

Example :

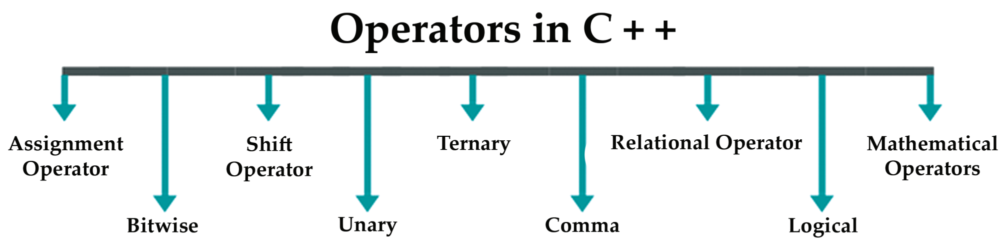
```
#include <iostream.h>
using namespace std;
int main()
{
 final int i=10;
 static int y=20;
}
```

1.6 OPERATORS IN C++

Operators are special type of functions that takes one or more arguments and produces a new value. For example: addition (+), subtraction (-), multiplication (*) etc., are all operators. Operators are used to perform various operations on variables and constants.

Keyword

C Programming **Operators**. C programming has various operators to perform tasks including arithmetic, conditional and bitwise operations. ... An operator is a symbol which operates on a value or a variable.



1.6.1 Types of operators

- Assignment Operator
- Mathematical Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Shift Operators
- Unary Operators
- Ternary Operator
- Comma Operator

Assignment Operator (=)

Operates '=' is used for assignment, it takes the right-hand side (called rvalue) and copy it into the left-hand side (called lvalue). Assignment operator is the only operator which can be overloaded but cannot be inherited.

Mathematical Operators

There are operators used to perform basic mathematical operations. Addition (+) , subtraction (-) , diversion (/) multiplication (*) and modulus (%) are the basic mathematical operators. Modulus operator cannot be used with floating-point numbers.

C++ and C also use a shorthand notation to perform an operation and assignment at same type. *Example,*

int x=10;

x += 4 // will add 4 to 10, and hence assign 14 to X.

x -= 5 // will subtract 5 from 10 and assign 5 to x.

Relational Operators

These operators establish a relationship between operands. The relational operators are: less than (<) , grater thatn (>) , less than or equal to (<=), greater than equal to (>=), equivalent (==) and not equivalent (!=).

You must notice that assignment operator is (=) and there is a relational operator, for equivalent (==). These two are different from each other, the assignment operator assigns the value to any variable, whereas equivalent operator is used to compare values, like in if-else conditions,

Example

```
int x = 10;  //assignment operator
x=5;         // again assignment operator
if(x == 5)   // here we have used equivalent relational operator,
for comparison
{
 cout <<"Successfully compared";
}
```

Remember

If two statement are connected using AND operator, the validity of both statements will be considered, but if they are connected using OR operator, then either one of them must be valid. These operators are mostly used in loops (especially while loop) and in Decision making.

Logical Operators

The logical operators are AND (&&) and OR (||). They are used to combine two different expressions together.

Bitwise Operators

There are used to change individual bits into a number. They work with only integral data types like char, int and long and not with floating point values.

- Bitwise AND operators &
- Bitwise OR operator |
- And bitwise XOR operator ^
- And, bitwise NOT operator ~

They can be used as shorthand notation too, & = , |= , ^= , ~= etc.

Shift Operators

Shift Operators are used to shift Bits of any variable. It is of three types,

- Left Shift Operator <<
- Right Shift Operator >>
- Unsigned Right Shift Operator >>>

Unary Operators

These are the operators which work on only one operand. There are many unary operators, but increment ++ and decrement -- operators are most used.

Other Unary Operators: address of &, dereference *, **new** and **delete**, bitwise not ~, logical not !, unary minus - and unary plus +.

Ternary Operator

The ternary if-else ? : is an operator which has three operands.

int a = 10;

a > 5 ? cout << "true" : cout << "false"

Comma Operator

This is used to separate variable names and to separate expressions. In case of expressions, the value of last expression is produced and used.

Example :

int a,b,c; // variables declaration using comma operator

a=b++, c++; // a = c++ will be done.

1.7 SIZEOF OPERATOR IN C++

sizeOf is also an operator not a function, it is used to get information about the amount of memory allocated for data types & Objects. It can be used to get size of user defined data types too.

sizeOf operator can be used with and without parentheses. If you apply it to a variable you can use it without parentheses.

cout << sizeOf(double);   //Will print size of double

int x = 2;

int i = sizeOf x;

1.7.1 typedef Operator

typedef is a keyword used in C language to assign alternative names to existing types. Its mostly used with user defined data types, when names of data types get slightly complicated. Following is the general syntax for using typedef,

typedef existing_name alias_name

Lets take an example and see how typedef actually works.

typedef unsigned long ulong;

The above statement define a term ulong for an unsigned long type. Now this ulong identifier can be used to define unsigned long type variables.

ulong i, j;

typedef and Pointers

typedef can be used to give an alias name to pointers also. Here we have a case in which use of typedef is beneficial during **pointer declaration**.

In Pointers * binds to the right and not the left.

int* x, y ;

By this declaration statement, we are actually declaring x as a pointer of type int, whereas y will be declared as a plain integer.

typedef int* IntPtr ;

IntPtr x, y, z;

But if we use typedef like in above example, we can declare any number of pointers in a single statement

Keyword

In computer science, a **pointer** is a programming language object that stores the memory address of another value located in computer memory.

1.8 LOOP TYPE

A Computer is used for performing many Repetitive types of tasks The Process of Repeatedly performing tasks is known as looping .The Statements in the block may be Executed any number of times from Zero to Up to the Condition is True. The Loop is that in which a task is repeated until the condition is true or we can say in the loop will Executes all the statements are until the given condition is not to be false.

These are generally used for repeating the statements. In this There is Either Entry Controlled loop or as Exit

Controlled Loop We know that before Execution of Statements all Conditions are Checked these are Performed by Entry Controlled Loops Which First Checks Condition And in Exit Controlled Loop it Checks Condition for Ending Loop Whether given Condition is False or not if a Loop First Checks Condition For Execution then it is called as Entry Controlled Loop and if a Loop Checks Condition after the Execution of Statement then they are Called as Exit Controlled Loops.

In The loop generally there are three basic operations are performed

- 1) Initialization
- 2) Condition check
- 3) Increment

Playing with loops makes programming fun. Before we try to understand loop, you should be thorough with all the previous topics of C++.

Suppose, we have to print the first 10 natural numbers.

One way to do this is to print the first 10 natural numbers individually using cout. But what if you are asked to print the first 100 natural numbers! You can easily do this with the help of loops.

- 1) while
- 2) do-while
- 3) for

all these are used for performing the repetitive tasks until the given condition is not true.

Did You Know?

C++ is considered to be an intermediate-level language, as it encapsulates both high- and low-level language features. Initially, the language was called "C with classes" as it had all the properties of the C language with an additional concept of "classes." However, it was renamed C++ in 1983.

1.8.1 While

While Loop is Known as Entry Controlled Loop because in The while loop first we initialize the value of variable or Starting point of Execution and then we check the condition and if the condition is true then it will execute the statements and then after it increments or decrements the value of a variable. But in the while loop if a Condition is false then it will never Executes the Statement So that For Execution, this is must that the Condition must be true.

Let's first look at the syntax of while loop.

```
while(condition)
{
    statement(s)
}
```

while loop checks whether the condition written in ( ) is true or not. If the condition is true, the statements written in the body of the while loop i.e., inside the braces { } are executed. Then again the condition is checked, and if found true, again the statements in the body of the while loop are executed. This process continues until the condition becomes false.

An example will make this clear.

```
#include <iostream>
int main(){
    using namespace std;
    int n = 1;
    while( n <= 10){
        cout << n << endl;
        n++;
    }
    return 0;
}
```

Output

In our example, firstly, we assigned a value 1 to a variable 'n'.

while(n <= 10) - checks the condition 'n <= 10'. Since the value of n is 1 which is less than 10, the statements within the braces { } are executed.

The value of 'n' i.e. 1 is printed and n++ increases the value of 'n' by 1. So, now the value of 'n' becomes 2.

Now, again the condition is checked. This time also 'n <= 10' is true because the value of 'n' is 2. So, again the value of 'n' i.e., 2 gets printed and the value of 'n' will be increased to 3.

When the value of 'n' becomes 10, again the condition 'n <= 10' is true and 10 gets printed for the tenth time. Now, n++ increases the value to 'n' to 11.

This time, the condition 'n <= 10' becomes false and the program terminates.

Quite interesting. Isn't it !

The following animation will also help you to understand the while loop.

```
int a = 1;
while (a < 4 )
{
cout << " Hello World" << endl;
a ++;                                           a becomes 4
}
```

Output

```
Hello World
Hello World
Hello World
```

Let's see one more example of while loop

```
#include <iostream>
int main(){
        using namespace std;
        int choice = 1;
        while( choice == 1 ){

                int a;

                cout << "Enter a number to check even or odd" << endl;
                cin >> a;               //input number

                //check whether number is even or odd

                if( a%2 == 0 ){
                        cout << "Your number is even" << endl;
                }
                else{
                        cout << "Your number is odd" << endl;
                }

                cout << "Want to check more : 1 for yes and 0 for no" << endl;
```

```
            cin >> choice;
        }

        cout << “I hope you checked all your numbers” << endl;

        return 0;
}
```

Output

The loop will run until the value of ‘choice’ becomes other than ‘1’. So, for the first time, it will run since the value of ‘choice’ is ‘1’. Then it will perform the codes inside the loop. At last, it will ask the user whether he wants to check more or not. This can change the value of variable ‘choice’ and may terminate the loop.

Initially, the value of ‘choice’ was 1, so, the condition of while got satisfied and codes inside it got executed. We were asked to give the value of choice and we gave 1 again. Things repeated and after that, we gave choice a value of 0. Now, the condition of while was not satisfied and the loop terminated.

1.8.2 Do while

This is Also Called as Exit Controlled Loop we know that in The while loop the condition is check before the execution of the program but if the condition is not true then it will not execute the statements so for this purpose we use the do while loop in this first it executes the statements and then it increments the value of a variable and then last it checks the condition So in this either the condition is true or not it Execute the statement at least one time.

This is another kind of loop. This is just like while and for loop but the only difference is that the code in its body is executed once before checking the conditions.

Syntax of do...while loop is:

```
do{
   statement(s)
}
while( condition );
```

Consider the same example of printing the first 10 natural numbers for which we wrote programs using while and for loop. Now, let’s write its program using do... while loop.

```
#include <iostream>
int main(){

    using namespace std;
    int n = 1;
    do{
        cout << n << endl;
        n++;
    }while( n <= 10 );
    return 0;
}
```

Output

Let's try to understand this.

At first, the statements inside the body of loop (i.e., within the braces { } following do) are executed. This will print the value of 'n' i.e., 1 and n++ increments the value of 'n' by 1. So now, the value of 'n' becomes 2.

Once the code inside the braces { } is executed, condition 'n <= 10' is checked. Since the value of 'n' is 2, so the condition is satisfied.

Again the code inside the body of loop is executed and the value of 'n' becomes 2. When the value of 'n' is 10 and 10 is printed, n++ increases the value of 'n' to 11. After this, the condition becomes false and the loop terminates.

As you have seen, in do while loop, codes inside the loop got executed for the first time without checking any condition and then it started checking the condition from the second time.

Nesting of loops

Like 'if/else' we can also use one loop inside another. This is called nesting of loop. See this example to make it clear.

```
#include <iostream>
int main(){
        using namespace std;
        int i;
        int j;
        for(i = 12; i <= 14; i++){              /*outer loop*/

                cout << "Table of " << i << endl;

                for(j = 1; j <= 10; j++){           /*inner loop*/

                        cout << i << "*" << j << "=" << (i*j) <<
endl;

                }
        }
        return 0;
}
```

Output

When the first for loop is executed, the value of i is 12 and "Table of 12" gets printed.

Now coming to the second loop, the value of j is 1 and thus 12*1 = 12 gets printed.

In the second iteration of the inner for loop, while the value of i is still 12, the value of j becomes 2 and thus 12 * 2 = 24 gets printed.

In the last iteration of the inner for loop, the value of i is still 12 and the value of j becomes 10, thus printing 12 * 10 = 120.

Now after all the iterations of the inner for loop are complete, there will be the second iteration of the outer for loop increasing the value of i to 13 and printing Table of 13. Again the inner for loop will be iterated with i equals 13.

We can use any loop inside any other loop according to the requirement. In the above example, we used one for loop .inside another

Keyword

A loop is used for executing a block of statements repeatedly until a particular condition is satisfied.

Infinite Loop

There may exist some loops which can iterate or occur infinitely. These are called Infinite Loop. These loops occur infinitely because their condition is always true. We can make an infinite loop by leaving its conditional expression empty (this is one of the many possible ways). When the conditional expression is empty, it is assumed to be true. Let's see an example on how to make a **for loop** infinite.

```
<include <iostream#
int main(){
        using namespace std;
        for( ; ; ){
                cout << "This loop will never end" << endl;
        }
        return 0;
}
```

Output

```
I told you, it's fun!
```

1.8.3 For loop

In This loop all the basic operations like initialization, condition checking and incrementing or decrementing all these are performed in only one line. This is similar to the while loop for performing its execution but only different in its syntax.

Another type of loop is for loop.

Let's go to our first example in which we printed first 10 natural numbers using while loop. We can also do this with for loop.

Let's look at the syntax of for loop.

```
for(initialization; condition; propagation)
{
    statement(s)
}
```

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

Output

```
1
2
3
4
5
6
7
8
9
10
```

Now let's see how for loop works.

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

n=1 - This step is used to **initialize** a **variable** and is executed **first and only once**. Here, **'n'** is assigned a value **1**.

n<=10 - This is a **condition** which is evaluated. If the condition is true, the statements written in the body of the loop are executed. If it is false, the statement just after the for loop is executed. This is similar to the condition we used in ‹while› loop which was being checked again and again.

n++ - This is executed after the code in the body of the for loop has been executed. In this example, the value of ‹n› increases by 1 every time the code in the body of for loop executes. There can be any expression here which you want to run after every loop.

In the above example, firstly, **'n=1'** assigns a value **1** to **'n'**.

Then the condition **'n<=10'** is checked. Since the value of ‹n› is 1, therefore the code in the body of for loop is executed and thus the current value of ‹n› i.e., 1 gets printed.

Once the codes in the body of for loop are executed, step **n++** is executed which increases the value of ‹n› by 1. So now the value of **'n'** is **2**.

Again the condition 'n<=10' is checked which is true because the value of 'n' is 2. Again codes in the body of for loop are executed and 2 gets printed and then the value of 'n' is again incremented.

When the value of 'n' becomes 10, the condition 'n <= 10' is true and 10 gets printed. Now, when n++ increases the value to 'n' to 11, the condition 'n<=10' becomes false and the loop terminates.

Don't you think it's just a different form of while loop? Yes, it is actually.

Let's see the example of adding 10 numbers.

Let's see the example of adding 10 numbers.

```
#include <iostream>
int main(){

    using namespace std;
    int sum = 0, i, n;

    for(i = 0; i < 10; i++){

        cout << "Enter number" << endl;
        cin >> n;

        sum = sum + n;

    }
    cout << "Sum is " << sum << endl;

    return 0;

}
```

Output

Initially, the value of the variable sum is 0.

In the first iteration, the value of n is entered 4 and thus the value of sum becomes 4 since sum = sum + n (i.e. sum = 0 + n).

In the second iteration, the value of sum is 4 and we entered the value of n as 3. Therefore, the expression sum = sum + n gets evaluated as sum = 4 + 3, thus making the value of sum as 7.

In this way, this loop will add all the 10 numbers entered by the user.

There are other ways also to write program of for loop.

The first example of for loop in which we printed the first 10 natural numbers can also be written in other ways which are:

```
int n = 1;
```

```
for( ; n <= 10; n++)
{
    cout << n << endl;
}
```

Another way is shown below.

```
int n;
for( n = 1; n <= 10; )
{
    cout << n << endl;
    n++;
}
```

It means that we can also write the for loop by skipping one or more of its three statements (initialization, condition, propagation) as done above.

CASE STUDY

REAL-WORLD APPLICATIONS OF C++

1. Games:

C++ overrides the complexities of 3D games, optimizes resource management and facilitates multiplayer with networking. The language is extremely fast, allows procedural programming for CPU intensive functions and provides greater control over hardware, because of which it has been widely used in development of gaming engines. For instance, the science fiction game Doom 3 is cited as an example of a game that used C++ well and the Unreal Engine, a suite of game development tools, is written in C++.

2. Graphic User Interface (GUI) based applications:

Many highly used applications, such as Image Ready, Adobe Premier, Photoshop and Illustrator, are scripted in C++.

3. Web Browsers:

With the introduction of specialized languages such as PHP and Java, the adoption of C++ is limited for scripting of websites and web applications. However, where speed and reliability are required, C++ is still preferred. For instance, a part of Google's back-end is coded in C++, and the rendering engine of a few open source projects, such as web browser Mozilla Firefox and email client Mozilla Thunderbird, are also scripted in the programming language.

4. Advance Computations and Graphics:

C++ provides the means for building applications requiring real-time physical simulations, high-performance image processing, and mobile sensor applications. Maya 3D software, used for integrated 3D modeling, visual effects and animation, is coded in C++.

5. Database Software:

C++ and C have been used for scripting MySQL, one of the most popular database management software. The software forms the backbone of a variety of database-based enterprises, such as Google, Wikipedia, Yahoo and YouTube etc.

6. Operating Systems:

C++ forms an integral part of many of the prevalent operating systems including Apple's OS X and various versions of Microsoft Windows, and the erstwhile Symbian mobile OS.

7. Enterprise Software:

C++ finds a purpose in banking and trading enterprise applications, such as those deployed by Bloomberg and Reuters. It is also used in development of advanced software, such as flight simulators and radar processing.

8. Medical and Engineering Applications:

Many advanced medical equipments, such as MRI machines, use C++ language for scripting their software. It is also part of engineering applications, such as high-end CAD/CAM systems.

9. Compilers:

A host of compilers including Apple C++, Bloodshed Dev-C++, Clang C++ and MINGW make use of C++ language. C and its successor C++ are leveraged for diverse software and platform development requirements, from operating systems to graphic designing applications. Further, these languages have assisted in the development of new languages for special purposes like C#, Java, PHP, Verilog etc.

SUMMARY

- C++ is a middle-level programming language developed by Bjarne Stroustrup starting in 1979 at Bell Labs.
- The ANSI standard is an attempt to ensure that C++ is portable; that code you write for Microsoft's compiler will compile without errors, using a compiler on a Mac, UNIX, a Windows box, or an Alpha.
- The purpose of learning a programming language is to become a better programmer; that is, to become more effective at designing and implementing new systems and at maintaining old ones.
- C++ is used by programmers to create computer software. It is used to create general systems software, drivers for various computer devices, software for servers and software for specific applications and also widely used in the creation of video games.
- Object Oriented programming is a programming style that is associated with the concept of Class, Objects and various other concepts revolving around these two, like Inheritance, Polymorphism, Abstraction, Encapsulation etc.
- Classes name must start with capital letter, and they contain data variables and member functions.
- Variable is the name of memory location allocated by the compiler depending upon the data type of the variable.
- All the variables have their area of functioning, and out of that boundary they don't hold their value, this boundary is called scope of the variable.
- Operators are special type of functions that takes one or more arguments and produces a new value. For example: addition (+), subtraction (-), multiplication (*) etc., are all operators. Operators are used to perform various operations on variables and constants.
- sizeOf is also an operator not a function, it is used to get information about the amount of memory allocated for data types & Objects. It can be used to get size of user defined data types too.
- A Computer is used for performing many Repetitive types of tasks The Process of Repeatedly performing tasks is known as looping .The Statements in the block may be Executed any number of times from Zero to Up to the Condition is True.

KNOWLEDGE CHECK

1. **#include<userdefined.h>**

   Which of the following is the correct syntax to add the header file in the C++ program?

   a. #include<userdefined>

   b. #include “userdefined.h”

   c. <include> “userdefined.h”

   d. Both A and B

2. **Which of the following is the correct syntax to print the message in C++ language?**

   a. cout <<”Hello world!”;

   b. Cout << Hello world! ;

   c. Out <<”Hello world!;

   d. None of the above

3. **Which of the following is the correct identifier?**

   a. $var_name

   b. VAR_123

   c. varname@

   d. None of the above

4. **Which of the following is the address operator?**

   a. @

   b. #

   c. &

   d. %

5. **Which of the following features must be supported by any programming language to become a pure object-oriented programming language?**

   a. Encapsulation

   b. Inheritance

   c. Polymorphism

   d. All of the above

6. **The programming language that has the ability to create new data types is called___.**

   a. Overloaded

   b. Encapsulated

c. Reprehensible

d. Extensible

7. **Which of the following is the original creator of the C++ language?**

a. Dennis Ritchie

b. Ken Thompson

c. Bjarne Stroustrup

d. Brian Kernighan

8. **Which of the following is the correct syntax to read the single character to console in the C++ language?**

a. Read ch()

b. Getline vh()

c. get(ch)

d. Scanf(ch)

9. **Which of the following statements is correct about the formal parameters in C++?**

a. Parameters with which functions are called

b. Parameters which are used in the definition of the function

c. Variables other than passed parameters in a function

d. Variables that are never used in the function

10. **The C++ language is ______ object-oriented language.**

a. Pure Object oriented

b. Not Object oriented

c. Semi Object-oriented or Partial Object-oriented

d. None of the above

REVIEW QUESTIONS

1. Define the use of c++.
2. What are the benefits of c++ over c language?
3. Explain the syntax and structure of c++ program.
4. Focus on declaration and initialization.
5. Describe the loop type with example.

Check Your Result

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

REFERENCES

1. A. B. Webber. Modern Programming Languages: A Practical Introduction (Franklin, Beedle & Associates, 2003).
2. B. Meyer. Object-Oriented Software Construction (2nd Edition) (Prentice Hall, 2000).
3. D. Parnas. “The Secret History of Information Hiding” (Software Pioneers: Contributions To Software Engineering, Springer-Verlag New York, 2002).
4. D. R. Musser, G. J. Derge, and A. Saini. STL Tutorial and Reference Guide, 2nd Edition (Addison-Wesley, 2001).
5. D. Vandevoorde and N. Josuttis. C++ Templates (Addison-Wesley, 2003).
6. K. Henney. “C++ Patterns: Executing Around Sequences” (EuroPLoP 2000 proceedings).
7. S. Meyers. “How Non-Member Functions Improve Encapsulation” (C/C++ Users Journal, 18(2), February 2000).

CHAPTER 2

LANGUAGE FEATURES

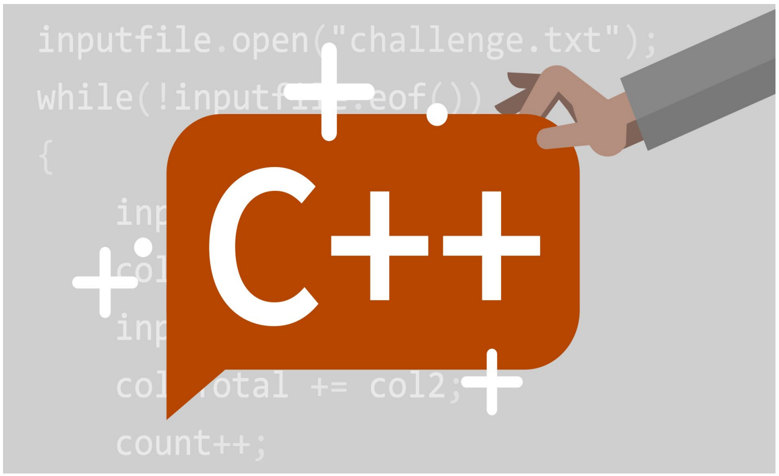
"*Within C++, there is a much smaller and cleaner language struggling to get out.*"

–Bjarne Stroustrup

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Focus on concept of C++ features
2. Difference between C and C++
3. Describe the variables declaration in C++

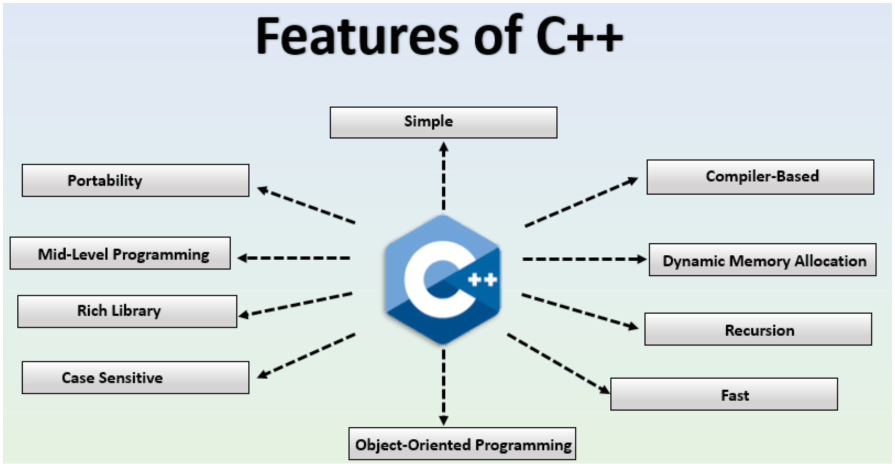


```
inputfile.open("challenge.txt");
while(!inputfile.eof())
{
```

INTRODUCTION

C++ is a general-purpose programming language that was developed with the intention to improve C language and include an object-oriented paradigm. Object-Oriented

Programming is a programming in which we design and build our application or program based on the object. Objects are instances (variables) of class. It is an imperative and a compiled language. With C++ inheritance, one object obtains all the properties and behaviours of its parent object automatically. It lets you reuse, extend or modify the attributes and behaviours defined in other class.

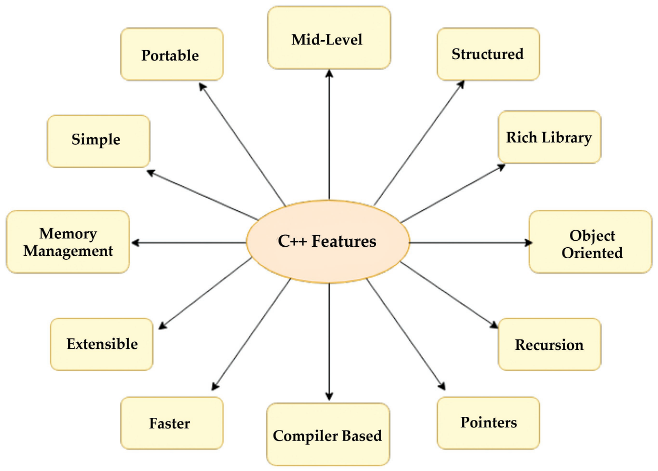


Being a middle-level language, C++ gives it the ability to develop low-level (drivers, kernels) and even higher-level software like games, GUI, desktop apps, etc. The main motive of creating C++ programming was to add object orientation to the C programming language. The main changes that were made are object-oriented programming methodology, namespace feature, operator overloading, error & exception handling. The other motive of OOP (object-oriented programming) is to try to understand the whole system in the form of classes & objects.

With Object-oriented programming, data does not flow freely around the system. It gets bound more closely to the functions that operate on it and gets protected from the coincidental change from outside functions. OOP breaks down a complex program into objects and then builds data and functions around these objects.

2.1 CONCEPT OF C++ FEATURES

C++ is object oriented programming language. It provides a lot of **features** that are given below.



1) Simple

C++ is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Machine Independent or Portable

Unlike assembly language, c programs can be executed in many machines with little bit or no change. But it is not platform-independent.

3) Mid-level programming language

C++ is also used to do low level programming. It is used to develop system applications such as kernel, driver etc. It also supports the feature of high level language. That is why it is known as mid-level language.

4) Structured programming language

C++ is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

5) Rich Library

C++ provides a lot of inbuilt functions that makes the development fast.

6) Memory Management

It supports the feature of dynamic memory allocation. In C++ language, we can free the allocated memory at any time by calling the free() function.

7) Speed

The compilation and execution time of C++ language is fast.

8) Pointer

C++ provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array etc.

9) Recursion

In C++, we can call the function within the function. It provides code reusability for every function.

10) Extensible

C++ language is extensible because it can easily adopt new features.

11) Object Oriented

C++ is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

12) Compiler based

C++ is a compiler based programming language, it means without compilation no C++ program can be executed. First we need to compile our program using compiler and then we can execute our program.

Keyword

Hybrid language is a programming language that supports multiple approaches of writing programs.

2.2 DIFFERENCE BETWEEN C AND C++

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

Difference between C and C++

C	C++
C was developed by Dennis Ritchie between 1969 and 1973 at AT&T Bell Labs.	C++ was developed by Bjarne Stroustrup in 1979 with C++'s predecessor "C with Classes".
When compared to C++, C is a subset of C++.	C++ is a superset of C. C++ can run most of C code while C cannot run C++ code.

C supports procedural programming paradigm for code development.	C++ supports both procedural and object oriented programming paradigms; therefore C++ is also called a hybrid language.
C does not support object oriented programming; therefore it has no support for polymorphism, encapsulation, and inheritance.	Being an object oriented programming language C++ supports polymorphism, encapsulation, and inheritance.
In C (because it is a procedural programming language), data and functions are separate and free entities.	In C++ (when it is used as object oriented programming language), data and functions are encapsulated together in form of an object. For creating objects class provides a blueprint of structure of the object.
In C, data are free entities and can be manipulated by outside code. This is because C does not support information hiding.	In C++, Encapsulation hides the data to ensure that data structures and operators are used as intended.
C, being a procedural programming, it is a function driven language.	While, C++, being an object oriented programming, it is an object driven language.
C does not support function and operator overloading.	C++ supports both function and operator overloading.
C does not allow functions to be defined inside structures.	In C++, functions can be used inside a structure.
C does not have namespace feature.	C++ uses NAMESPACE which avoid name collisions. A namespace is a declarative region that provides a scope to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to prevent name collisions that can occur especially when your code base includes multiple libraries. All identifiers at namespace scope are visible to one another without qualification. Identifiers outside the namespace can access the members by using the fully qualified name for each identifier.

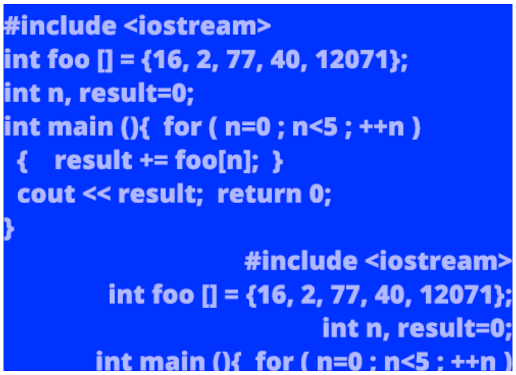
C uses functions for input/output. For example scanf and printf.	C++ uses objects for input output. For example cin and cout.
C does not support reference variables.	C++ supports reference variables.
C has no support for virtual and friend functions.	C++ supports virtual and friend functions.
C provides malloc() and calloc() functions for dynamic memory allocation, and free() for memory de-allocation.	C++ provides new operator for memory allocation and delete operator for memory de-allocation.
C does not provide direct support for error handling (also called exception handling)	C++ provides support for exception handling. Exceptions are used for “hard” errors that make the code incorrect.

Hope you have enjoyed reading differences between C and C++. This comparison of C and C++ explains feature-wise difference between both programming languages. Please do write us if you have any suggestion/comment or come across any error on this page.

Remember

The most common is that C++ is an advanced C and one should have in-depth understanding of the latter language before moving to the former one. However, this is .just a myth

2.2.1 Key Differences between C and C++



```
#include <iostream>
int foo [] = {16, 2, 77, 40, 12071};
int n, result=0;
int main (){  for ( n=0 ; n<5 ; ++n )
 {   result += foo[n];  }
 cout << result;  return 0;
}
                #include <iostream>
        int foo [] = {16, 2, 77, 40, 12071};
                        int n, result=0;
       int main (){  for ( n=0 ; n<5 ; ++n )
```

There are several misconceptions that prevail among developers regarding the programming languages, C and C++.

Enlisted below are the major set of differences between the programming languages, C and C++.

Sl.No	Basis Of Distinction	C	C++
1	**Nature Of Language**	C is a structural or procedural type of programming language.	C++ is an object-oriented programming language and supports Polymorphism, Abstract Data Types, Encapsulation, among others. Even though C++ derives basic syntax from C, it cannot be classified as a structural or a procedural language.
2	**Point Of Emphasis**	C lays emphasis on the steps or procedures that are followed to solve a problem.	C++ emphasizes the objects and not the steps or procedures. It has higher abstraction level.
3	**Compatibility With Overloading**	C does not support function overloading.	C++ supports function overloading, implying that one can have name of functions with varying parameters.
4	**Data Types**	C does not provide String or Boolean data types. It supports primitive & built-in data types.	C++ provides Boolean or String data types. It supports both user-defined and built-in data types.
5	**Compatibility With Exception Handling**	C does not support Exception Handling directly. It can be done through some other functions.	C++ supports Exception Exception:Handling can be done through try & catch block.
6	**Compatibility With Functions**	C does not support functions with default arrangements	C++ supports functions with default arrangements.
7	**Compatibility With Generic Programming**	C is not compatible	C++ is compatible with generic programming
8	**Pointers And References**	C supports only Pointers	C++ supports both pointers and references.
9	**Inline Function**	C does not have inline function.	C++ has inline function.
10	**Data Security**	In C programming language, the data is unsecured.	Data is hidden in C++ and is not accessible to external functions. Hence, is more secure
11	**Approach**	C follows the top-down approach.	C++ follows the bottom-up approach.

12	**Functions For Standard Input And Output**	scanf and printf	cin and cout
13	**Time Of Defining Variables**	In C, variable has to be defined at the beginning, in the function.	Variable can be defined anywhere in the function.
14	**Namespace**	Absent	Present
15	**Division Of Programs**	The programs in C language are divided into modules and functions.	The programs are divided into classes and functions in the C++ programming language.
16	**File Extension**	.C	.CPP
17	**Function And Operator Overloading**	Absent	Present
18	**Mapping**	Mapping between function and data is complicated in C.	Mapping between function and data can be done easily using 'Objects'.
19	**Calling Of Functions**	main() function can be called through other functions.	main() function cannot be called through other functions.
20	**Inheritance**	Possible	Not possible
21	**Functions Used For Memory Allocation And Deallocation**	malloc() and calloc for Memory Allocation and free() function for Deallocation.	New and delete operators are used for Memory Allocation and Deallocation in C++.
22	**Influences**	C++, C#, Objective-C, PHP, Perl, BitC, Concurrent C, Java, JavaScript, Perl, csh, awk, D, Limbo	C#, PHP, Java, D, Aikido, Ada 95
23	**Influenced By**	B (BCPL,CPL), Assembly, ALGOL 68,	C, ALGOL 68, Simula, Ada 83, ML, CLU
24	**Level of Language**	Mid-level	High-level
25	**Classes**	C uses structures thereby, giving freedom to use internal design elements	class and structures

:Some important differences between the C and C++ structures

- **Member functions inside structure**: Structures in C cannot have member functions inside structure but Structures in C++ can have member functions along with data members.
- **Direct Initialization:** We cannot directly initialize structure data members in C but we can do it in C++.

```
// CPP program to initialize data member in c++
#include <iostream>
using namespace std;

struct Record {
    int x = 7;
};

// Driver Program
int main()
{
    Record s;
    cout << s.x << endl;
    return 0;
}
// Output
7 //
```

Keyword

Default initialization applies when no initializer is specified at all, or when a class member is omitted from the member initialization list.

- **Using struct keyword:** In C, we need to use struct to declare a struct variable. In C++, struct is not necessary. For example, let there be a structure for Record. In C, we must use "struct Record" for Record variables. In C++, we need not use struct and using 'Record' only would work.
- **Static Members:** C structures cannot have static members but is allowed in C++.

```
// C++ program with structure static member
```

```
struct Record {
    static int x;
};

// Driver program
int main()
{
    return 0;
}
```

This will generate an error in C but no error in C++.

- **Constructor creation in structure**: Structures in C cannot have constructor inside structure but Structures in C++ can have Constructor creation.

```
// CPP program to initialize data member in c++
#include <iostream>
using namespace std;

struct Student {
    int roll;
    Student(int x)
    {
        roll = x;
    }
};

// Driver Program
int main()
{
    struct Student s(2);
    cout << s.roll;
    return 0;
}
// Output
// 2
```

sizeof operator: This operator will generate 0 for an empty structure in C whereas 1 for an empty structure in C++.

```
// C program to illustrate empty structure
#include <stdio.h>

// empty structure
struct Record {
};

// Driver program
int main()
{
    struct Record s;
    printf("%d\n", sizeof(s));
    return 0;
}

Output in C:
0
Output in C++:
1
```

- **Data Hiding:** C structures do not allow concept of **Data hiding** but is permitted in C++ as C++ is an object oriented language whereas C is not.
- **Access Modifiers:** C structures do not have access modifiers as these modifiers are not supported by the language. C++ structures can have this concept as it is inbuilt in the language.

Keyword

Data hiding is a software development technique specifically used in object-oriented programming (OOP) to hide internal object details (data members).

2.3 VARIABLES DECLARATION IN C++

In C++ variable is used to store data in a memory location, which can be modified or used in the program during program execution.

Variables play a major role in constructing a program, storing values in memory and dealing with them. Variables are required in various functions of every program.

When we check for conditions to execute a block of statements, variables are required. Again for iterating or repeating a block of the statement(s) several times, a counter variable is set along with a condition, or simply if we store the age of an employee, we need an integer type variable. So in every respect, the variable is used.

What is Variables?

Variables are used in C++ where you will need to store any type of values within a program and whose value can be changed during the program execution. These variables can be declared in various ways each having different memory requirements and storing capability. Variables are the name of memory locations that are allocated by **compilers** and the allocation is done based on the data type used for declaring the variable.

Keyword

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses.

A variable definition means that the programmer writes some instructions to tell the compiler to create the storage in a memory location.

The syntax for defining variables is:

Syntax:

```
data_type variable_name;
    data_type variable_name, variable_name, variable_name;
```

Here data_type means the valid C++ data type which includes int, float, double, char, wchar_t, bool and variable list is the lists of variable names to be declared which is separated by commas.

Example

```
/* variable definition */int    width, height, age;
char   letter;
float  area;
double d;
```

2.3.1 Variables initialization in C++

Variables are declared in the above example but none of them has been assigned any value. Variables can be initialized and the initial value can be assigned along with their declaration.

Syntax

data_type variable_name = value;

Example

```
/* variable definition and initialization */int    width, height=5, age=32;
char   letter='A';
float  area;
double d;
    /* actual initialization */width = 10;
    area = 26.5;
```

There is some rules must be in your knowledge to work with C++ variables.

2.3.2 Rules of Declaring variable in C++

- A variable name can consist of Capital letters A-Z, lowercase letters a-z, digits 0-9, and the underscore character.
- The first character must be a letter or underscore.
- Blank spaces cannot be used in variable names.
- Special characters like #, $ are not allowed.
- C++ keywords cannot be used as variable names.
- Variable names are case-sensitive.
- A variable name can be consisting of 31 characters only if we declare a variable more than 1 characters compiler will ignore after 31 characters.
- Variable type can be bool, char, int, float, double, void or wchar_t.

Example

```
#include <iostream>
using namespace std;
```

```
int main()
{
    int x = 5;
    int y = 2;
    int Result;
    Result = x * y;
    cout << Result;
}
```

Another program showing how **Global variables** are declared and used within a program:

```
#include <iostream>
     using namespace std;

     // Global Variable declaration:
     int x, y;
     float f;

     int main()
     {
         // Local variable
         int tot;
         float f;
         x = 10;
         y = 20;
         tot = x + y;

         cout << tot;
         cout << endl;
         f = 70.0 / 3.0;
         cout << f;
         cout << endl;
{
```

Keyword

In computer programming, a **global variable** is a variable with global scope, meaning that it is visible (hence accessible) throughout the program, unless shadowed.

2.3.3 Scope of Variables in C++

In general, scope is defined as the extent up to which something can be worked with. In programming also scope of a variable is defined as the extent of the program code within which the variable can we accessed or declared or worked with. There are mainly two types of variable scopes as discussed below:

Local Variables

Variables defined within a function or block are said to be local to those functions.

- Anything between '{' and '}' is said to inside a block.
- Local variables do not exist outside the block in which they are declared, i.e. they cannot be accessed or used outside that block.
- Declaring local variables: Local variables are declared inside a block

```
// CPP program to illustrate
// usage of local variables
#include<iostream>
using namespace std;

void func()
{
    // this variable is local to the
    // function func() and cannot be
    // accessed outside this function
    int age=18;
}

int main()
{
    cout<<"Age is: "<<age;

    return 0;
}
```

Output:

Error: age was not declared in this scope

The above program displays an error saying "age was not declared in this scope". The variable age was declared within the function func () so it is local to that function and not visible to portion of program outside this function.

Rectified Program: To correct the above error we have to display the value of variable age from the function func() only. This is shown in the below program:

```
// CPP program to illustrate
// usage of local variables
#include<iostream>
using namespace std;

void func()
{
    // this variable is local to the
    // function func() and cannot be
    // accessed outside this function
    int age=18;
    cout<<age;
}

int main()
{
    cout<<"Age is: ";
    func();

    return 0;
}
```

Output:

```
Age is: 18
```

Global Variables

As the name suggests, Global Variables can be accessed from any part of the program. They are available throughout the life time of a program.

They are declared at the top of the program outside all of the functions or blocks.

Declaring global variables: Global variables are usually declared outside of all of the functions and blocks, at the top of the program. They can be accessed from any portion of the program.

```
// CPP program to illustrate
// usage of global variables
#include<iostream>
using namespace std;

// global variable
int global = 5;

// global variable accessed from
// within a function
void display()
{
    cout<<global<<endl;
}

// main function
int main()
{
    display();

    // changing value of global
    // variable from main function
    global = 10;
    display();
}
```

Output:

```
5
10
```

What if there exists a local variable with the same name ?as that of global variable inside a function

If there is a variable inside a function with the same name as that of a global variable and if the function tries to access the variable with that name, then which variable will be given precedence? **Local variable** or Global variable? Look at the below program to understand the question:

Keyword

A **local variable** is a variable which is either a variable declared within the function or is an argument passed to a function.

```
// CPP program to illustrate
// scope of local variables
// and global variables together
#include<iostream>
using namespace std;

// global variable
int global = 5;

// main function
int main()
{
    // local variable with same
    // name as that of global variable

    int global = 2;
    cout << global << endl;
}
```

Look at the above program. The variable “global” declared at the top is global and stores the value 5 whereas that declared within main function is local and stores a value 2. So, the question is when the value stored in the variable named “global” is printed from the main function then what will be the output? 2 or 5?

Usually when two variable with same name are defined then the compiler produces a compile time error. But if the variables are defined in different scopes then the compiler allows it.

Whenever there is a local variable defined with same name as that of a global variable then the compiler will give precedence to the local variable.

Here in the above program also, the local variable named "global" is given precedence. So the output is 2.

How to access a global variable when there is a local variable with same name?

To solve this problem we will need to use the scope resolution operator. Below program explains how to do this with the help of scope resolution operator.

```
// C++ program to show that we can access a global
// variable using scope resolution operator:: when
// there is a local variable with same name
#include<iostream>
using namespace std;

// Global x
int x = 0;

int main()
{
  // Local x
  int x = 10;
  cout << "Value of global x is " << ::x;
  cout<< "\nValue of local x is " << x;
  return 0;
}
```

Output:

```
Value of global x is 0
Value of local x is 10
```

Remember

If storage classis "static", initialization value must be provided.

2.3.4 Variable declaration syntax in C/C++ language

A variable is the name of memory blocks, whose value can be changed at any time (runtime), we can declare a variable by using following syntax:

[storage_class] data_type variable_name [=value];

Here, [storage-class] and [=value] are optional.

Keyword

Garbage value is a waste or unused values which are available in memory during declaration of variables.

Declaration Example 1:

auto int age = 10;

age is an automatic integer variable (learn more about auto (automatic): storage classes in C language), it's initial value is 10, which can be changed at any time. (i.e. you may change the value of age at any time).

Declaration Example 2:

int age=10;

If we do not specify storage class, variable's default storage class will be an automatic (auto). Thus, declaration example 1 and 2 are same.

Declaration Example 3:

int age;

This declaration is same as declaration example 1 and 2 without initial value, age will have either 0 or **garbage value** based on the different compilers.

Other examples:

float percentage; //a float variable

char gender; //a character variable

//an array of characters, maximum number of characters will be 100

char name [100];

int marks [5]; //an array of integers to store marks of 5 subjects

2.3.5 Variable Declaration Rule in C++

To declare any variable in C++ you need to follow rules and regulation of C++ Language, which is given below;

- Every variable name should start with alphabets or underscore (_).
- No spaces are allowed in variable declaration.
- Except **underscore (_)** no special symbol are allowed in the middle of the variable declaration.
- Maximum length of variable is 8 characters depend on compiler and operation system.
- Every variable name always should exist in the left hand side of assignment operator.
- No keyword should access variable name.

Note: In a c program variable name always can be used to identify the input or output data.

Did You Know?

C++ is standardized by the International Organization for Standardization (ISO), with the latest standard version ratified and published by ISO in December 2017 as ISO/IEC 14882:2017 (informally known as C++17)

ROLE MODEL

ALAN KAY: BEST KNOWN FOR HIS PIONEERING WORK ON OBJECT-ORIENTED PROGRAMMING AND WINDOWING GRAPHICAL USER INTERFACE DESIGN.

BIOGRAPHY

Alan Kay, (born May 17, 1940, Springfield, Mass., U.S.), American computer scientist and winner of the 2003 A.M. Turing Award, the highest honor in computer science, for his contributions to object-oriented programming languages, including Smalltalk.

Kay received a doctorate in computer science from the University of Utah in 1969. In 1972 he joined Xerox Corporation's Palo Alto Research Center and continued work on the first object-oriented programming language (Smalltalk) for educational applications. He contributed to the development of Ethernet, laser printing, and client-server architecture. He left Xerox in 1983 and became a fellow at Apple Computer, Inc. (now Apple Inc.), in 1984. His design of a graphical user interface for operating systems (OS) was used in Apple's Mac OS and later in Microsoft Corporation's Windows OS. He was a fellow at the Walt Disney Company (1996–2001) and the Hewlett-Packard Company (2002–05).

Recent work and Recognition

From 1981 to 1984, Kay was Atari's Chief Scientist. He became an Apple Fellow in 1984. Following the closure of the company's Advanced Technology Group in 1997, he was recruited by his friend Bran Ferren, head of research and development at Disney, to join Walt Disney Imagineering as a Disney Fellow. He remained there until Ferren left to start Applied Minds Inc with Imagineer Danny Hillis, leading to the cessation of the Fellows program. In 2001, he founded Viewpoints Research Institute, a non-profit organization dedicated to children, learning, and advanced software development. For its first ten years, Kay and his Viewpoints group were based at Applied Minds in Glendale, California, where he and Ferren continued to work together on various projects. Kay was also a Senior Fellow at Hewlett-Packard until HP disbanded the Advanced Software Research Team on July 20, 2005.

Kay taught a Fall 2011 class, "Powerful Ideas: Useful Tools to Understand the World", at New York University's Interactive Telecommunications Program (ITP) along with full-time ITP faculty member Nancy Hechinger. The goal of the class was to devise new forms of teaching/learning based on fundamental, powerful concepts rather than traditional rote learning.

Squeak, Etoys, and Croquet

In December 1995, while still at Apple, Kay collaborated with many others to start the open source Squeak version of Smalltalk, and he continues to work on it. As part of this effort, in November 1996, his team began research on what became the Etoys system. More recently he started, along with David A. Smith, David P. Reed, Andreas Raab, Rick McGeer, Julian Lombardi and Mark McCahill, the Croquet Project, an open source networked 2D and 3D environment for collaborative work.

Tweak

In 2001, it became clear that the Etoy architecture in Squeak had reached its limits in what the Morphic interface infrastructure could do. Andreas Raab was a researcher working in Kay's group, then at Hewlett-Packard. He proposed defining a "script process" and providing a default scheduling mechanism that avoids several more general problems. The result was a new user interface, proposed to replace the Squeak Morphic user interface in the future. Tweak added mechanisms of islands, asynchronous messaging, players and costumes, language extensions, projects, and tile scripting. Its underlying object system is class-based, but to users (during programming) it acts like it is prototype-based. Tweak objects are created and run in Tweak project windows.

Awards and Honors

Alan Kay has received many awards and honors. Among them:

2001: UdK 01-Award in Berlin, Germany for pioneering the GUI; J-D Warnier Prix D'Informatique; NEC C&C Prize

2002: Telluride Tech Festival Award of Technology in Telluride, Colorado

2003: ACM Turing Award "For pioneering many of the ideas at the root of contemporary object-oriented programming languages, leading the team that developed Smalltalk, and for fundamental contributions to personal computing."

2004: Kyoto Prize; Charles Stark Draper Prize with Butler W. Lampson, Robert W. Taylor and Charles P. Thacker

2012: UPE Abacus Award awarded to individuals who have provided extensive support and leadership for student-related activities in the computing and information disciplines,

SUMMARY

- C++ is object oriented programming language.
- The major difference between C and C++ is that C is a procedural programming language and does not support classes and objects, while C++ is a combination of both procedural and object oriented programming language; therefore C++ can be called a hybrid language.
- Structures in C cannot have constructor inside structure but Structures in C++ can have Constructor creation.
- In C++ variable is used to store data in a memory location, which can be modified or used in the program during program execution.
- Variables are used in C++ where you will need to store any type of values within a program and whose value can be changed during the program execution.
- Variables can be initialized and the initial value can be assigned along with their declaration.
- In general, scope is defined as the extent up to which something can be worked with. In programming also scope of a variable is defined as the extent of the program code within which the variable can we accessed or declared or worked with.

KNOWLEDGE CHECK

1. **What are the actual parameters in C++?**
   a. Parameters with which functions are called
   b. Parameters which are used in the definition of a function
   c. Variables other than passed parameters in a function
   d. Variables that are never used in the function
2. **What are the formal parameters in C++?**
   a. Parameters with which functions are called
   b. Parameters which are used in the definition of the function
   c. Variables other than passed parameters in a function
   d. Variables that are never used in the function
3. **Which function is used to read a single character from the console in C++?**
   a. cin.get(ch)
   b. getline(ch)
   c. read(ch)
   d. scanf(ch)
4. **Which function is used to write a single character to console in C++?**
   a. cout.put(ch)
   b. cout.putline(ch)
   c. write(ch)
   d. printf(ch)
5. **What are the escape sequences?**
   a. Set of characters that convey special meaning in a program
   b. Set of characters that whose use are avoided in C++ programs
   c. Set of characters that are used in the name of the main function of the program
   d. Set of characters that are avoided in cout statements
6. **Which of the following escape sequence represents carriage return?**
   a. \r
   b. \n
   c. \n\r
   d. \c

7. **Which of the following escape sequence represents tab?**
   a. \t
   b. \t\r
   c. \b
   d. \a
8. **Who created C++?**
   a. Bjarne Stroustrup
   b. Dennis Ritchie
   c. Ken Thompson
   d. Brian Kernighan
9. **Which of the following is called insertion/put to operator?**
   a. <<
   b. >>
   c. >
   d. <
10. **Which of the following is called extraction/get from operator?**
    a. <<
    b. >>
    c. >
    d. <

REVIEW QUESTIONS

1. What is variable declaration and initialization?
2. What are variables in C++?
3. What are variables List C++ rules for variable naming?
4. What is a declaration in C++?
5. How to declare variables in c++?
6. What if we want to access global variable when there is a local variable with same name?

Check Your Result

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

REFERENCES

1. Abrahams, David; Gurtovoy, Aleksey (2005). C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley. ISBN 0-321-22725-5.
2. Alexandrescu, Andrei (2001). Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley. ISBN 0-201-70431-5.
3. Alexandrescu, Andrei; Sutter, Herb (2004). C++ Design and Coding Standards: Rules and Guidelines for Writing Programs. Addison-Wesley. ISBN 0-321-11358-6.
4. Becker, Pete (2006). The C++ Standard Library Extensions : A Tutorial and Reference. Addison-Wesley. ISBN 0-321-41299-0.
5. Brokken, Frank (2010). C++ Annotations. University of Groningen. ISBN 978-90-367-0470-0. Archived from the original on 28 April 2010. Retrieved 28 April 2010.
6. Dewhurst, Stephen C. (2005). C++ Common Knowledge: Essential Intermediate Programming. Addison-Wesley. ISBN 0-321-32192-8.
7. Information Technology Industry Council (15 October 2003). Programming languages – C++ (Second ed.). Geneva: ISO/IEC. 14882:2003(E).
8. Josuttis, Nicolai M. (2012). The C++ Standard Library, A Tutorial and Reference (Second ed.). Addison-Wesley. ISBN 978-0-321-62321-8.
9. Koenig, Andrew; Moo, Barbara E. (2000). Accelerated C++ – Practical Programming by Example. Addison-Wesley. ISBN 0-201-70353-X.
10. Lippman, Stanley B.; Lajoie, Josée; Moo, Barbara E. (2011). C++ Primer (Fifth ed.). Addison-Wesley. ISBN 978-0-321-71411-4.
11. Meyers, Scott (2005). Effective C++ (Third ed.). Addison-Wesley. ISBN 0-321-33487-6.
12. Stroustrup, Bjarne (2013). The C++ Programming Language (Fourth ed.). Addison-Wesley. ISBN 978-0-321-56384-2.
13. Stroustrup, Bjarne (2014). Programming: Principles and Practice Using C++ (Second ed.). Addison-Wesley. ISBN 978-0-321-99278-9.
14. Sutter, Herb (2001). More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions. Addison-Wesley. ISBN 0-201-70434-X.
15. Sutter, Herb (2004). Exceptional C++ Style. Addison-Wesley. ISBN 0-201-76042-8.
16. Vandevoorde, David; Josuttis, Nicolai M. (2003). C++ Templates: The complete Guide. Addison-Wesley. ISBN 0-201-73484-2.

CHAPTER
3

C++ OVERLOADING (FUNCTION AND OPERATOR)

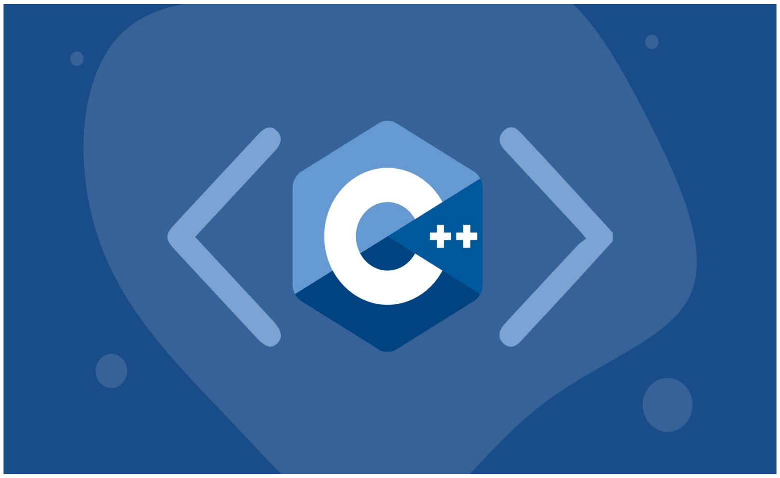
"The problem with using C++ ... is that there's already a strong tendency in the language to require you to know everything before you can do anything. "

–Larry Wall

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Define the concept of overloading
2. Explain the type of overloading
3. Describe the function overloading in C++



INTRODUCTION

Two or more functions can have the same name but different parameters; such functions are called function overloading. C++ has many features, and one of the most important features is function overloading. It is a code with more than one function with the same name having various types of argument lists. This argument list includes the data type of arguments and the sequence of the arguments.

The function overloading feature is used to improve the readability of the code. It is used so that the programmer does not have to remember various function names. If any class has multiple functions with different parameters having the same name, they are said to be overloaded. If we have to perform a single operation with different numbers or types of arguments, we need to overload the function.

In OOP, function overloading is known as a function of polymorphism. The function can perform various operations best on the argument list. It differs by type or number of arguments they hold. By using a different number of arguments or different types of arguments, the function can be redefined.

Keyword

The term **parameter** (sometimes called formal parameter) is often used to refer to the variable as found in the function definition, while argument (sometimes called actual parameter) refers to the actual input supplied at function call.

3.1 CONCEPT OF OVERLOADING

Overloading refers to the ability to use a single identifier to define multiple methods of a class that differ in their input and output parameters. Overloaded methods are generally used when they conceptually execute the same task but with a slightly different set of **parameters**.

Overloading is a concept used to avoid redundant code where the same method name is used multiple times but with a different set of parameters. The actual method that gets called during runtime is resolved at compile time, thus avoiding runtime errors. Overloading provides code clarity, eliminates complexity, and enhances runtime performance.

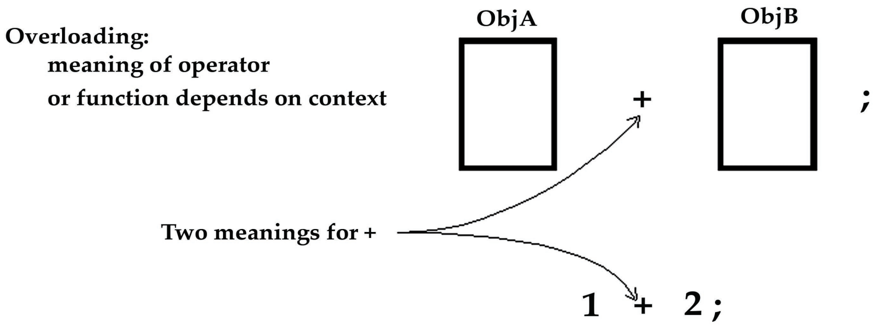
Overloading is used in programming languages that enforce type-checking in function calls during compilation. When a method is overloaded, the method chosen will be selected at compile time. This is not the same as virtual functions where the method is defined at runtime.

Unlike Java, C# allows operators to be overloaded, in addition to methods, by defining static members using the operator keyword. This feature helps to extend and customize the semantics of operators relevant to user-defined types so that they can be used to manipulate object instances with operators.

The overload resolution in C# is the method by which the right function is selected on the basis of arguments passed and the list of candidate function members that have the same name. The different contexts in which the overload resolution is used include:

- Invocation of a method in an expression
- Constructor during object creation
- Indexer access or through an element access and predefined or user-defined operator expression

It is recommended to avoid overloading across inheritance boundaries because it can cause confusion. Overloading can become cumbersome to developers if it is used excessively and with user-defined types as parameters because it can reduce the readability and maintainability of code.



Overloading is the reuse of the same symbol or function name for two or more distinct operations or functions. Whilst this may sound confusing, used carefully it helps to keep code transparent. Overloading can be used with operators and functions.

Operators

FORTRAN used a limited form of operator overloading, for example:-

```
 1 + 2
```

and

```
 1. + 2.
```

Here the plus sign stands for addition, but in the first example, the addition is integer arithmetic whilst in the second it is floating point. As they both perform the same logical operation, and the compiler can be relied on to pick the appropriate physical operation, the overloading of the plus sign is far better than inventing separate symbols for each physical operation. In C++, this concept is pushed much further. It has a large operator set and almost all of them can be overloaded by user functions when involving expressions with objects. So it would be possible to develop a matrix class and then define addition, multiplication and so forth for it. C++ considers [] array and () - function call as operators.

So it is possible to write constructions like:-

```
    MyObj[]
and  MyObj()
```

and have C++ treat this as a call to MyObj's member functions. One place where the function operator is overloaded is in ROOT's TIter class. It not uncommon to see code similar to this:-

```
TIter(MyCollection) next;

...     next() ...
```

In this class the function operator just calls the Next member function so next() is the same as next.Next(), but looks less odd (possibly).

Functions

FORTRAN also overloads generic functions. For example MAX stands for a family of maximum functions, with the compiler selecting the right one depending on context. So:-

```
max( 3, 5 )
```

and

```
max( 3. ,5. )
```

in one case selecting the largest integer and in the other the largest real. Again C++ extends the overloading concept, allowing the user to overload functions. We have already seen a simple example of this in the OO topic **Constructors** & Destructors in which our Track class had 2 constructors:-

```
  Track::Track(Float_t mass, Float_t energy);
  Track::Track();
```

Constructors are frequently overloaded. C++ requires a default i.e. one to be used without any arguments, but many interesting constructors require the user to qualify the initial state. Sometimes a single constructor can serve both functions as C++ allows default arguments. We could declare our track constructor as:-

```
  Track::Track(Float_t mass = 0., Float_t energy = 0.);
```

and then create a track with:-

Track MyTrack(0.135);

and have the compiler call the constructor function with energy = 0. As it stands we would now be in trouble if we did:-

Track MyTrack;

as the compiler has a choice: either use the default constructor, or the other with both arguments set to zero.

Keyword

A **constructor** is automatically called when an object is created.

3.2 TYPE OF OVERLOADING

Overloading: When a single Object has multiple behaviors. Then it is called as Overloading. Overloading is that in which a Single Object has a same name and Provides Many Functions. In Overloading followings things denotes Overloading:-

- When an Object has Same Name.
- Difference is Return type.
- Difference in Function, with Multiple Arguments.
- Difference in Data Type.

3.2.1 Constructor Overloading

Constructor overloading is that in which a Constructor has a same name and has multiple Functions, then it is called as Constructor Overloading. As we Know that Constructor are of Default, Parameterized and Copy Constructors. So that when we are creating a Single Constructor with Multiple Arguments then it is called as Constructor Overloading.

3.2.2 Operator Overloading

As we know that Operators are used for Performing Operations on the Operands. But Each and Every Operator has Some Limitations Means an Operator which is also called as Binary are used for Performing Operations on the two Operands and Unary Operators performs their Operation on the single Operand.

So with the help of Operator Overloading, we can Change the Operation of the Operator. Means With the help of Operators we can Change the Operation of the Operators.

But With the help of Operator Overloading we can Change the behavior of the unary Operator means we can perform Operations means we can Increase or Decrease the values of two or more Operands at a Time. And With the Help of Comparison Operators we can also compare the Two Objects Means all the Data Members of one Object can be compared with the Data Members of the Other Object. Without the help of Operator Overloading this is not possible to compare two Objects. So with the help of Comparison Operators we can compare two Objects.

The help of Binary Operators we can add two Objects Means not only the two Data Members of the Class, This will add all the Data Members of the Class. So Like this Way we can Also Change the Behavior of the Unary Operator Means Unary Operators are used for Performing the .Operation on the Single Operand

3.2.3 Method Overloading

Method Overloading is also called as Function Overloading. Overloading Means a Functions has many Behaviors occurred When in class when a functions has same name but different behaviors A Functions said to be overloaded When :-

- Function has same Name but Different Return Type
- Difference in No of Arguments
- Different Return Type in Arguments

When We Pass a Call for Execution then it will match the Criteria of Function like Number of Arguments and Data types etc.

3.3 FUNCTION OVERLOADING IN C++

You can have multiple definitions for the same function name in the same scope.

Following is the example where same function **print ()** is being used to print different data types −

```
#include <iostream>
using namespace std;


class printData {
   public:
      void print(int i) {
        cout << “Printing int: “ << i << endl;
      }
      void print(double  f) {
        cout << “Printing float: “ << f << endl;
      }
      void print(char* c) {
        cout << “Printing character: “ << c << endl;
      }
};

int main(void) {
   printData pd;

   // Call print to print integer
   pd.print(5);

   // Call print to print float
   pd.print(500.263);

   // Call print to print character
   pd.print(“Hello C++”);

   return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Printing int: 5
Printing float: 500.263
Printing character: Hello C++
```

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters. Function overloading can be considered as an example of **polymorphism** feature in C++.

Keyword

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance

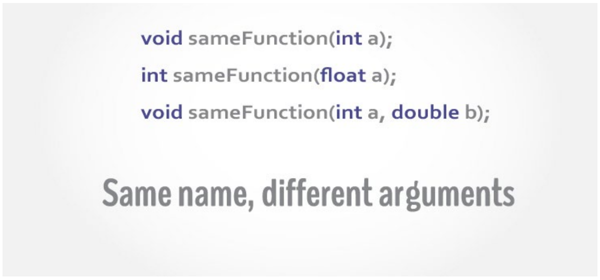
Following is a simple C++ example to demonstrate function overloading.

```
#include <iostream>
using namespace std;

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double  f) {
  cout << " Here is float " << f << endl;
}
void print(char* c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

Output:

```
Here is int 10
Here is float 10.1
Here is char* ten
```

Two or more functions having same name but different argument(s) are known as overloaded functions. You will learn about function overloading with examples.



```
void sameFunction(int a);
int sameFunction(float a);
void sameFunction(int a, double b);
```

Remember

The definition of the function must differ from each other by the types and/ or the number of arguments in the argument list. You cannot overload function declarations that differ only by return type.

Function refers to a segment that groups code to perform a specific task.

In C++ programming, two functions can have same name if number and/or type of arguments passed are different.

These functions having different number or type (or both) of parameters are known as overloaded functions. For example:

```
int test() { }
int test(int a) { }
float test(double a) { }
int test(int a, double b) { }
```

Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.

Notice that, the return type of all these 4 functions are not same. Overloaded functions may or may not have different return type but it should have different argument(s).

```
// Error code
int test(int a) { }
double test(int b){ }
```

The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.

Example 1: Function Overloading

```
#include <iostream>
using namespace std;
```

```
void display(int);
void display(float);
void display(int, float);

int main() {

    int a = 5;
    float b = 5.5;

    display(a);
    display(b);
    display(a, b);

    return 0;
}

void display(int var) {
    cout << "Integer number: " << var << endl;
}

void display(float var) {
    cout << "Float number: " << var << endl;
}

void display(int var1, float var2) {
    cout << "Integer number: " << var1;
    cout << " and float number:" << var2;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

Here, the display() function is called three times with different type or number of arguments.

The return type of all these functions are same but it’s not necessary.

Example 2: Function Overloading

```
// Program to compute absolute value
// Works both for integer and float

#include <iostream>
using namespace std;

int absolute(int);
float absolute(float);

int main() {
    int a = -5;
    float b = 5.5;

    cout << “Absolute value of “ << a << “ = “ << absolute(a) << endl;
    cout << “Absolute value of “ << b << “ = “ << absolute(b);
    return 0;
}

int absolute(int var) {
     if (var < 0)
         var = -var;
    return var;
}
float absolute(float var){
    if (var < 0.0)
        var = -var;
    return var;
}
```

Output

```
Absolute value of -5 = 5
Absolute value of 5.5 = 5.5
```

In the above example, two functions absolute () are overloaded.

Both functions take single argument. However, one function takes integer as an argument and other takes float as an argument.

When absolute() function is called with integer as an argument, this function is called:

```
int absolute(int var) {
     if (var < 0)
          var = -var;
    return var;
}
```

When absolute() function is called with float as an argument, this function is called:

```
float absolute(float var){
    if (var < 0.0)
        var = -var;
    return var;
}
```

Keyword

In programming, **operator overloading,** sometimes termed operator ad hoc polymorphism, is a specific case of polymorphism, where different operators have different implementations depending on their arguments.

3.3.1 Operators Overloading in C++

You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.

Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an **overloaded operator** has a return type and a parameter list.

Box operator+(const Box&);

declares the addition operator that can be used to add two Box objects and returns final Box object. Most overloaded operators may be defined as ordinary non-member functions or as class member functions. In case we define above function as non-member function of a class then we would have to pass two arguments for each operand as follows –

Box operator+(const Box&, const Box&);

Following is the example to show the concept of operator over loading using a member function. Here an object is passed as an argument whose properties will be accessed using this object, the object which will call this operator can be accessed using this operator as explained below –

```
#include <iostream>
using namespace std;

class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }
      void setLength( double len ) {
         length = len;
      }
      void setBreadth( double bre ) {
         breadth = bre;
      }
      void setHeight( double hei ) {
         height = hei;
{

      // Overload + operator to add two Box objects.
      Box operator+(const Box& b) {
         Box box;
         box.length = this->length + b.length;
         box.breadth = this->breadth + b.breadth;
         box.height = this->height + b.height;
```

Did You Know?

In 1983, "C with Classes" was renamed to "C++" (++ being the increment operator in C), adding new features that included virtual functions, function name and operator overloading, references, constants, type-safe free-store memory allocation (new/delete), improved type checking, and BCPL style single-line comments with two forward slashes (//).

```
            return box;
         }

      private:
         double length;       // Length of a box
         double breadth;      // Breadth of a box
         double height;       // Height of a box
};

// Main function for the program
int main() {
   Box Box1;                  // Declare Box1 of type Box
   Box Box2;                  // Declare Box2 of type Box
   Box Box3;                  // Declare Box3 of type Box
   double volume = 0.0;       // Store the volume of a box here

   // box 1 specification
   Box1.setLength(6.0);
   Box1.setBreadth(7.0);
   Box1.setHeight(5.0);

   // box 2 specification
   Box2.setLength(12.0);
   Box2.setBreadth(13.0);
   Box2.setHeight(10.0);

   // volume of box 1
   volume = Box1.getVolume();
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.getVolume();
   cout << "Volume of Box2 : " << volume <<endl;
   // Add two object as follows:
   Box3 = Box1 + Box2;

   // volume of box 3
   volume = Box3.getVolume();
   cout << "Volume of Box3 : " << volume <<endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Volume of Box1 : 210
Volume of Box2 : 1560
Volume of Box3 : 5400
```

Overloadable/Non-overloadableOperators

Following is the list of operators which can be overloaded −

+	-	*	/	%	^
&	|	~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=
|=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

Following is the list of operators, which can not be overloaded −

::	.*	.	?:

Operator Overloading Examples

Here are various operator overloading examples to help you in understanding the concept.

Sr.No	Operators & Example
1	Unary Operators Overloading
2	Binary Operators Overloading
3	Relational Operators Overloading
4	Input/Output Operators Overloading
5	++ and -- Operators Overloading
6	Assignment Operators Overloading
7	Function call () Operator Overloading
8	Subscripting [] Operator Overloading
9	Class Member Access Operator -> Overloading

Example

The following example illustrates how overloading can be used.

```
// function_overloading.cpp
// compile with: /EHsc
#include <iostream>
#include <math.h>

// Prototype three print functions.
int print( char *s );                    // Print a string.
int print( double dvalue );              // Print a double.
int print( double dvalue, int prec );  // Print a double with a
//  given precision.
using namespace std;
int main( int argc, char *argv[] )
{
const double d = 893094.2987;
if( argc < 2 )
    {
// These calls to print invoke print( char *s ).
print( “This program requires one argument.” );
print( “The argument specifies the number of” );
print( “digits precision for the second number” );
print( “printed.” );
exit(0);
```

```
    }

// Invoke print( double dvalue ).
print( d );

// Invoke print( double dvalue, int prec ).
print( d, atoi( argv[1] ) );
}

// Print a string.
int print( char *s )
{
cout << s << endl;
return cout.good();
}

// Print a double in default precision.
int print( double dvalue )
{
cout << dvalue << endl;
return cout.good();
}

// Print a double in specified precision.
//  Positive numbers for precision indicate how many digits
//  precision after the decimal point to show. Negative
//  numbers for precision indicate where to round the number
//  to the left of the decimal point.
int print( double dvalue, int prec )
{
// Use table-lookup for rounding/truncation.
static const double rgPow10[] = {
10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1, 10E0,
10E1,  10E2,  10E3,  10E4, 10E5,  10E6
    };
const int iPowZero = 6;
// If precision out of range, just print the number.
if( prec < -6 || prec > 7 )
return print( dvalue );
// Scale, truncate, then rescale.
dvalue = floor( dvalue / rgPow10[iPowZero - prec] ) *
rgPow10[iPowZero - prec];
cout << dvalue << endl;
return cout.good();
}
```

The preceding code shows overloading of the print function in file scope.

The default argument is not considered part of the function type. Therefore, it is not used in selecting overloaded functions. Two functions that differ only in their default arguments are considered multiple definitions rather than overloaded functions.

Default arguments cannot be supplied for overloaded operators.

3.3.2 Argument Matching

Overloaded functions are selected for the best match of function declarations in the current scope to the arguments supplied in the function call. If a suitable function is found, that function is called. "Suitable" in this context means one of the following:

- An exact match was found.
- A trivial conversion was performed.
- An integral promotion was performed.
- A standard conversion to the desired argument type exists.
- A user-defined conversion (either conversion operator or constructor) to the desired argument type exists.
- Arguments represented by an ellipsis were found.

The compiler creates a set of candidate functions for each argument. Candidate functions are functions in which the actual argument in that position can be converted to the type of the formal argument.

A set of "best matching functions" is built for each argument, and the selected function is the intersection of all the sets. If the intersection contains more than one function, the overloading is ambiguous and generates an error. The function that is eventually selected is always a better match than every other function in the group for at least one argument. If this is not the case (if there is no clear winner), the function call generates an error.

Consider the following declarations (the functions are marked Variant 1, Variant 2, and Variant 3, for identification in the following discussion):

```
Fraction &Add( Fraction &f, long l );       // Variant 1
Fraction &Add( long l, Fraction &f );       // Variant 2
Fraction &Add( Fraction &f, Fraction &f );  // Variant 3

Fraction F1, F2;
```

Consider the following statement:

```
F1 = Add( F2, 23 );
```

The preceding statement builds two sets:

Set 1: Candidate Functions That Have First Argument of Type Fraction	Set 2: Candidate Functions Whose Second Argument Can Be Converted to Type int
Variant 1	Variant 1 (int can be converted to long using a standard conversion)
Variant 3	

Functions in Set 2 are functions for which there are implicit conversions from actual parameter type to **formal parameter** type, and among such functions there is a function for which the "cost" of converting the actual parameter type to its formal parameter type is the smallest.

The intersection of these two sets is Variant 1. An example of an ambiguous function call is:

```
F1 = Add( 3, 6 );
```

The preceding function call builds the following sets:

Set 1: Candidate Functions That Have First Argument of Type int	Set 2: Candidate Functions That Have Second Argument of Type int
Variant 2 (int can be converted to long using a standard conversion)	Variant 1 (int can be converted to long using a standard conversion)

Note that the intersection between these two sets is empty. Therefore, the compiler generates an error message.

For argument matching, a function with *n* default arguments is treated as *n*+1 separate functions, each with a .different number of arguments

The ellipsis (...) acts as a wildcard; it matches any actual argument. This can lead to many ambiguous sets, if you do not design your overloaded function sets with extreme care.

Keyword

When a function is called, the values (expressions) that are passed in the call are called the arguments or actual parameters (both terms mean the same thing). At the time of the call each actual parameter is assigned to the corresponding **formal parameter** in the function definition.

Argument Type Differences

Overloaded functions differentiate between argument types that take different initializers. Therefore, an argument of a given type and a reference to that type are considered the same for the purposes of overloading. They are considered the same because they take the same initializers. For example, max (double, double) is considered the same as max(double &, double &). Declaring two such functions causes an error.

For the same reason, function arguments of a type modified by **const** or volatile are not treated differently than the base type for the purposes of overloading.

However, the function overloading mechanism can distinguish between references that are qualified by **const** and volatile and references to the base type. This makes code such as the following possible:

```
// argument_type_differences.cpp
// compile with: /EHsc /W3
// C4521 expected
#include <iostream>

using namespace std;
class Over {
public:
   Over() { cout << "Over default constructor\n"; }
   Over( Over &o ) { cout << "Over&\n"; }
   Over( const Over &co ) { cout << "const Over&\n"; }
  Over( volatile Over &vo ) { cout << "volatile Over&\n"; }
};
```

```
int main() {
   Over o1;              // Calls default constructor.
   Over o2( o1 );        // Calls Over( Over& ).
   const Over o3;        // Calls default constructor.
   Over o4( o3 );        // Calls Over( const Over& ).
   volatile Over o5;     // Calls default constructor.
   Over o6( o5 );        // Calls Over( volatile Over& ).
}
```

Output

```
Over default constructor
Over&
Over default constructor
const Over&
Over default constructor
volatile Over&
```

Pointers to **const** and volatile objects are also considered different from pointers to the base type for the purposes of overloading.

Argument matching and conversions

When the compiler tries to match actual arguments against the arguments in function declarations, it can supply standard or user-defined conversions to obtain the correct type if no exact match can be found. The application of conversions is subject to these rules:

- Sequences of conversions that contain more than one user-defined conversion are not considered.
- Sequences of conversions that can be shortened by removing intermediate conversions are not considered.

The resultant sequence of conversions, if any, is called the best matching sequence. There are several ways to convert an object of type int to type unsigned long using standard conversions (described in Standard Conversions):

- Convert from int to long and then from long to unsigned long.
- Convert from int to unsigned long.

The first sequence, although it achieves the desired goal, is not the best matching sequence — a shorter sequence exists.

The following table shows a group of conversions, called trivial conversions, which have a limited effect on determining which sequence is the best matching. The instances in which trivial conversions affect choice of sequence are discussed in the list following the table.

Trivial Conversions

Convert from Type	**Convert to Type**
type-name	*type-name* **&**
type-name **&**	*type-name*
type-name **[ ]**	*type-name**
type-name **(** *argument-list* **)**	**(** **type-name* **) (** *argument-list* **)**
type-name	**const** *type-name*
type-name	volatile *type-name*
*type-name**	**const** *type-name**
*type-name**	volatile *type-name**

The sequence in which conversions are attempted is as follows:

- Exact match. An exact match between the types with which the function is called and the types declared in the **function prototype** is always the best match. Sequences of trivial conversions are classified as exact matches. However, sequences that do not make any of these conversions are considered better than sequences that convert:
  - From pointer, to pointer to **const** (type * to **const** type *).
  - From pointer, to pointer to volatile (type * to volatile type *).
  - From reference, to reference to **const** (type **&** to **const** type **&**).
  - From reference, to reference to volatile (type **&** to volatile type **&**).
- Match using promotions. Any sequence not classified as an exact match that contains only integral promotions, conversions from **float** to **double**, and trivial conversions is classified as a match using promotions. Although not as good a match as any exact match, a match using promotions is better than a match using standard conversions.
- Match using standard conversions. Any sequence not classified as an exact match or a match using promotions that contains only standard conversions and trivial conversions is classified as a match using standard conversions. Within this category, the following rules are applied:
  - Conversion from a pointer to a derived class, to a pointer to a direct or indirect base class is preferable to converting to **void** * or **const void** *.

- Conversion from a pointer to a derived class, to a pointer to a base class produces a better match the closer the base class is to a direct base class. Suppose the class hierarchy is as shown in the following figure.

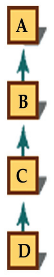


Figure 1: Graph illustrating preferred conversions.

Conversion from type D* to type C* is preferable to conversion from type D* to type B*. Similarly, conversion from type D* to type B* is preferable to conversion from type D* to type A*.

This same rule applies to reference conversions. Conversion from type D& to type C& is preferable to conversion from type D& to type B&, and so on.

This same rule applies to pointer-to-member conversions. Conversion from type T D::* to type T C::* is preferable to conversion from type T D::*to type T B::*, and so on (where T is the type of the member).

The preceding rule applies only along a given path of derivation. Consider the graph shown in the following figure.

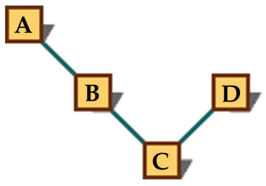


Figure 2: Multiple-inheritance graph illustrating preferred conversions.

Conversion from type C* to type B* is preferable to conversion from type C* to type A*. The reason is that they

Keyword

In computer programming, a **function prototype** or function interface is a declaration of a function that specifies the function's name and type signature (arity, data types of parameters, and return type), but omits the function body.

are on the same path, and B* is closer. However, conversion from type C* to type D* is not preferable to conversion to type A*; there is no preference because the conversions follow different paths.

- Match with user-defined conversions. This sequence cannot be classified as an exact match, a match using promotions, or a match using standard conversions.
- Match with an ellipsis. Any sequence that matches an ellipsis in the declaration is classified as a match with an ellipsis. This is considered the weakest match.

User-defined conversions are applied if no built-in promotion or conversion exists. These conversions are selected on the basis of the type of the argument being matched. Consider the following code:

```
// argument_matching1.cpp
class UDC
{
public:
   operator int()
   {
      return 0;
   }
   operator long();
};

void Print( int i )
{
};

UDC udc;

int main()
{
   Print( udc );
}
```

The available user-defined conversions for class UDC are from type int and type **long**. Therefore, the compiler considers conversions for the type of the object being matched: UDC. A conversion to int exists, and it is selected.

During the process of matching arguments, standard conversions can be applied to both the argument and the result of a user-defined conversion. Therefore, the following code works:

```
    void LogToFile( long l );
    ...
    UDC udc;
    LogToFile( udc );
```

In the preceding example, the user-defined conversion, **operator long**, is invoked to convert udc to type **long**. If no user-defined conversion to type **long** had been defined, the conversion would have proceeded as follows: Type UDC would have been converted to type int using the user-defined conversion. Then the standard conversion from type int to type **long** would have been applied to match the argument in the declaration.

If any user-defined conversions are required to match an argument, the standard conversions are not used when evaluating the best match. This is true even if more than one candidate function requires a user-defined conversion; in such a case, the functions are considered equal. For example:

```
// argument_matching2.cpp
// C2668 expected
class UDC1
{
public:
   UDC1( int );  // User-defined conversion from int.
};

class UDC2
{
public:
   UDC2( long ); // User-defined conversion from long.
};

void Func( UDC1 );
void Func( UDC2 );

int main()
{
    Func( 1 );
}
```

Both versions of Func require a user-defined conversion to convert type int to the class type argument. The possible conversions are:

- Convert from type int to type UDC1 (a user-defined conversion).
- Convert from type int to type **long**; then convert to type UDC2 (a two-step conversion).

Even though the second of these requires a standard conversion, as well as the user-defined conversion, the two conversions are still considered equal.

Remember

The sequence must contain only user-defined conversions, standard conversions, or trivial conversions to be classified as a match with user-defined conversions. A match with user-defined conversions is considered a better match than a match with an ellipsis but not as good a match as a match with standard conversions.

3.3.3 Argument matching and the this pointer

Class member functions are treated differently, depending on whether they are declared as static. Because nonstatic functions have an implicit argument that supplies the pointer, nonstatic functions are considered to have one more argument than static functions; otherwise, they are declared identically.

These nonstatic member functions require that the implied this pointer match the object type through which the function is being called, or, for overloaded operators, they require that the first argument match the object on which the operator is being applied.

Unlike other arguments in overloaded functions, no temporary objects are introduced and no conversions are attempted when trying to match the this pointer argument.

When the – > member-selection operator is used to access a member function, the this pointer argument has a type of class-name * const. If the members are declared as const or volatile, the types are const class-name``* const and volatile class-name * const, respectively.

The member-selection operator works exactly the same way, except that an implicit & (address-of) operator is prefixed to the object name. The following example shows how this works:

```
// Expression encountered in code
```

obj.name

// How the compiler treats it
(&obj)->name

The left operand of the –>* and .* (pointer to member) operators are treated the same way as the . and –> (member-selection) operators with respect to argument matching.

Restrictions

Several restrictions govern an acceptable set of overloaded functions:

- Any two functions in a set of overloaded functions must have different argument lists.
- Overloading functions with argument lists of the same types, based on return type alone, is an error.

Microsoft Specific

You can overload **operator new** solely on the basis of return type — specifically, on the basis of the memory-model modifier specified.

3.3.4 END Microsoft Specific

- Member functions cannot be overloaded solely on the basis of one being static and the other nonstatic.
- typedef declarations do not define new types; they introduce synonyms for existing types. They do not affect the overloading mechanism. Consider the following code:
  - typedef char * PSTR;
  - void Print(char *szToPrint);
  - void Print(PSTR szToPrint);

The preceding two functions have identical argument lists. PSTR is a synonym for type **char** *. In member scope, this code generates an error.

- Enumerated types are distinct types and can be used to distinguish between overloaded functions.
- The types “array of “ and “pointer to” are considered identical for the purposes of distinguishing between overloaded functions. This is true only for singly dimensioned arrays. Therefore, the following overloaded functions conflict and generate an error message:

- void Print(char *szToPrint);
- void Print(char szToPrint[]);

For multiply dimensioned arrays, the second and all succeeding dimensions are considered part of the type. Therefore, they are used in distinguishing between overloaded functions:

```
void Print( char szToPrint[] );
void Print( char szToPrint[][7] );
void Print( char szToPrint[][9][42] );
```

3.3.5 Declaration Matching

Any two function declarations of the same name in the same scope can refer to the same function, or to two discrete functions that are overloaded. If the argument lists of the declarations contain arguments of equivalent types (as described in the previous section), the function declarations refer to the same function. Otherwise, they refer to two different functions that are selected using overloading.

Class scope is strictly observed; therefore, a function declared in a base class is not in the same scope as a function declared in a derived class. If a function in a derived class is declared with the same name as a function in the base class, the derived-class function hides the base-class function instead of causing overloading.

Block scope is strictly observed; therefore, a function declared in file scope is not in the same scope as a function declared locally. If a locally declared function has the same name as a function declared in file scope, the locally declared function hides the file-scoped function instead of causing overloading. For example:

```
// declaration_matching1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void func( int i )
{
    cout << “Called file-scoped func : “ << i << endl;
}

void func( char *sz )
{
   cout << "Called locally declared func : " << sz << endl;
}

int main()
{
   // Declare func local to main.
   extern void func( char *sz );

   func( 3 );   // C2664 Error. func( int ) is hidden.
   func( "s" );
}
```

The preceding code shows two definitions from the function func. The definition that takes an argument of type char * is local to main because of the extern statement. Therefore, the definition that takes an argument of type int is hidden, and the first call to func is in error.

For overloaded member functions, different versions of the function can be given different access privileges. They are still considered to be in the scope of the enclosing class and thus are overloaded functions. Consider the following code, in which the member function Deposit is overloaded; one version is public, the other, private.

The intent of this sample is to provide an Account class in which a correct password is required to perform deposits. This is accomplished using overloading.

Note that the call to Deposit in Account::Deposit calls the private member function. This call is correct because Account::Deposit is a member function and therefore has access to the private members of the class.

```
// declaration_matching2.cpp
class Account
{
public:
   Account()
   {
   }
   double Deposit( double dAmount, char *szPassword );

private:
      double Deposit( double dAmount )
      {
         return 0.0;
      }
      int Validate( char *szPassword )
      {
         return 0;
      }

   };

   int main()
   {
       // Allocate a new object of type Account.
       Account *pAcct = new Account;

       // Deposit $57.22. Error: calls a private function.
       // pAcct->Deposit( 57.22 );

       // Deposit $57.22 and supply a password. OK: calls a
       //  public function.
       pAcct->Deposit( 52.77, "pswd" );
   }

   double Account::Deposit( double dAmount, char *szPassword )
   {
      if ( Validate( szPassword ) )
         return Deposit( dAmount );
      else
         return 0.0;
}
```

SUMMARY

- Overloading refers to the ability to use a single identifier to define multiple methods of a class that differ in their input and output parameters. Overloaded methods are generally used when they conceptually execute the same task but with a slightly different set of parameters.
- Unlike Java, C# allows operators to be overloaded, in addition to methods, by defining static members using the operator keyword. This feature helps to extend and customize the semantics of operators relevant to user-defined types so that they can be used to manipulate object instances with operators.
- Overloading is the reuse of the same symbol or function name for two or more distinct operations or functions.
- FORTRAN also overloads generic functions. For example MAX stands for a family of maximum functions, with the compiler selecting the right one depending on context.
- Overloading: When a single Object has multiple behaviors. Then it is called as Overloading. Overloading is that in which a Single Object has a same name and Provides Many Functions.
- Constructor overloading is that in which a Constructor has a same name and has multiple Functions, then it is called as Constructor Overloading.
- Function overloading is a feature in C++ where two or more functions can have the same name but different parameters. Function overloading can be considered as an example of polymorphism feature in C++.
- In C++ programming, two functions can have same name if number and/or type of arguments passed are different.
- Overloaded operators are functions with special names: the keyword "operator" followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.
- Overloaded functions are selected for the best match of function declarations in the current scope to the arguments supplied in the function call.
- Overloaded functions differentiate between argument types that take different initializers. Therefore, an argument of a given type and a reference to that type are considered the same for the purposes of overloading. They are considered the same because they take the same initializers.
- Class member functions are treated differently, depending on whether they are declared as static. Because nonstatic functions have an implicit argument that supplies the pointer, nonstatic functions are considered to have one more argument than static functions; otherwise, they are declared identically.

- Any two function declarations of the same name in the same scope can refer to the same function, or to two discrete functions that are overloaded. If the argument lists of the declarations contain arguments of equivalent types (as described in the previous section), the function declarations refer to the same function. Otherwise, they refer to two different functions that are selected using overloading.

KNOWLEDGE CHECK

1. **Which is the correct statement anout operator overloading in C++?.**
   a. Only arithmetic operators can be overloaded
   b. Associativity and precedence of operators does not change
   c. Precedence of operators are changed after overlaoding
   d. Only non-arithmetic operators can be overloaded
2. **Which of the following operators cannot be overloaded?**
   a. .* (Pointer-to-member Operator)
   b. :: (Scope Resolution Operator)
   c. .* (Pointer-to-member Operator)
   d. All of the above
3. **While overloading binary operators using member function, it requires ___ argument?**
   a. 2
   b. 1
   c. 0
   d. 3
4. **Which of the following operators should be preferred to overload as a global function rather than a member method?**
   a. Postfix ++
   b. Comparison Operator
   c. Insertion Operator <<
   d. prefix ++
5. **Which of the following operator functions cannot be global, i.e., must be a member function.**
   a. new
   b. delete
   c. Converstion Operator
   d. All of the above
6. **Which of the following is correct option?**
   a. x = 5, y = 10
   b. x = 10, y = 5
   c. Compile Error
   d. x = 5, y = 5

7. **Which of the following is correct option?**
   a. x = 15, y = 3
   b. x = 3, y = 15
   c. Compile Error
   d. x = 15, y = 15
8. **Which of the following is correct option?**
   a. lets(int) called
   b. lets(lfc 2) called
   c. Compiler Error: Ambiguous call to lets()
   d. No error and No output
9. **Which of the following is the correct order involves in the process of operator overloading. i) Define the operator function to implement the required operations. ii) Create a class that defines the data type that is to be used in the overloading operation. iii) Declare the operator function op() in the public part of the class.**
   a. 1-i, 2-ii, 3-iii
   b. 1-ii, 2-iii, 3-i
   c. 1-ii, 2-i, 2-iii
   d. 1-iii, 2-ii, 3-i
10. **Which of the following is correct option?**
   a. Compiler Error
   b. 8 10
   c. 8 8
   d. 10 8

REVIEW QUESTIONS

1. What's the deal with operator overloading?
2. What are the benefits of operator overloading?
3. What are some examples of operator overloading?
4. Focus on function overloading and return type.
5. Differentiate between function overloading vs function overriding in c++.

Check Your Result

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

REFERENCES

1. Bracha, Gilad (3 September 2009). "Systemic Overload". Room 101.
2. Drayton, Peter; Albahari, Ben; Neward, Ted (2003). C# in a Nutshell. O'Reilly Media, Inc. ISBN 978-0-596-00526-9.
3. Fisher, Charles N. (2008). "Issues in Overloading" (PDF). University of Wisconsin–Madison.
4. Meyer, Bertrand (October 2001). "Overloading vs Object Technology" (pdf). Eiffel column. Journal of Object-Oriented Programming. 101 Communications LLC. 14 (4): 3–7. Retrieved 27 August 2020.
5. Smith, Chris (9 October 2012). Programming F# 3.0: A Comprehensive Guide for Writing Simple Code to Solve Complex Problems. O'Reilly Media, Inc. ISBN 978-1-4493-2604-3.
6. Stroustrup, Bjarne. "Operator Overloading". C++ FAQ. Archived from the original on 14 August 2011. Retrieved 27 August 2020.

CHAPTER 4

INHERITANCE

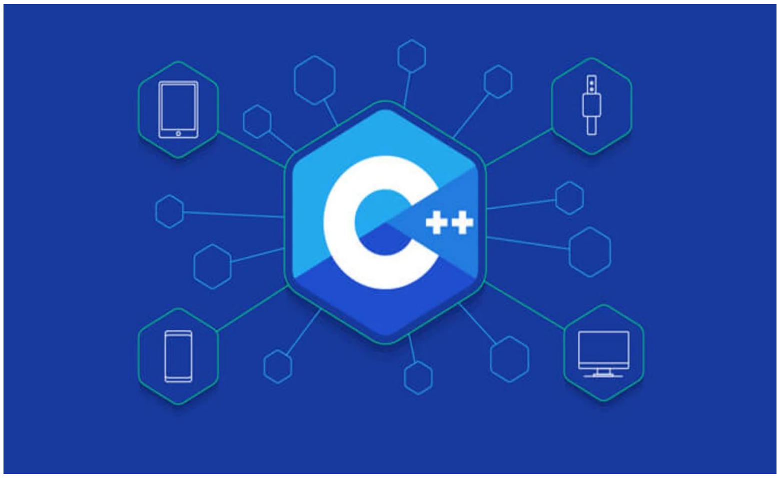
"Estate planning is an important and everlasting gift you can give your family. And setting up a smooth inheritance isn't as hard as you might think."

–Suze Orman

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Concept of inheritance
2. Multiple inheritance



INTRODUCTION

In C++, it is possible to inherit attributes and methods from one class to another. We group the "inheritance concept" into two categories:

derived class (child) - the class that inherits from another class

base class (parent) - the class being inherited from

To inherit from a class, use the : symbol.

In the example below, the Car class (child) inherits the attributes and methods from the Vehicle class (parent):

Example:

```
// Base class
class Vehicle {
  public:
    string brand = "Ford";
    void honk() {
      cout << "Tuut, tuut! \n" ;
    }
};

// Derived class
class Car: public Vehicle {
  public:
    string model = "Mustang";
};

int main() {
  Car myCar;
  myCar.honk();
  cout << myCar.brand + " " + myCar.model;
  return 0;
}
```

4.1 CONCEPT OF INHERITANCE

A class can inherit from zero or more base classes. A class with at least one base class is said to be a derived class. A derived class inherits all the data members and member functions of all of its base classes and all of their base classes, and so on. A class's immediate base classes are called direct base classes. Their base classes are indirect base classes. The complete set of direct and indirect base classes is sometimes .called the ancestor classes

A class can derive directly from any number of base classes. The base-class names follow a colon and are separated by commas. Each class name can be prefaced by an access specifier. The same class cannot be listed more than once as a direct base class, but it can appear more than once in the inheritance graph. For example, derived3 in the following code has base2 twice in its inheritance tree, once as a direct base class, and once as an indirect base class (through derived2):

```
class base1 { ... };
class derived1 : public base1 { ... };
class base2 { ... }
class derived2 : public derived1, public base2 { ... }
class derived3 : protected derived2, private base2 { ... }
```

A derived class can access the members that it inherits from an ancestor class, provided the members are not private. To look up a name in class scope, the compiler looks first in the class itself, then in direct base classes, then in their direct base classes, and so on.

To resolve overloaded functions, the compiler finds the first class with a matching name and then searches for **overloads** in that class. An object with a derived-class type can usually be converted to a base class, in which case the object is sliced. The members of the derived class are removed, and only the base class members remain:

```
struct file {

  std::string name;

};

struct directory : file {

  std::vector<file*> entries;

};
```

```
directory d;

file f;

f = d; // Only d.name is copied to f; entries are lost.
```

Slicing usually arises from a programming error. Instead, you should probably use a pointer or reference to cast from a derived class to a base class. In that case, the derived-class identity and members are preserved.

For example:

```
directory* dp = new directory;

file* fp;

fp = dp; // Keeps entries and identity as a directory object
```

Keyword

Overload is a six-degree-of-freedom shooter from the creators of Descent with intuitive controls, amazing lighting and graphics, and the best zero-G robot-blasting combat ever devised.

4.1.1 Inheritance in C++

When someone tells you, 'You've inherited your mom's looks!', it means that you got some of your features from her. In the programming world, the word inheritance basically means the same thing. There is a parent class or base class, which denotes a class from which a child class or a derived class inherits its features from. But why would we want to use inheritance in programming? Well, in the same way that a lot of DNA is shared between a mother and a child, a lot of code can be shared, or rather, reused.

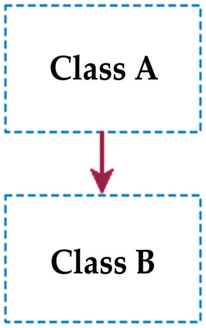
The general syntax of inheritance in C++ is as follows:

- class DerivedClass : accessSpecifier BaseClass

Types of inheritance

- Single Inheritance

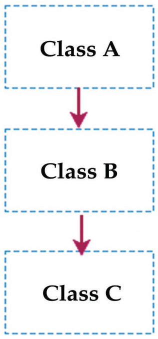
In this type of inheritance, we define one base class and one derived class.



Single level inheritance

```
//Base Class
class A
{
public void fooA()
{
//Inside base class
 }{
}
//Derived Class
class B : A
{
public void fooB()
 {
//Inside derived class
 }
}
```

Multilevel Inheritance



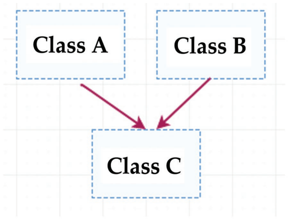
Multievel inheritance

In this type of inheritance, a derived class is itself derived from by another class.

```
//Base Class
class A {
public void fooA()
{
 //Inside base class
 }
//Derived Class
class B : A
{
public void fooB()
 {
//Inside derived class
 }
//Derived Class
class C : B
{
public void fooC()
 {
//Inside derived class
 }
}
```

Multiple Inheritance

In this type of inheritance, a derived class derives from two or more base classes.



Did You Know?

Inheritance was invented in 1969 for Simula.

Multiple inheritance

```
//Base Class
class A
{
public void fooA()
 {
 //Inside base class
 }
}//Base Class
class B
{
public void fooB()
{
  //Inside base class
  }
 }
//Derived Class
class C : A, B
{
public void fooC()
{
  //Inside derived class
  }
}
```

4.1.2 Virtual Functions

A nonstatic member function can be declared with the virtual function specifier, and is then known as a virtual function. A virtual function can be overridden in a derived class. To override a virtual function, declare it in a derived class with the same name and parameter types. The return type is usually the same but does not have to be identical. The virtual specifier is optional in the derived class but is recommended as a hint to the human reader. A constructor cannot be virtual, but a destructor can be. A virtual function cannot be static.

Keyword

Polymorphism is the ability of an object to take on many forms.

A class that has at least one virtual function is polymorphic. This form of polymorphism is more precisely known as type polymorphism. (C++ also supports parametric **polymorphism** with templates; Most programmers mean type polymorphism when they talk about object-oriented programming.

```
struct base {

  virtual void func( );

};

struct derived : base {

  virtual void func( ); // Overload

};

base* b = new derived;    // Static type of b is base*.

                          // Dynamic type is derived*.

b->func( );               // Calls dynamic::func( )
```

When any function is called, the compiler uses the static type to pick a function signature. If the function is virtual, the compiler generates a virtual function call. Then, at runtime, the object's dynamic type determines which function is actually callednamely, the function in the most-derived class that overrides the virtual function. This is known as a polymorphic function call.

4.1.3 Dispatching Virtual Functions

Virtual functions are most commonly implemented using virtual function tables, or vtables. Each class that declares at least one virtual function has a hidden data member (e.g., _ _vtbl). The _ _vtbl member points to an array of function pointers. Every derived class has a copy of the table. Every instance of a class shares a common table. Each entry in the table points to a function in a base class, or if the function is overridden, the entry points to the derived class function. Any new virtual functions that the derived class declares are added at the end of the table.

When an object is created, the compiler sets its _ _vtbl pointer to the vtable for its dynamic class. A call to a virtual function is compiled into an index into the table and into a call to the function at that index. Note that the dynamic_cast<> operator can use the same mechanism to identify the dynamic type of the object.

Multiple inheritance complicates matters slightly, yet the basic concept remains the same: indirection through a table of pointers.

Compilers do not have to use vtables, but they are used so widely, the term "vtable" has entered the common parlance of many C++ programmers.

An object's dynamic type can differ from its static type only if the object is accessed via a pointer or reference. Thus, to call a virtual function, you typically access the target object via a pointer (e.g., ptr->func( )). Inside a member function, if you call a virtual member function using its unqualified name, that is the same as calling the function via this->, so the function is called virtually. If a nonpointer, nonreference object calls a virtual function, the compiler knows that the static type and dynamic type always match, so it can save itself the lookup time and call the function in a nonvirtual manner.

Example shows a variety of virtual functions for implementing a simple calculator. A parser constructs a parse tree of expr nodes, in which each node can be a literal value or an operator. The operator nodes point to operand nodes, and an operand node, in turn, can be any kind of expr node. The virtual evaluate function evaluates the expression in the parse tree, returning a double result. Each kind of node knows how to evaluate itself. For example, a node can return a literal value or add the values that result from evaluating two operands.

Example. Declaring and using virtual functions

```
class expr {
```

```
public:
  virtual ~expr(  ) {}
  virtual double evaluate(  ) const = 0;
  std::string as_string(  ) const {
    std::ostringstream out;
    print(out);
    return out.str(  );
  }
  virtual void print(std::ostream& out) const {}
  virtual int precedence(  ) const = 0;
  template<typename charT, typename traits>
  static std::auto_ptr<expr> parse(
    std::basic_istream<charT,traits>& in);
};
// cout << *expr prints any kind of expression because expr->print(  ) is virtual.
template<typename charT, typename traits>
std::basic_ostream<charT,traits>&
```

```
  operator<<(std::basic_ostream<charT,traits>& out, const expr& e)

{
  e.print(out);

  return out;

}

class literal : public expr {

public:

  literal(double value) : value_(value) {}

  virtual double evaluate(  ) const { return value_; }

  virtual void print(std::ostream& out) const {

    out << value_;

  }

  virtual int precedence(  ) const { return 1; }

private:

  double value_;

};

// Abstract base class for all binary operators
```

```
class binop : public expr {

public:

  binop(std::auto_ptr<expr> left, std::auto_ptr<expr> right)

  : left_(left), right_(right) {}

  virtual double evaluate(  ) const {

    return eval(left_->evaluate(  ), right_->evaluate(  ));

  }

  virtual void print(std::ostream& out) const {

    if (left_->precedence(  ) > precedence(  ))

      out << '(' << *left_ << ')';

    else

      out << *left_;



    out << op(  );


    if (right_->precedence(  ) > precedence(  ))

      out << '(' << *right_ << ')';
    else
      out << *right_;
  }
  // Reminder that derived classes must override precedence
  virtual int precedence(  ) const = 0;
protected:
  virtual double eval(double left, double right) const = 0;
  virtual const char* op(  ) const = 0;
private:
  // No copying allowed (to avoid messing up auto_ptr<>s)
  binop(const binop&);
  void operator=(const binop&);
  std::auto_ptr<expr> left_;
  std::auto_ptr<expr> right_;
};

// Example binary operator.
```

```
class plus : public binop {

public:

  plus(std::auto_ptr<expr> left, std::auto_ptr<expr> right)

  : binop(left, right) {}

  virtual int precedence(  ) const { return 3; }

protected:

  virtual double eval(double left, double right) const {

    return left + right;

  }

  virtual const char* op(  ) const { return "+"; }

};



int main(  )

{

  while(std::cin) {

    std::auto_ptr<expr> e(expr::parse(std::cin));
```

```
        std::cout << *e << '\n';

        std::cout << e->evaluate(  ) << '\n';

      }

    }
```

Sometimes you do not want to take advantage of the virtualness of a function. Instead, you may want to call the function as it is defined in a specific base class. In such a case, qualify the function name with the base-class name, which tells the compiler to call that class's definition of the function:

```
literal* e(new literal(2.0));

e->print(std::cout);          // Calls literal::print

e->expr::print(std::cout);  // Calls expr::print
```

Keyword

A **base class** is the parent class of a derived class. Classes may be used to create other classes.

Covariant Return Types

The return type of an overriding virtual function must be the same as that of the base function, or it must be covariant. In a derived class, a covariant return type is a pointer or reference to a class type that derives from the return type used in the **base class**. Note that the return type classes do not necessarily have to match the classes that contain the functions, but they often do. The return type in the derived class can have additional const or volatile qualifiers that are not present in the base-class return type.

In a function call, the actual return value is implicitly cast to the static type used in the function call. Example shows one typical use of covariant types.

Example. Covariant return types

```
struct shape {

  virtual shape* clone(  ) = 0;
```

```
};
struct circle : shape {
  virtual circle* clone( ) {
    return new circle(*this);
  }
  double radius( ) const { return radius_; }
  void radius(double r) { radius_ = r; }
private:
  double radius_;
  point center_;
};
struct square : shape {
  virtual square* clone( ) {
    return new square(*this);
  }
private:
```

Remember

A class that has at least one virtual function should also have a virtual destructor. If a delete expression deletes a polymorphic pointer (for which the dynamic type does not match the static type), the static class must have a virtual destructor. Otherwise, the behavior is undefined.

```
  double size_;

  point corners_[4];

};



circle unit_circle;



circle* big_circle(double r)

{

  circle* result = unit_circle.clone( );

  result->radius(r);

  return result;

}



int main( )

{

  shape* s = big_circle(42.0);

  shape* t = s->clone( );
```

```
    delete t;

    delete s;

  }
```

4.1.4 Pure Virtual Functions

A virtual function can be declared with the pure specifier (=0) after the function header. Such a function is a pure virtual function (sometimes called an abstract function). The syntax for a pure specifier requires the symbols = 0. You cannot use an expression that evaluates to 0.

Even though a function is declared pure, you can still provide a function definition (but not in the class definition). A definition for a pure virtual function allows a derived class to call the inherited function without forcing the programmer to know which functions are pure.

Keyword

A **debugger** or debugging tool is a computer program that is used to test and debug other programs (the "target" program).

A derived class can override a pure virtual function and provide a body for it, override it and declare it pure again, or simply inherit the pure function.

Example shows some typical uses of pure virtual functions. A base class, shape, defines several pure virtual functions (clone, **debug**, draw, andnum_sides). The shape class has no behavior of its own, so its functions are pure virtual.

Example. Pure virtual functions

```
class shape {

public:

  virtual ~shape(  );

  virtual void draw(graphics* context)        = 0;

  virtual size_t num_sides(  )            const = 0;
  virtual shape* clone( )              const = 0;

  virtual void debug(ostream& out)        const = 0;

};



class circle : public shape {

public:

  circle(double r) : radius_(r) {}

  virtual void draw(graphics* context);

  virtual size_t num_sides( )     const { return 0; }

  virtual circle* clone( )      const { return new circle(radius(
)); }

  virtual void debug(ostream& out) const {

    shape::debug(out);

    out << "radius=" << radius_ << '\n';

  }

  double radius( ) const { return radius_; }

private:
```

Remember

A pure destructor must have a definition because a derived-class destructor always calls the base-class destructor.

```
  double radius_;

};



class filled_circle : public circle {

public:

  filled_circle(double r, ::color c) : circle(r), color_(c) {}

  virtual filled_circle* clone(  ) const {

    return new filled_circle (radius(  ), color(  ));

  }

  virtual void draw(graphics* context);

  virtual void debug(ostream& out) const {

    circle::debug(out);

    out << "color=" << color_ << '\n';

  }

  ::color color(  ) const { return color_;}

private:
  color color_;

};

void shape::debug(ostream& out)

const

{}
```

Even though shape::debug is pure, it has a function body. Derived classes must override shape::debug, but they can also call it, which permits uniform implementation of the various debug functions. In other words, every implementation of debug starts by calling the base class debug. Classes that inherit directly from shape do not need to implement debug differently from classes that inherit indirectly.

Abstract Classes

An abstract class declares at least one pure virtual function or inherits a pure virtual function without overriding it. A concrete class has no pure virtual functions (or all inherited pure functions are overridden). You cannot create an object whose type is an abstract class. Instead, you must create objects of concrete type. In other words, a concrete class that inherits from an abstract class must override every pure virtual function.

Abstract classes can be used to define a pure interface class, that is, a class with all pure virtual functions and no nonstatic data members. Java and Delphi programmers recognize this style of programming because it is the only way these languages support multiple inheritance. Example shows how interface classes might be used in C++.

Example. Using abstract classes as an interface specification

```
struct Runnable {

  virtual void run(  ) = 0;

};
```

```
struct Hashable {

  virtual size_t hash(  ) = 0;

};


class Thread : public Runnable, public Hashable {

public:

  Thread(  )                              { start_thread(*this); }

  Thread(const Runnable& thread)    { start_thread(thread); }

  virtual void run(  );

  virtual size_t hash(  ) const { return thread_id(  ); }

  size_t thread_id(  )     const;

  ...

private:

  static void start_thread(const Runnable&);

};

// Derived classes can override run to do something useful.

void Thread::run(  )
{}
```

4.2 MULTIPLE INHERITANCE

A class can derive from more than one base class. You cannot name the same class more than once as a direct base class, but a class can be used more than once as an indirect base class, or once as a direct base class and one or more times as an indirect base class. Some programmers speak of inheritance trees or hierarchies, but with multiple base classes, the organization of inheritance is a directed acyclic graph, not a tree. Thus, C++ programmers sometimes speak of inheritance graphs. If multiple base classes declare the same name, the derived class must qualify references to the name or else the **compiler** reports an ambiguity error:

Keyword

Compilers are a type of translator that support digital devices, primarily computers.

```
struct base1 { int n; };

struct base2 { int n; };

struct derived : base1, base2 {

  int get_n(  ) { return base1::n; } // Plain n is an error.

};
```

Objects of a derived-class type contain separate sub objects for each instance of every base class to store the base class's no static data members. To refer to a member of a particular sub object, qualify its name with the name of its base class. Static members, nested types, and enumerators are shared among all instances of a repeated base class, so they can be used without qualification (unless the derived class hides a name with its own declaration), as shown in Example.

Example. Multiple inheritance

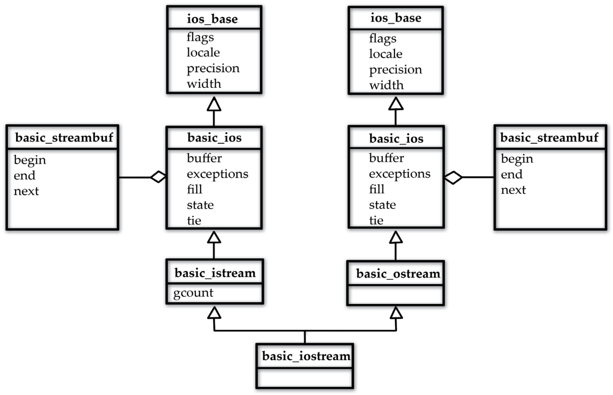
```
struct base1 {
  int n;
};
struct base2 {
  enum color { black, red };
  int n;
};
struct base3 : base2 {
  int n; // Hides base2::n
};
struct derived : base1, base2, base3 {
  color get_color( ); // OK: unambiguous use of base2::color
  int get_n( )  { return n; } // Error: ambiguous
  int get_n1( ) { return base2::n; } // Error: which base2?
  int get_n2( ) { return base3::n; } // OK
  int get_n3( ) {          // OK: another way to get to a specific member n
    base3& b3 = *this;
    base2& b2 = b3;
```

```
        return b2.n;

      }

    };
```

A well-designed inheritance graph avoids problems with ambiguities by ensuring that names are unique throughout the graph, and that a derived class inherits from each base class no more than once. Sometimes, however, a base class must be repeated in an inheritance graph. Figure illustrates the organization of multiple base classes, modeled after the standard I/O stream classes. Because basic_iostream derives from basic_istream and from **basic_ostream**, it inherits two sets of flags, two sets of buffers, and so on, even though it should have only one set of each.

Keyword

Basic_ ostream defines a number of member function signatures that assist in formatting and writing output to sequences controlled by a stream buffer.



Inheritance can be done in a number of ways. Till now, we have come across different types of inheritances in different examples.

The different types of inheritances which we have come across are:

Single Inheritance

single inheritance in C++

In single inheritance, a class inherits another class.

Multilevel Inheritance

multiple inheritance in C++

In this type of inheritance, one class inherits from another class. This base class inherits from some other class.

Hierarchical Inheritance

hierarchial in C++

In hierarchical inheritance, more than one class inherit from a base class.

In multiple inheritance, a class can inherit from more than one classes. In simple words, a class can have more than one parent classes. This type of inheritance is not present in Java.

Suppose we have to make two classes A and B as the parent classes of class C, then we have to define class C as follows.

```
class C: public A, public B
{
        // code
};
```

Let's see an example of multiple inheritance

```
#include <iostream>

using namespace std;

class Area
{
        public:
                int getArea(int l, int b)
                {
                        return l * b;
                }
};

class Perimeter
```

```
{
        public:
                int getPerimeter(int l, int b)
                {
                        return 2*(l + b);
                }
};

class Rectangle : public Area, public Perimeter
{
        int length;
        int breadth;
        public:
                Rectangle()
                {
                        length = 7;
                        breadth = 4;
                }
                int area()
                {
                        return Area::getArea(length, breadth);
                }
                int perimeter()
                {
                        return Perimeter::getPerimeter(length, breadth);
                }
};

int main()
{
        Rectangle rt;
        cout << "Area : " << rt.area() << endl;
        cout << "Perimeter : " << rt.perimeter() << endl;
```

```
    return 0;
}
```

Output

In this example, class Rectangle has two parent classes Area and Perimeter. Class 'Area' has a function getArea(int l, int b) which returns area. Class 'Perimeter' has a function getPerimeter(int l, int b) which returns the perimeter.

When we created the object 'rt' of class Rectangle, its constructor got called and assigned the values 7 and 4 to its data members length and breadth respectively. Then we called the function area() of the class Rectangle which returned getArea(length, breadth) of the class Area, thus calling the function getArea(int l, int b) and assigning the values 7 and 4 to l and b respectively. This function returned the area of the rectangle of length 7 and breadth 4.

Similarly, we returned the perimeter of the rectangle by the class Perimeter.

Let's see one more example.

```
#include <iostream>

using namespace std;

class P1
{
    public:
            P1()
            {
                    cout << "Constructor of P1" << endl;
            }
};

class P2
{
    public:
            P2()
            {
                    cout << "Constructor of P2" << endl;
```

```
                }
    };

    class A : public P2, public P1
    {
        public:
                A()
                {
                        cout << "Constructor of A" << endl;
                }
    };

    int main()
    {
        A a;
        return 0;
    }
```

Output

Here, when we created the object 'a' of class 'A', its constructor got called. As seen before, the compiler first calls the constructor of the parent class. Since class 'A' has two parent classes 'P1' and 'P2', so the constructors of both these classes will be called before executing the body of the constructor of 'A'. The order in which the **constructors** of the two parent classes are called depends on the following code.

```
class A : public P2, public P1
```

The order in which the constructors are called depends on the order in which their respective classes are inherited. Since we wrote 'public P2' before 'public P1', therefore the constructor of P2 will be called before that of P1.

Keyword

Overloaded **constructors** have the same name (name of the class) but different number of arguments.

4.2.1 Inheritance and Composition

The classes you have seen so far are complete classes: you can create an instance of the class on the free store or the stack. You can do this because the data members of the class have been defined and so it is possible to calculate how much memory is needed for the object, and you have provided the full functionality of the class. These are called concrete classes.

If you have a routine in a class that proves useful and you want to reuse in a new class, you have a few choices. The first is called composition. With composition you add an instance of your utility class as a data member of the classes that will use the routine. A simple example is the string class--this provides all the functionality that you want from a string. It will allocate memory according to how many characters have to be stored and deal locate the memory it uses when the string object is destroyed. Your class uses the functionality of a string, but it is not a string itself, hence it has the string as a data member.

What are the advantages of using inheritance in C++ Programming

The main advantages of inheritance are code reusability and readability. When child class inherits the properties and functionality of parent class, we need not to write the same code again in child class. This makes it easier to reuse the code, makes we write the less code and the code becomes much more readable.

Lets take a real life example to understand this: Let's assume that Human is a class that has properties such as height, weight, color etc. and functionality such as eating (), sleeping (), dreaming (), working () etc.

Now we want to create Male and Female class, these classes are different but since both Male and Female are humans they share some common properties and behaviors (functionality) so they can inherit those properties and functionality from Human class and rest can be written in their class separately. This approach makes us write less code as both the classes inherited several properties and functions from base class thus we didn't need to re-write them. Also, this makes it easier to read the code.

SUMMARY

- A class can inherit from zero or more base classes. A class with at least one base class is said to be a derived class.
- A class can derive directly from any number of base classes. The base-class names follow a colon and are separated by commas. Each class name can be prefaced by an access specifier.
- A derived class can access the members that it inherits from an ancestor class, provided the members are not private. To look up a name in class scope, the compiler looks first in the class itself, then in direct base classes, then in their direct base classes, and so on.
- Slicing usually arises from a programming error. Instead, you should probably use a pointer or reference to cast from a derived class to a base class. In that case, the derived-class identity and members are preserved.
- In the programming world, the word inheritance basically means the same thing. There is a parent class or base class, which denotes a class from which a child class or a derived class inherits its features from.
- A nonstatic member function can be declared with the virtual function specifier, and is then known as a virtual function. A virtual function can be overridden in a derived class.
- Virtual functions are most commonly implemented using virtual function tables, or vtables. Each class that declares at least one virtual function has a hidden data member (e.g., _ _vtbl).
- The return type of an overriding virtual function must be the same as that of the base function, or it must be covariant. In a derived class, a covariant return type is a pointer or reference to a class type that derives from the return type used in the base class.
- A virtual function can be declared with the pure specifier (=0) after the function header. Such a function is a pure virtual function (sometimes called an abstract function).
- An abstract class declares at least one pure virtual function or inherits a pure virtual function without overriding it.
- A class can derive from more than one base class. You cannot name the same class more than once as a direct base class, but a class can be used more than once as an indirect base class, or once as a direct base class and one or more times as an indirect base class.
- A well-designed inheritance graph avoids problems with ambiguities by ensuring that names are unique throughout the graph, and that a derived class inherits from each base class no more than once.

- The main advantages of inheritance are code reusability and readability. When child class inherits the properties and functionality of parent class, we need not to write the same code again in child class. This makes it easier to reuse the code, makes we write the less code and the code becomes much more readable.

KNOWLEDGE CHECK

1. **When the inheritance is private, the private methods in base class are __________ in the derived class (in C++).**
   a. Inaccessible
   b. Accessible
   c. Protected
   d. Public
2. **Which design patterns benefit from the multiple inheritances?**
   a. Adapter and observer pattern
   b. Code pattern
   c. Glue pattern
   d. None of the mentioned
3. **What is meant by multiple inheritance?**
   a. Deriving a base class from derived class
   b. Deriving a derived class from base class
   c. Deriving a derived class from more than one base class
   d. None of the mentioned
4. **What will be the order of execution of base class constructors in the following method of inheritance.class a: public b, public c {...};**
   a. b(); c(); a();
   b. c(); b(); a();
   c. a(); b(); c();
   d. b(); a(); c();
5. **Inheritance allow in C++ Program?**
   a. Class Re-usability
   b. Creating a hierarchy of classes
   c. Extendibility
   d. All of the above
6. **Can we pass parameters to base class constructor though derived class or derived class constructor?**
   a. Yes
   b. No
   c. May Be
   d. Can't Say

7. **What are the things are inherited from the base class?**
   a. Constructor and its destructor
   b. Operator=() members
   c. Friends
   d. All of the above
8. **Which of the following advantages we lose by using multiple inheritance?**
   a. Dynamic binding
   b. Polymorphism
   c. Both Dynamic binding & Polymorphism
   d. None of the mentioned
9. **What will be the output of the following program?**

```
Note:Includes all required header files
class find {
public:
   void print()  { cout <<" In find"; }
};

class course : public find {
public:
   void print() { cout <<" In course"; }
};

class tech: public course { };

int main(void)
{
  tech t;
  t.print();
  return 0;
}
```

   a. In find
   b. In course

c. Compiler Error: Ambiguous call to print()

d. None of the above

10. Which symbol is used to create multiple inheritance?

a. Dot

b. Comma

c. Dollar

d. None of the above

REVIEW QUESTIONS

1. Discuss about inheritance in C++.
2. What are the different types of inheritance?
3. Focus on virtual functions.
4. What is inheritance example?
5. Discuss about multiple inheritance.

Check Your Result

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

REFERENCES

1. Dr. K. R. Venugopal, Rajkumar Buyya (2013). Mastering C++. Tata McGrawhill Education Private Limited. p. 609. ISBN 9781259029943.
2. E Balagurusamy (2010). Object Orientedprogramming With C++. Tata McGrawhill Education Pvt. Ltd. p. 213. ISBN 978-0-07-066907-9.
3. Herbert Schildt (2003). The complete reference C++. Tata McGrawhill Education Private Limited. p. 417. ISBN 978-0-07-053246-5.
4. Holub, Allen (1 August 2003). "Why extends is evil". Retrieved 10 March 2015.
5. Mike Mintz, Robert Ekendahl (2006). Hardware Verification with C++: A Practitioner's Handbook. United States of America: Springer. p. 22. ISBN 978-0-387-25543-9.
6. Mitchell, John (2002). "10 "Concepts in object-oriented languages"". Concepts in programming language. Cambridge, UK: Cambridge University Press. p. 287. ISBN 978-0-521-78098-8.
7. Tempero, Ewan; Yang, Hong Yul; Noble, James (2013). What programmers do with inheritance in Java (PDF). ECOOP 2013–Object-Oriented Programming. pp. 577–601.

CHAPTER

5

POLYMORPHISM IN C++

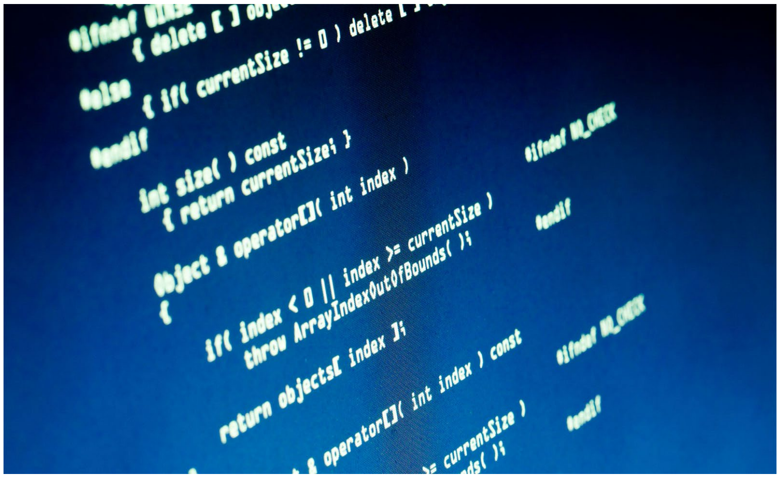
"I invented the term Object-Oriented, and I can tell you I did not have C++ in mind. "

–Alan Kay

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

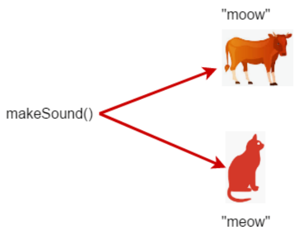
1. Describe the concept of polymorphism
2. Define the importance of polymorphism
3. Understand the implementing polymorphism c++
4. Explain the other applications of polymorphism
5. Focus on polymorphism explanation



INTRODUCTION

In C++, polymorphism causes a member function to behave differently based on the object that calls/invokes it. Polymorphism is a Greek word that means to have many forms. It occurs when you have a hierarchy of classes related through inheritance.

For example, suppose we have the function makeSound(). When a cat calls this function, it will produce the meow sound. When a cow invokes the same function, it will provide the moow sound.



Though we have one function, it behaves differently under different circumstances. The function has many forms; hence, we have achieved polymorphism.

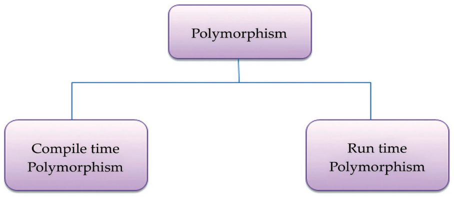
5.1 CONCEPT OF POLYMORPHISM

Simply speaking, polymorphism is the ability of something to be displayed in multiple forms. Let's take a real life scenario; a person at the same time can perform several duties as per demand, in the particular scenario. Such as, a man at a same time can serve as a father, as a husband, as a son, and as an employee. So, single person possess different behaviors in respective situations. This is the real life example of polymorphism. Polymorphism is one of the important features of OOP (Object Oriented Programming).

In C++ Polymorphism is mainly divided into two types

1) Compile Time Polymorphism and

2) Runtime Polymorphism



C++ Runtime Polymorphism Example

Let's see a simple example of runtime polymorphism in C++.

```
#include <iostream>
```

```
using namespace std;
class Animal {
    public:
void eat(){
cout<<"Eating...";
    }
};
class Dog: public Animal
{
 public:
 void eat()
    {
        cout<<"Eating bread...";
    }
};
int main(void) {
   Dog d = Dog();
   d.eat();
   return 0;
}
```

Output:

```
Eating bread...
```

C++ Runtime Polymorphism Example: By using two derived class. Let's see another example of runtime polymorphism in C++ where we are having two derived classes.

```
#include <iostream>
using namespace std;
class Shape {
    public:
virtual void draw(){
cout<<"drawing..."<<endl;
```

```
    }
};
class Rectangle: public Shape
{
 public:
 void draw()
    {
        cout<<"drawing rectangle..."<<endl;
    }
};
class Circle: public Shape
{
 public:
 void draw()
    {
        cout<<"drawing circle..."<<endl;
    }
};
int main(void) {
    Shape *s;
    Shape sh;
        Rectangle rec;
        Circle cir;
        s=&sh;
     s->draw();
        s=&rec;
     s->draw();
    s=○
     s->draw();
}
```

Output:

```
drawing...
```

```
drawing rectangle...
drawing circle...
```

Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let’s see an example where we are accessing the field by **reference variable** which refers to the instance of derived class.

Remember

A **reference variable** is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable.

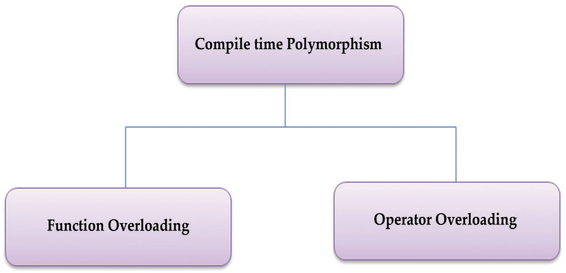
```
#include <iostream>
using namespace std;
class Animal {
    public:
    string color = “Black”;
};
class Dog: public Animal
{
 public:
    string color = “Grey”;
};
int main(void) {
     Animal d= Dog();
    cout<<d.color;
}
```

Output:

```
Black
```

5.1.1 Compile Time Polymorphism

This type of polymorphism is also known as static polymorphism and is achieved by overloading a function/ method or an operator.



Function Overloading in Compile Time

When multiple functions are used at different places with same name but different parameters then these functions are known as overloaded function. Functions can be overloaded in two ways:

By change in number of arguments

By change in type of arguments.

Let's consider an example.

Example of function overloading in C++C++

```
#include <iostream.h>
using namespace std;
class mycplus
{
    public:
      // function # 01: with 1 parameter of type int
    void function(int a)
    {
        cout << "value of a is " << a << endl;
    }
      // function # 02: having same name but a double parameter
    void function(double a)
    {
                    cout << "value of a is " << a << endl;
    }
        // function # 03: with same name but 2 int parameters
    void function(int a, int b)
    {
        cout << "value of a and b is " << a << ", " << b << endl;
    }
};

int main() {

    mycplus obj;
      // The call of function will depend on the type of parameters
     // passed
    //The first 'function with one int parameter' is called
    obj.function(3);
     // The second 'function with a double parameter' is called
    obj.function(6.456);
     // The third 'function with 2 int parameters' is called
    obj.function(8,71);
    return 0;
}
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
```

```
#include <iostream.h>
using namespace std;
class mycplus
{
    public:
      // function # 01: with 1 parameter of type int
    void function(int a)
    {
        cout << "value of a is " << a << endl;
    }
      // function # 02: having same name but a double parameter
    void function(double a)
    {
```

```
                    cout << "value of a is " << a << endl;
    }
        // function # 03: with same name but 2 int parameters
    void function(int a, int b)
    {
        cout << "value of a and b is " << a << ", " << b << endl;
    }
};

int main() {

    mycplus obj;
    // The call of function will depend on the type of parameters
     // passed
    //The first 'function with one int parameter' is called
    obj.function(3);
    // The second 'function with a double parameter' is called
    obj.function(6.456);
     // The third 'function with 2 int parameters' is called
    obj.function(8,71);
    return 0;
}
```

Output:

```
value of a is 3

value of a is 6.456

value of a and b is 8, 71
```

Remember

In computer programming, two notions of parameter are commonly used, and are referred to as parameters and arguments—or more formally as a formal parameter and an actual parameter.

The above example perfectly explains the concept of function overloading, a single function/method named function acts differently in 3 different situations which is the property of polymorphism.

Operator Overloading in Compile Time

The second method of compile time polymorphism is operator overloading. For example, we can make the operator ('+') for string class to concatenate two strings. The general concept regarding "+" operator is that it is an addition operator whose task is to sum .up two operands. But here in polymorphism, it will be used for a different purpose

So, a single operator '+' when placed between strings, concatenate them and when placed between integer operands, adds them. Let's take an example of illustrating Operator Overloading using C++.

C++ example to illustrate Operator OverloadingC++

```
#include<iostream>
using namespace std;
class Testop {
private:
    int real, imag;
public:
    Testop(int r = 0, int i =0): real(r), imag(i)  {}
      // The following function will automatically called when '+'
// is used with between two Testop objects
    Testop operator + (Testop const &obj) {
         Testop obj1;
         obj1.real = real + obj.real;
         obj1.imag = imag + obj.imag;
         return obj1;
    }
    void print()
{
cout << "The result of adding two complex numbers by operator overloading is
" << real << " + i" << imag << endl;
}
};

int main()
```

```
{
    Testop t1(12, 15), t2(1, 6);
    Testop t3 = t1 + t2; // call to "operator +"
    t3.print();
}
```

```
#include<iostream>
```

```
    using namespace std;
    class Testop {
    private:
        int real, imag;
    public:
        Testop(int r = 0, int i =0): real(r), imag(i)  {}
           // The following function will automatically called
when '+'
    // is used with between two Testop objects
        Testop operator + (Testop const &obj) {
             Testop obj1;
             obj1.real = real + obj.real;
             obj1.imag = imag + obj.imag;
             return obj1;
        }
        void print()
    {
    cout << "The result of adding two complex numbers by
operator overloading is  " << real << " + i" << imag << endl;
    }
    };

    int main()
    {
   Testop t1(12, 15), t2(1, 6);
   Testop t3 = t1 + t2; // call to "operator +"
   t3.print();
}
```

Output:

```
The result of adding two complex numbers by operator
overloading is 13 + i21
```

Keyword

Operator overloading (less commonly known as ad-hoc polymorphism) is a specific case of polymorphism (part of the OO nature of the language) in which some or all operators like +, = or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments.

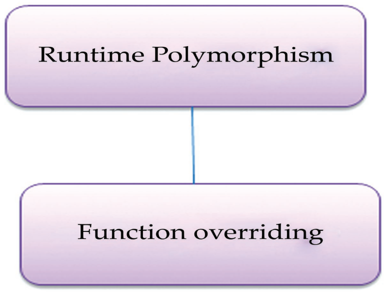
In above mentioned example the "+" operator is overloaded. Here the operator is made to perform addition of two complex numbers.

Testop operator + (Testop const &obj) is called automatically when the operator "+" is used in the main function.

Here it is important to mention that, operator overloading can be done on both unary as well as binary operators.

5.1.2 Runtime Polymorphism

This is the second type of polymorphism. It can be achieved by Function Overriding.



Function Overriding in Runtime

Function overriding is giving another definition to an existing method with same parameters or we can say that a method has same prototype in base as well as derived class.

For example: when an inherited class in c++ has a different definition for one of functions of the base class then here the function of base class is said to be overridden.

Example of function overriding in C++C++

```
 #include <bits/stdc++.h>
using namespace std;
 // Parent class
class Base
{
    public:
    void result()
    {
        cout << “Result method of Base class is called” << endl;
    }
};

// Following is the Derived class having similar result() method as of
//base class
class Derive : public Base
{
    public:

    // definition of a result method already exists in Base class
    void result()
    {
        cout << “The result method of derived class is called “ << endl;
    }

};

int main()
{
    //instantiating object of Base class
    Base obj;
         //object of child class
    Derive obj1 = Derive();

    // obj will call the result method in Base Class
    obj1.result();

    // obj1 will override the result method in Base
    // and call the result method in Derive class
    Obj1.result();
    return 0;
}
```

```
 #include <bits/stdc++.h>
using namespace std;
 // Parent class
class Base
{
    public:
    void result()
    {
        cout << "Result method of Base class is called" << endl;
    }
};

// Following is the Derived class having similar result() method as of
//base class
class Derive : public Base
{
    public:

    // definition of a result method already exists in Base class
    void result()
    {
        cout << "The result method of derived class is called " << endl;
    }

};
```

```
int main()
{
    //instantiating object of Base class
    Base obj;
        //object of child class
    Derive obj1 = Derive();

    // obj will call the result method in Base Class
    obj1.result();

    // obj1 will override the result method in Base
    // and call the result method in Derive class
    Obj1.result();
    return 0;
}
Output:

Result method of Base class is called

The result method of derived class is called
```

Keyword

A **derived class** is a class created or derived from another existing class. The existing class from which the derived class is created through the process of inheritance is known as a base class or superclass.

In the above-mentioned example, the result () method of base class is overridden and redefined in the base class.

Runtime Polymorphism using Virtual Functions

Runtime polymorphism can also be achieved by virtual functions. A virtual Function is the member of base class and is overrided in the derived class. The syntax of a virtual function is to precede its declaration with keyword "Virtual".

Here are some c programs to demonstrate how virtual pointers, virtual tables and virtual functions work in C++

Virtual function actually tells the compiler to perform Dynamic Binding (resolves function call at runtime) on it.

```
#include <iostream>

using namespace std;

class Polygon {

  protected:

    int w , h;

  public:
    void set_values (int x, int y)
      {

           w=x; h=y;

       }

    virtual int area ()
      {
         return 0;

       }
};

class Rect: public Polygon {

  public:

    int area ()
      {
        return w * h;
 }
};

class Tri: public Polygon {
  public:
    int area ()
      {

         return (w * h / 2);

        }
};

int main () {
  Rect rec;
  Tri trg;
  Polygon poly;
  Polygon * ppoly1 = &rec;
  Polygon * ppoly2 = &trg;
  Polygon * ppoly3 = &poly;
  ppoly1->set_values (11,25);
  ppoly2->set_values (11,25);
  ppoly3->set_values (11,25);
  cout << ppoly1->area() << '\n';
  cout << ppoly2->area() << '\n';
  cout << ppoly3->area() << '\n';

  return 0;
}
```

Output:

```
275
```

137

In the above-amentioned example, there are three classes in all (Polygon, Rectangle and Triangle) and all have the same functions set_values and area and same members: w, h.

Did You Know?

Row polymorphism is a similar, but distinct concept from subtyping. It deals with structural types. It allows the usage of all values whose types have certain properties, without losing the remaining type .information

Function main declares three pointers, ppoly1, ppoly2 and ppoly3 to Polygon. The first two pointers are assigned the addresses of rec and trg that are objects of Rect and Tri. This type of assignment is valid, as both classes Rect and Tri are derived from Polygon.

*ppoly1 and *ppoly2 is dereferencing of both pointers and enables us to access the members of their pointed objects. To understand the concept, consider the following example,

ppoly1->set_values (11,25); is equivalent to rec. set_values (11,25);

In base class, the member function "area" is declared as virtual and is later redefined in both of the derived classes Rect and Tri. So, it allows us to access the non-virtual members of derived classes via a reference of the base class. But if the keyword "virtual" is removed from the declaration of area in the base class, all three calls to area would return zero.

The virtual keyword allows the members of a derived class to be called appropriately who have the same name as one of the member of the base class. More accurately when the type of the pointer is a pointer to the parent class, which is pointing to an instance of the derived class, as you can see in the above example.

(Ad-hoc Polymorphism (Overloading

Ad-hoc polymorphism allows functions with the same name act differently for each type. For example, given two ints and the + operator, it adds them together. Given two std::strings it .concatenates them together. This is called overloading

Here is a concrete example that implements function add for ints and strings,

```
#include <iostream>
#include <string>

int add(int a, int b) {
 return a + b;
}

std::string add(const char *a, const char *b) {
 std::string result(a);
 result += b;
 return result;
}

int main() {
 std::cout << add(5, 9) << std::endl;
 std::cout << add("hello ", "world") << std::endl;
}
```

Ad-hoc polymorphism also appears in C++ if you specialize templates.

```
template <>
const char *max(const char *a, const char *b) {
 return strcmp(a, b) > 0 ? a : b;
}
```

Now you can call ::max("foo", "bar") to find maximum of strings "foo" and "bar".

Coercion Polymorphism (Casting)

Coercion happens when an object or a primitive is cast into another object type or primitive type. For example,

```
float b = 6; // int gets promoted (cast) to float implicitly
int a = 9.99 // float gets demoted to int implicitly
```

Explicit casting happens when you use C's type-casting expressions, such as (unsigned int *) or (int) or C++'s static_cast, const_cast, reinterpret_cast, or dynamic_cast.

Coercion also happens if the constructor of a class isn't explicit, for example,

```
#include <iostream>

class A {
 int foo;
public:
 A(int ffoo) : foo(ffoo) {}
 void giggidy() { std::cout << foo << std::endl; }
};

void moo(A a) {
 a.giggidy();
}

int main() {
 moo(55);      // prints 55
}
```

If you made the constructor of A explicit, that would no longer be possible. It's always a good idea to make your constructors explicit to avoid accidental conversions.

Also if a class defines conversion operator for type T, then it can be used anywhere where type T is expected.

For example,

```
class CrazyInt {
 int v;
public:
 CrazyInt(int i) : v(i) {}
```

Keyword

Conversion functions (C++ only) You can define a member function of a class, called a conversion function, that converts from the type of its class to another specified type. ... Classes, enumerations, typedef names, function types, or array types cannot be declared or defined in the conversion_type.

```
 operator int() const { return v; } // conversion from CrazyInt
to int
};
```

The CrazyInt defines a **conversion operator** to type int. Now if we had a function, let's say, print_int that took int as an argument, we could also pass it an object of type CrazyInt,

```
#include <iostream>

void print_int(int a) {
 std::cout << a << std::endl;
}

int main() {
 CrazyInt b = 55;
 print_int(999);    // prints 999
 print_int(b);      // prints 55
{
```

5.2 IMPORTANCE OF POLYMORPHISM

Polymorphism saves the programmer a lot of time in re-creating code. You don't want to have to write completely different .modules for every possible permutation

If you had methods for tree growth, it would be hard to have to write a specific growth method for maple, spruce, pine, etc. Instead, you can have a growth function that spans across all three types, and tweak it as you see fit for each possible tree. We can all understand the concept of growth, and understand there are variation in growth, depending on the .object (tree) we are working with

5.3 IMPLEMENTING POLYMORPHISM C++

Polymorphism allows you to create a pointer to a derived class which is also compatible with the base class. Still confused? Let's look at an example of how this would look.

Below is code for a Container base (parent) class. The derived (child) classes are Cylinder and Sphere. Create the following in your compiler:

```
#include <iostream>
#include <cmath>
using namespace std;
const float PI = 3.1415927;
class Container {
 protected:
  float height;
  float width;
  float radius;
 public:
  void set_volume (float h, float w, float r) {
   height = h;
   width = w;
   radius = r;
  }
};
class Sphere: public Container {
 public:
  float volume() {
   float v = ((4/3) * PI * pow(radius, 3));
   return v;
  }

};
class Cylinder: public Container {
 public:
  float volume() {
   float v = PI * pow(radius, 2) * height;
   return v;
  }
};
int main() {
}
```

Next, in the main function we will create an instance of a Sphere (sphere) and an instance of a Cylinder (cylinder). We will also create pointers from the parent/base class of Container, that point to each of these instances.

```
Sphere sphere;
Cylinder cylinder;
Container *ptrContainer1 = &sphere;
Container *ptrContainer2 = &cylinder;
ptrContainer1 -> set_volume(33.53, 25.11, 0);
ptrContainer2 -> set_volume(13, 15, 0);
cout << sphere.volume() << endl;
cout << cylinder.volume() << endl;
```

Notice how each pointer calls the set_volume function. The key here is that, because of polymorphism, we can have any number of set_volume functions, but used a little differently. Recall that polymorphism really means multiple forms.

Remember

Before digging into polymorphism in C++, you should have a good sense of pointers and how inheritance works in C++. Additionally, in other lesson we learned about derived and base classes. If you are still unclear, please review lessons which speaks on .that subject

5.4 OTHER APPLICATIONS OF POLYMORPHISM

Let's take a look at some more examples of polymorphism in C++

Overloading

This is a prime example of polymorphism.

In our volume example, we could add a second function for setting volume with only two parameters, and one with three. When you call the function, C++ is smart enough to know which one to use, based on the number of parameters that you send.

```
void set_volume (float h, float r, float w) {
 height = h;
 width = w;
 radius = r;
}
```

```
void set_volume (float h, float w) {
 height = h;
 width = w;
}
```

In the main function, you would call the function differently, depending on the need:

```
Square square;
square.set_volume (22.5, 17, 38.5);
Tester test;
test.set_volume(33,8.35);
```

5.5 POLYMORPHISM EXPLANATION

A polymorphic function or operator has many forms. For example, in c++ the division operator is polymorphic. If the arguments to the division operator are integral, then integer division is used. However, if one or both arguments are floating –point then floating –point division is used.object oriented programming language support polymorphism, which is characterized by the phrases, "ONE INTERFACE MULTIPLE METHODS".A real world example of polymorphism is a thermostat. no matter what type of furnace your house has (gas, oil, electric etc.)

In your program, you will create three specific version of these function one for each type of stack, but names of the function will be the same.

In C++, a function name or operator can be overloaded. A function is called based on its signature which is the list of argument types in its parameter list.

Polymorphism Explanation with Example

For example, in the division expression

a/b the result depends on the arguments being automatically coerced to the widest type. so if both arguments are integer, the result is an integer division. But if one or both arguments are floating-point, the result is floating-point.

Another example is the output statement

```
cout<<a;
```

Where the shift operator << is invoking a function that is able to output an object of the type. A technique for implementing a package of routines to provide a shape type could rely on a comprehensive structural description of any shape for instance,

```
Struct shape
{
    enum {
        circle, rectangle, ?.
    } e - val;
    double center, radius;

    // other code
};
```

would have all the members necessary for any shape currently drawable in our system.. It would also have an enumerator value so that it could be identified. The area routine would then be written as:

```
double area(shape *s) {
    switch (s->e - val) {
        case circle:return (PI * s > radius * s > radius);
        case rectangle:return (s > height * s > width);
     // other code
    }
};
```

An additional case in the code body and additional members in the structure are needed. Unfortunately, these would have ripple effects throughout our entire code body.

C++ code following this design uses shape as an abstract class containing one or more pure virtual functions, as shown in the following code.

```
//shape is an abstract base class

class shape {
public:
    virtual double area() = 0;
};
```

```
class rectangle : public shape
{
public:
    rectangle double area()
    {
        return (height * width)
    }
private:
    double height, width;
};

class circle : public shape
{
public:
    double area()
    {
        return (3.14159 * radius * radius);
    }
private:
    double radius;
};
```

Remember

Hierarchical design should minimize interface parameter passing. Each layer tends to absorb, within its implementation, state detail that is affected by function invocation.

the client code for computing an arbitrary area is polymorphic the appropriate area() function is selected at run time.

```
shape * ptr _shape;
cout << "area =" << ptr_shape->area();
```

Now imagine is improving our hierarchy of types by

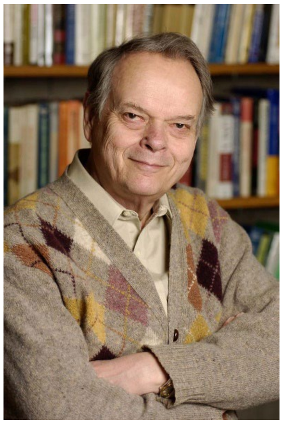
developing a square class:

```
Class square : public rectangle
{
public:
    square(double h) : rectangle(h, h)
    {


    }
    double area()
    {
        return (rectangle::area());
    }
};
```

The client code remains unchanged. This was not the case with the non-oops code.

This practice is almost universally condemned because it leads to opaque side-effect-style coding that is difficult to debug, revise and maintain. It is the compiler's job to select the specific action as it applies to each situation.

The first object -oriented programming language were interpreters, polymorphism was, of course, supported at run-time. However,c++ is a compiled language. Therefore, in c++, both run-time and compile-time polymorphism are supported.



ROLE MODEL

JOHN C. REYNOLDS: DESIGNER OF PROGRAMMINGLANGUAGESANDLANGUAGES FOR SPECIFYING PROGRAM BEHAVIOR

Despite his responsibilities as a prominent faculty member at Carnegie Mellon University, John C. Reynolds managed to find time outside the classroom to nurture the ideas of students, junior faculty and other developing academics.

A research colleague said he discovered that when he was invited as a visiting graduate student nearly a quarter-century ago to stay in Mr. Reynolds' home in Shadyside and found the man willing to spend hours engaging him in his ideas. He also found it later in life as the two worked together on cutting-edge research.

"For me, this was an amazing experience, talking to one of the greatest minds computer science has known," Peter O'Hearn, who now works at University College London, recalled Thursday in an email. "He was sharp but forgiving. He caught my mistakes but always met them with a more positive suggestion of new possibilities."

Mr. Reynolds, a computer science professor at Carnegie Mellon since 1986, died Sunday at Forbes Hospice of cancer and congestive heart disease. He was 77 and continued to teach until retiring in January of this year.

He and Mr. O'Hearn went on to collaborate a decade ago on what Carnegie Mellon said was some of the most influential published works done by Mr. Reynolds, whose interests centered around designing programming languages and developing tools and methods for uses, including verification of program correctness.

According to Carnegie Mellon, the pair created "a framework for reasoning about programs called separation logic." The university said their work has blossomed into a sizable area of research.

There were others at Carnegie Mellon who recalled Mr. Reynolds' nurturing side, among them Frank Pfenning, head of the Computer Science Department, who was a postdoctoral student there when he met Mr. Reynolds in 1986.

He recalled how Mr. Reynolds counseled against chasing after trends and urged researchers to trust their instincts. Mr. Pfenning said that when Mr. Reynolds would be asked about whether to publish a paper on a particular topic, "He would say, 'Well, do you think it's worth publishing?' He would put things back to you. It made you see things in a different light."

Mr. Reynolds was raised in Glen Ellyn, Ill. He completed his undergraduate studies at Purdue University. He received a doctoral degree in theoretical physics from Harvard University in 1961.

He and his future wife, Mary Allen, met while he attended Harvard. She said he liked to discuss ideas with colleagues. "They would have some heated discussions, but what came out of it became very important and influential work."

He was employed by Argonne National Laboratory for nine years starting in 1961. He also was a visiting junior faculty member at Stanford University. He also worked part time at the University of Chicago.

He was a professor of computer and information science at Syracuse University for 16 years starting in 1970 and then moved to Carnegie Mellon.

His professional titles included editor of the Communications of the Association for Computing Machinery. Among the awards he received was Lovelace Medal of the British Computer Society in 2010 and Carnegie Mellon's Dana Scott Distinguished Research Career Award in 2006.

In addition to his wife of 52 years, Mr. Reynolds is survived by two sons, Edward of Syracuse, N.Y., and Matthew of Seattle.

Arrangements are being handled by John A. Freyvogel Sons, Shadyside. Carnegie Mellon said a memorial service will take place May 11 at 11 a.m. at Allegheny Cemetery Mausoleum. In lieu of flowers, donations can be directed to East End Cooperative Ministry, 250 N. Highland Ave., Pittsburgh 15206.

SUMMARY

- The word polymorphism means having many forms. In simple words, Simply speaking, polymorphism is the ability of something to be displayed in multiple forms.
- Runtime Polymorphism can be achieved by data members in C++.
- When multiple functions are used at different places with same name but different parameters then these functions are known as overloaded function.
- The second method of compile time polymorphism is operator overloading.
- Function overriding is giving another definition to an existing method with same parameters or we can say that a method has same prototype in base as well as derived class.
- Runtime polymorphism can also be achieved by virtual functions. A virtual Function is the member of base class and is overrided in the derived class. The syntax of a virtual function is to precede its declaration with keyword "Virtual".
- Function main declares three pointers, ppoly1, ppoly2 and ppoly3 to Polygon. The first two pointers are assigned the addresses of rec and trg that are objects of Rect and Tri. This type of assignment is valid, as both classes Rect and Tri are derived from Polygon.
- The virtual keyword allows the members of a derived class to be called appropriately who have the same name as one of the member of the base class.
- Ad-hoc polymorphism allows functions with the same name act differently for each type. For example, given two ints and the + operator, it adds them together. Given two std::strings it concatenates them together. This is called overloading.
- Coercion happens when an object or a primitive is cast into another object type or primitive type.
- Polymorphism saves the programmer a lot of time in re-creating code. You don't want to have to write completely different modules for every possible permutation.
- Polymorphism allows you to create a pointer to a derived class which is also compatible with the base class.
- A polymorphic function or operator has many forms. For example, in c++ the division operator is polymorphic. If the arguments to the division operator are integral, then integer division is used.

KNOWLEDGE CHECK

1. **Polymorphism is achieved through ___**
    a. Heritance
    b. Poly programming
    c. Encapsulation
    d. Overloading
2. **The word polymorphism means ____**
    a. Many programs
    b. Two forms
    c. Single form
    d. Many shapes
3. **The mechanism of giving special meaning to an operator is called ____**
    a. Object
    b. Inheritance
    c. Function overloading
    d. Operator Overloading
4. **In function overloading do not use the ___ function name for two unrelated functions.**
    a. Same
    b. Different
    c. Similar
    d. Complement
5. **Strcat() function is used for ____**
    a. Substring
    b. String calculation
    c. String comparison
    d. String concatenation
6. **While invoking functions, if the C++ compiler does not find the exact match of the function call statement then ___**
    a. Looks for the next nearest match
    b. Deletes the function
    c. Generates an error
    d. It will ignore the function call

7. **The functionality of operator like '+' can be extended using ___**
   a. Operator precedence
   b. Operator overloading
   c. Operator definition
   d. None of the given
8. **Binary operators overloaded through a member function take one ____ argument.**
   a. Default
   b. Complete
   c. Implicit
   d. Explicit
9. **The operator function must be ____**
   a. A member function
   b. A friend function
   c. Either member or friend function
   d. None of the given
10. **____ Promotions are purely compiler oriented.**
   a. Constant
   b. Integral
   c. Floating point
   d. Character

REVIEW QUESTIONS

1. Discuss about polymorphism.
2. What is the meaning of polymorphism in OOP?
3. What is polymorphism and its types in C++?
4. When you should use virtual inheritance?
5. What is the output of the following code:

```
<include <iostream#
} ([]int main(int argc, const char * argv
;{int a[] = {1, 2, 3, 4, 5, 6
;[std::cout << (1 + 3)[a] - a[0] + (a + 1)[2
{
```

Check Your Result

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

REFERENCES

1. Booch, et al 2007 Object-Oriented Analysis and Design with Applications. Addison-Wesley.
2. Christopher Strachey. Fundamental Concepts in Programming Languages (PDF). www.itu.dk. Kluwer Academic Publishers. Archived from the original (PDF) on 2017-08-12. Retrieved 2012-10-13.
3. Pierce, B. C. 2002 Types and Programming Languages. MIT Press.
4. Ralf Lammel and Joost Visser, “Typed Combinators for Generic Traversal”, in Practical Aspects of Declarative Languages: 4th International Symposium (2002), p. 153.
5. Allen B. Tucker (28 June 2004). Computer Science Handbook, Second Edition. Taylor & Francis. pp. 91–. ISBN 978-1-58488-360-9.
6. Bjarne Stroustrup (February 19, 2007). “Bjarne Stroustrup’s C++ Glossary”. polymorphism – providing a single interface to entities of different types.

CHAPTER 6

C++ EXCEPTION HANDLING

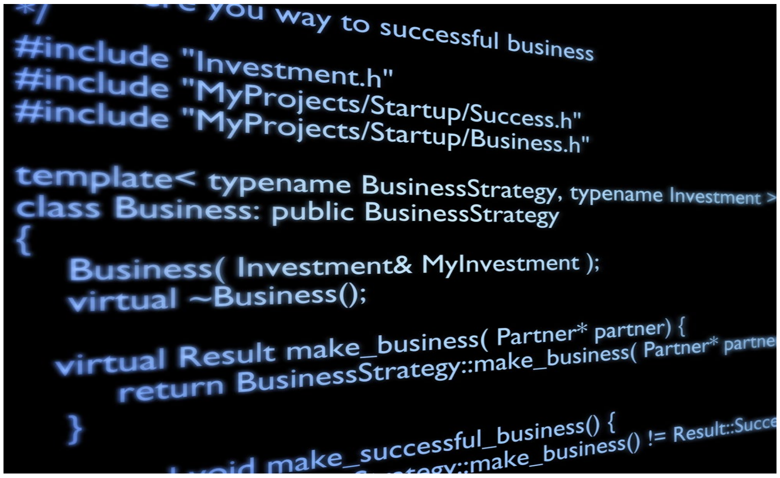
"Programming is like sex: It may give some concrete results, but that is not why we do it."

– Bjarne Stroustrup

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Exception handling in C++ programming
2. Exception handling over traditional error handling



```
*/   you way to successful business
#include "Investment.h"
#include "MyProjects/Startup/Success.h"
#include "MyProjects/Startup/Business.h"

template< typename BusinessStrategy, typename Investment >
class Business: public BusinessStrategy
{
    Business( Investment& MyInvestment );
    virtual ~Business();

    virtual Result make_business( Partner* partner) {
        return BusinessStrategy::make_business( Partner* partne
    }

    void make_successful_business() {
        tegy::make_business() != Result::Succe
```

INTRODUCTION

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

throw − A program throws an exception when a problem shows up. This is done using a throw keyword.

catch − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

try − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows −

```
try {
   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
} catch( ExceptionName eN ) {
   // catch block
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

6.1 CONCEPT OF EXCEPTION HANDLING IN C++ PROGRAMMING

Exceptions are runtime anomalies that a program encounters during execution. It is a situation where a program has an unusual condition and the section of code containing it can't handle the problem. Exception includes condition such as division by zero, accessing an array outside its bound, running out of memory, etc.

In order to handle these exceptions, exception handling mechanism is used which identifies and deal with such condition.

Exception handling mechanism consists of following parts:

- Find the problem (Hit the exception)
- Inform about its occurrence (Throw the exception)
- Receive error information (Catch the exception)
- Take proper action (Handle the exception)

C++ consists of 3 keywords for handling the exception. They are

- **try:** Try block consists of the code that may generate exception. Exception are thrown from inside the try block.
- **throw:** Throw keyword is used to throw an exception encountered inside try block. After the exception is thrown, the control is transferred to catch block.
- **catch:** Catch block catches the exception thrown by throw statement from try block. Then, exception are handled inside catch block.

Syntax

```
try
{
    statements;
    ... ... ...
    throw exception;
}

catch (type argument)
{
    statements;
    ... ... ...
}
```

6.1.1 Multiple Catch Exception

Multiple catch exception statements are used when a user wants to handle different exceptions differently. For this, a user must include catch statements with different declaration.

Syntax

```
try
{
    body of try block
}

catch (type1 argument1)
{
    statements;
    ... ... ...
}

catch (type2 argument2)
{
    statements;
    ... ... ...
}
... ... ...
... ... ...
catch (typeN argumentN)
{
    statements;
    ... ... ...
}
```

Keyword

Each **catch block** is an exception handler that handles the type of exception indicated by its argument.

6.1.2 Catch all Exceptions

Sometimes, it may not be possible to design a separate **catch block** for each kind of exception. In such cases, we can use a single catch statement that catches all kinds of exceptions.

Syntax

```
catch (...)
{
    statements;
    ... ... ...
}
```

Note: A better way is to use catch (...) as a default statement along with other catch statement so that it can catch all those exception which are not handled by other catch statements.

Example of exception handling

C++ program to divide two numbers using try catch block.

```
#include <iostream>
#include <conio.h>
using namespace std;

int main()
{
    int a,b;
    cout << "Enter 2 numbers: ";
    cin >> a >> b;
    try
    {
        if (b != 0)
        {
            float div = (float)a/b;
            if (div < 0)
                throw 'e';
            cout << "a/b = " << div;
        }
        else
            throw b;

    }
    catch (int e)
    {
        cout << "Exception: Division by zero";
    }
    catch (char st)
    {
        cout << "Exception: Division is less than 1";
    }
    catch(...)
    {
        cout << "Exception: Unknown";
    }
    getch();
    return 0;
}
```

This program demonstrate how exception are handled in C++. This program performs division operation. Two numbers are entered by user for division operation. If the dividend is zero, then division by zero will cause exception which is thrown into catch block. If the answer is less than 0, then exception "Division is less than 1" is thrown. All other exceptions are handled by the last catch block throwing "Unknown" exception.

Output

```
Enter 2 numbers: 8 5
a/b = 1.6

Enter 2 numbers: 9 0
Exception: Division by zero

Enter 2 numbers: -1 10
Exception: Division is less than 1
```

6.1.3 Some Useful Facts to Know Before Using C++ Exceptions

Exceptions provide many benefits over error codes for error handling. Some of these benefits are:

Exceptions cannot be silently ignored whereas checking the error code of a method can be ignored by the method caller.

Exceptions propagate automatically over method boundaries, but error codes do not.

Exception handling removes error handling and recovery from the main line of control flow that makes code more readable and maintainable.

Exceptions are the best way to report errors from **constructors** and operators.

Despite these benefits, most people still do not prefer to use exceptions due to its overhead. Depending on the implementation, this overhead comes in two forms: time overhead (increased execution time) and space overhead (increased executable size and memory consumption). From these two, most are concerned about time overhead. However, in a good C++ exception implementation, unless an exception is thrown, no run-time overhead is incurred. The real issue with C++ exceptions is not in their execution performance, but how to use them correctly. Following are some useful facts to know in order to use C++ exceptions correctly.

Keyword

Constructors in C++ Constructor has same name as the class itself. Constructors don't have return type. A constructor is automatically called when an object is created.

1. Exceptions Should Not Be Thrown From Destructors

Consider the following code:

```
try
{
  MyClass1 Obj1;
  Obj1.DosomeWork();
  ...
}
catch(std::exception & ex)
{
   //do error handling
}
```

If an exception is thrown from the MyClass1::DosomeWork() method, before execution moves out from the try block, the destructor of *obj1* needs to be called as *obj1* is a properly constructed object. What if an exception is also thrown from the destructor of MyClass1? This exception occurred while another exception was active. If an exception is thrown while another exception is active, the C++ behavior is to call the terminate() method, which halts the execution of the application. Therefore, to avoid two exceptions being active at the same time, destructors must not throw exceptions.

2. Objects Thrown as Exceptions Are Always Copied

Remember

If we use such objects as exception objects we should make sure that our exception classes have proper copy constructors.

When an exception is thrown, a copy of the exception object always needs to be created as the original object goes out of the scope during the stack unwinding process. Therefore, the exception object's copy constructor will always be called. C++ provides a copy constructor by default if we don't provide one. But there may be cases where the default copy constructor doesn't work; especially when the class members are pointers.

The most important thing about copying objects in C++ is that the copy constructor is always called based on the static type, not the dynamic type. Consider the following code [1].

```
class Widget { ... };
class SpecialWidget: public Widget { ... };
void passAndThrowWidget()
{
    SpecialWidget localSpecialWidget;
    ...
    Widget& rw = localSpecialWidget;     // rw refers to a
SpecialWidget
    throw rw;          // this throws an exception of type
Widget!
}
```

In the above code, the throw statement throws an object of type Widget, rw's static type is Widget. This might not be the behavior that we want to execute.

3. Exceptions Should Be Caught by Reference

There are three ways to catch exceptions in the catch clause:

- 1. Catch exception by value.
- 2. Catch exception as a pointer.
- 3. Catch exception by reference.

Catching exceptions by value is costly and it suffers from the slicing problem. It is costly because it needs to create two exception objects every time. When an exception is thrown, a copy of the exception object needs to be created no matter how we catch it, because when the stack unwinds the original object would go out the scope. If the exception was caught by value, another copy is made to pass it to the catch clause. Therefore, if the exception was caught by value, two exception objects are created, making the exception throwing process slower.

The slicing problem comes into effect when the catch clause is declared to catch a super class type and a derived class object is thrown as an exception. In that case, the catch clause only receives a copy of the super class object, which lacks the attributes of the original exception object. Therefore, catching exceptions by value must be avoided.

If exceptions were caught as pointers, the code would be as follows;

```
void doSomething()
{
  try
  {
     someFunction();              // might throw an exception*
  }
  catch (exception* ex)
  {                // catches the exception*;
    ...            // no object is copied
  }
}
```

In order to catch the exception as a pointer, the pointer to the exception object should be thrown and the throwing party must ensure that the exception object will be kept alive even after the stack unwinding, as a result of the throw. Even though a copy of the exception object will be created, it is a copy of the pointer in this case. Therefore, other measures should be taken to keep the exception object alive after the throw. This can be achieved by passing the pointer to a global or a static object, or creating the exception object in the heap.

Nevertheless, the user who catches the exception has no idea about how the exception object was created, and thus

Remember

Catching exceptions by reference doesn't have any of these issues that are observable in 'catch as a pointer' or 'catch by value.' The user doesn't need to worry about the deletion of the exception object. No additional exception object will be copied, as a reference to the original exception object is passed. Pass by reference doesn't have the slicing problem. Therefore, the exception should be caught by reference for proper and efficient operation.

they are uncertain about whether to delete the receiving pointer at the catch clause or not. Therefore, catching the exception as a pointer is sub-optimal.

4. Prevent Resource Leaks in Case of Exceptions

Consider the following code:

```
void SomeFunction()
{
    SimpleObject* pObj = new SimpleObject();
    pObj->DoSomeWork();//could throw exceptions
    delete pObj;
}
```

In this method new SimpleObject is created, then some work has been done through SimpleObject::DoSomeWork() method and finally the object is destroyed. But what if Object::DoSomeWork() threw an exception? In that case we don't get a chance to delete the pObj. This introduces a memory leak. This is a simple example to illustrate that exceptions could lead to resource leaks and of course this can be eliminated by putting a simple try catch block. But in real life this scenario could happen from points of code where we can't figure out the leak at first glance. One remedy for this type of cases is to use auto pointers from standard library (std::auto_ptr) [1].

5. When throwing an exception, do not leave the object in an inconsistent state

Consider the following example code [4]:

```
template <class T>
class Stack
{
   unsigned nelems;
   int top;
   T* v;
 public:
   void push(T);
   T pop();
   Stack();
   ~Stack();
};
template <class T>
```

```
void Stack<T>::push(T element)
{
  top++;
  if( top == nelems-1 )
  {
    T* new_buffer = new (nothrow) T[nelems+=10];
    if( new_buffer == 0 )
      throw "out of memory";
    for(int i = 0; i < top; i++)
      new_buffer[i] = v[i];
    delete [ ] v;
    v = new_buffer;
  }
  v[top] = element;
}
```

If the exception "out of memory" was thrown, the Stack::push () method leaves the Stack object in an inconsistent state, because the top of the stack has been incremented without pushing any element. Of course, this code can be modified so that it won't happen. However, special care needs to be taken when throwing an exception so that the object which throws the exception will be in a valid state even after the exception. In the following naïve implementation of ThreadSafeQueue::Pushback() method, if an exception was thrown from the DoPushBack() method, _mutex will be kept locked, leaving the ThreadSafeQueue object in an inconsistent state. To overcome this situation, lock_guards can be used, as auto pointers are used to prevent memory leaks. It should be noted that lock_guard has only been available in standard libraries since C++11, yet one can easily implement a lock_guard class.

```
template <class T>
void ThreadSafeQueue::Pushback(T element)
{
   _mutex.Lock();
   DoPushBack(T);
   _mutex.Unlock();
}
```

6. Exception Specification Should Be Used Carefully

If a method throws an exception not listed in its **exception specification**, that fault is detected at runtime, and the special function unexpected()is automatically invoked. The default behavior for unexpected is to call terminate(), and the default behavior for terminate() is to call abort(). Therefore, the default behavior for a program with a violated exception specification is to halt. Consider the following code:

```
void f1();                    // might throw anything
void f2() throw(int);           //throws only int
void f2() throw(int)
{
  ...
  f1();                   // legal even though f1 might throw
                          // something besides an int
  ...
}
```

Remember

A function with no exception specification allows all exceptions. A function with an exception specification that has an empty type_id_list, throw (), does not allow any exceptions to be thrown. An exception specification is not part of a function's type.

The above code is legal because this kind of situation might arise when integrating old code that lacks exception specification, with new code. But if f1() throws something other than int the program will terminate because f2() is not allowed throw anything other than int. It is a bit hard to stop this problem from arising, but there are many ways to handle this if it does. Reference 1: Item 14 details remedies for this problem.

7. Exceptions Should Be Re-Thrown With Re-Throw

There are two ways to propagate a caught exception to callers. Consider the following two code blocks:

```
catch (Widget& w)                  // catch Widget exceptions
{
    ...                          // handle the exception
    throw;                          // rethrow the exception so it
continues to propagate
}
catch (Widget& w)                  // catch Widget exceptions
```

```
{
    ...                             // handle the exception
     throw w;                            // propagate a copy of the
}
```

The only difference between these two blocks is that the first one re-throws the current exception, while the second one throws a new copy of the current exception. There are two problems with the second case. One is the performance cost because of the copy operation. The other thing is the slicing problem. If the exception object is a child type of Widget, only the Widget part of the exception object is re-thrown as the copy is always based on the static type.

8. Catch Clause for the Base Class Should Be Placed After the Catch Clause for the Derived Class

When an exception is thrown, catch clauses are matched in the order they appear in the code. An exception object's type also matches with its super types in catch clauses, because child type is also a subset of super type. Therefore, when a child object's exception is thrown, if a super type catch clause appears first in the code that will be executed, even though the catch clause for the child type is there after the super type's catch clause.

6.2 EXCEPTION HANDLING OVER TRADITIONAL ERROR HANDLING

Following are main advantages of exception handling over traditional error handling.

1) *Separation of Error Handling code from Normal Code:* In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.
2) *Functions/Methods can handle any exceptions they choose:* A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller. In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).
3) *Grouping of Error Types:* In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

Following Example

1) Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks.

```
#include <iostream>
using namespace std;

int main()
{
   int x = -1;

   // Some code
   cout << "Before try \n";
   try {
      cout << "Inside try \n";
      if (x < 0)
      {
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }
   catch (int x ) {
      cout << "Exception Caught \n";
   }

   cout << "After catch (Will be executed) \n";
   return 0;
}
```

Run on IDE

Output:

```
Before try
Inside try
```

```
Exception Caught
After catch (Will be executed)
```

2) There is a special catch block called ‘catch all’ catch (…) that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so catch(…) block will be executed.

```
#include <iostream>
using namespace std;

int main()
{
    try  {
       throw 10;
    }
    catch (char *excp)  {
        cout << “Caught “ << excp;
    }
    catch (...)  {
        cout << “Default Exception\n”;
    }
    return 0;
}
Run on IDE
```

Output:

```
Default Exception
```

3) Implicit type conversion doesn’t happen for primitive types. For example, in the following program ‘a’ is not implicitly converted to int

```
#include <iostream>
using namespace std;

int main()
{
    try  {
        throw ‘a’;
    }
    catch (int x)  {
        cout << “Caught “ << x;
    }
    catch (...)  {
        cout << “Default Exception\n”;
    }
    return 0;
}
Run on IDE
```

Output:

```
Default Exception
```

Keyword

iostream uses the objects cin , cout , cerr , and clog for sending data to and from the standard streams input, output, error (unbuffered), and log (buffered) respectively.

4) If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch a char.

```
<include <iostream#
using namespace std;

int main()
{
    try  {
        throw ‘a’;
    }
    catch (int x)  {
        cout << “Caught “;
    }
    return 0;
}
Run on IDE
```

Output:

```
terminate called after throwing an instance of 'char'

This application has requested the Runtime to terminate it in an
unusual way. Please contact the application's support team for
more information.
```

We can change this abnormal termination behavior by writing our own unexpected function.

5) A derived class exception should be caught before a base class exception. See this for more details.
6) Like Java, C++ library has a standard exception class which is base class for all standard exceptions. All objects thrown by components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type
7) Unlike Java, in C++, all exceptions are unchecked. Compiler doesn't check whether an exception is caught or not (See this for details). For example, in C++, it is not necessary to specify all uncaught exceptions in a function declaration. Although it's a recommended practice to do so. For example, the following program compiles fine, but ideally signature of fun () should list unchecked exceptions.

```
#include <iostream>
using namespace std;

// This function signature is fine by the compiler, but not recommended.
// Ideally, the function should specify all uncaught exceptions and function
// signature should be "void fun(int *ptr, int x) throw (int *, int)"
void fun(int *ptr, int x)
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}
```

```
int main()
{
    try {
       fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}
```

Run on IDE

Output:

```
Caught exception from fun()
```

8) In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using "throw; "

```
#include <iostream>
using namespace std;

int main()
{
    try {
        try  {
            throw 20;
        }
        catch (int n) {
             cout << "Handle Partially ";
             throw;   //Re-throwing an exception
        }
    }
    catch (int n) {
        cout << "Handle remaining ";
    }
    return 0;
}
```

Run on IDE

Output:

```
Handle Partially Handle remaining
```

A function can also re-throw a function using same "throw; ". A function can handle a part and can ask the caller to handle remaining.

9) When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.

```
#include <iostream>
using namespace std;

class Test {
public:
   Test() { cout << "Constructor of Test " << endl; }
  ~Test() { cout << "Destructor of Test "  << endl; }
};

int main() {
  try {
    Test t1;
    throw 10;
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
}
```

Run on IDE

Output:

```
Constructor of Test
Destructor of Test
Caught 10
```

Example shows handling of division by zero exception.

```
#include<iostream>
using namespace std;

double division(int var1, int var2)
{
if (var2 == 0) {
throw “Division by Zero.”;
}
return (var1 / var2);
}

int main()
{
int a = 30;
int b = 0;
double d = 0;

try {
d = division(a, b);
cout << d << endl;
}
catch (const char* error) {
cout << error << endl;
}
return 0;
}
```

6.2.1 Advantage of C++ Exception Handling

Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

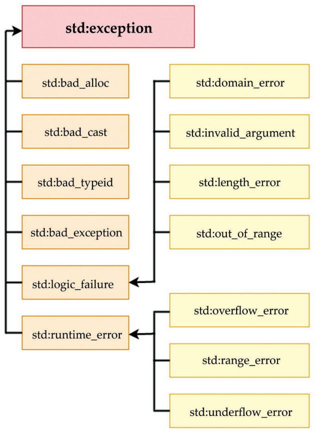
In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Keyword

Standard exceptions: The C++ Standard library provides a base class specifically designed to declare objects to be thrown as exceptions.

C++ Exception Classes

In C++ **standard exceptions** are defined in <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:



All the exception classes in C++ are derived from std::exception class. Let's see the list of C++ common exception classes.

Exception	**Description**
std::exception	It is an exception and parent class of all standard C++ exceptions.
std::logic_failure	It is an exception that can be detected by reading a code.
std::runtime_error	It is an exception that cannot be detected by reading a code.
std::bad_exception	It is used to handle the unexpected exceptions in a c++ program.
std::bad_cast	This exception is generally be thrown by **dynamic_cast.**
std::bad_typeid	This exception is generally be thrown by **typeid.**
std::bad_alloc	This exception is generally be thrown by **new.**

SUMMARY

- Exceptions are runtime anomalies that a program encounters during execution. It is a situation where a program has an unusual condition and the section of code containing it can't handle the problem.
- Multiple catch exception statements are used when a user wants to handle different exceptions differently.
- Sometimes, it may not be possible to design a separate catch block for each kind of exception. In such cases, we can use a single catch statement that catches all kinds of exceptions.
- Exceptions provide many benefits over error codes for error handling. Some of these benefits are:
- Exceptions cannot be silently ignored whereas checking the error code of a method can be ignored by the method caller.
- A function can also re-throw a function using same "throw; ". A function can handle a part and can ask the caller to handle remaining.
- Exception Handling in C++ is a process to handle runtime errors.
- In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

KNOWLEDGE CHECK

1. **Which keyword is used to handle the expection?**
   a. Try
   b. Throw
   c. Catch
   d. None of the above
2. **Which is used to throw a exception?**
   a. Try
   b. Throw
   c. Catch
   d. None of the above
3. **Which exception is thrown by dynamic_cast?**
   a. bad_cast
   b. bad_typeid
   c. bad_exception
   d. bad_alloc
4. **How do define the user-defined exceptions?**
   a. Inherting & overriding exception class functionlity
   b. Overriding class functionlity
   c. Inherting class functionlity
   d. None of the above
5. **We can prevent a function from throwing any exceptions.**
   a. TRUE
   b. FALSE
   c. May Be
   d. Can't Say
6. **In nested try block, if inner catch handler gets executed, then __________?**
   a. Program execution stops immediately.
   b. Outer catch handler will also get executed.
   c. Compiler will jump to the outer catch handler and then executes remaining executable statements of main().
   d. Compiler will execute remaining executable statements of outer try block and then the main().

7. **Return type of uncaught_exception() is ___________.**
    a. int
    b. bool
    c. char *
    d. double
8. **Which of the following statements are true about Catch handler? i) It must be placed immediately after try block T. ii) It can have multiple parameters. iii) There must be only one catch handler for every try block. iv) There can be multiple catch handler for a try block T. v) Generic catch handler can be placed anywhere after try block.**
    a. Only i, iv, v
    b. Only i, ii, iii
    c. Only i, iv
    d. Only i, ii
9. **If inner catch handler is not able to handle the exception then__________ .**
    a. Compiler will look for outer try handler
    b. Program terminates abnormally
    c. Compiler will check for appropriate catch handler of outer try block
    d. None of the above
10. **Which type of program is recommended to include in try block?**
    a. Static memory allocation
    b. Dynamic memory allocation
    c. Const reference
    d. Pointer

REVIEW QUESTIONS

1. Describe Exception handling concept with an example.
2. Explain how we implement exception handling in C++.
3. What is the advantage of exception handling?
4. Explain terminate () and unexpected () function - C++.
5. What is the output of this program?

```
#include <iostream>
using namespace std;
int main()
```

```
{
    char* ptr;
    unsigned long int Test = sizeof(size_t(0) / 3);
    cout << Test << endl;
    try
    {
        ptr = new char[size_t(0) / 3];
        delete[ ] ptr;
    }
    catch (bad_alloc &thebadallocation)
    {
        cout << thebadallocation.what() << endl;
    };
    return 0;
}
```

Check Your Result

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

REFERENCES

1. Bloch, Joshua (2008). "Item 57: Use exceptions only for exceptional situations". Effective Java (Second edition). Addison-Wesley. p. 241. ISBN 978-0-321-35668-0.
2. Graham Hutton, Joel Wright, "Compiling Exceptions Correctly Archived 2014-09-11 at the Wayback Machine". Proceedings of the 7th International Conference on Mathematics of Program Construction, 2004.
3. John Hauser (1996). "Handling Floating-Point Exceptions in Numeric Programs, ACM Transactions on Programming Languages and Systems 18(2)": 139–174.
4. Kiniry, J. R. (2006). "Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application". Advanced Topics in Exception Handling Techniques. Lecture Notes in Computer Science. 4119. pp. 288–300. doi:10.1007/11818502_16. ISBN 978-3-540-37443-5.
5. W.Kahan (July 5, 2005). "A Demonstration of Presubstitution for ∞/∞" (PDF). Archived (PDF) from the original on March 10, 2012.
6. Weimer, W; Necula, G.C. (2008). "Exceptional Situations and Program Reliability" (PDF). ACM Transactions on Programming Languages and Systems. 30 (2). Archived (PDF) from the original on 2015-09-23.

CHAPTER 7

I/O STREAMS

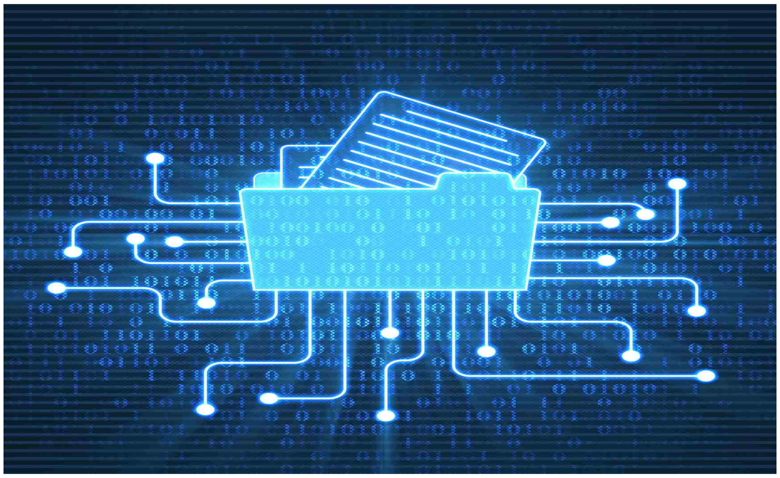
"More good code has been written in languages denounced as "bad" than in languages proclaimed "wonderful" -- much more."

– Bjarne Stroustrup

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Understand basic input/output
2. Explain C++ class hierarchy
3. Describe file stream
4. Perform text file handling
5. Know binary file handling
6. Discuss about error handling during file operations
7. Focus on overloading << and >> operators



INTRODUCTION

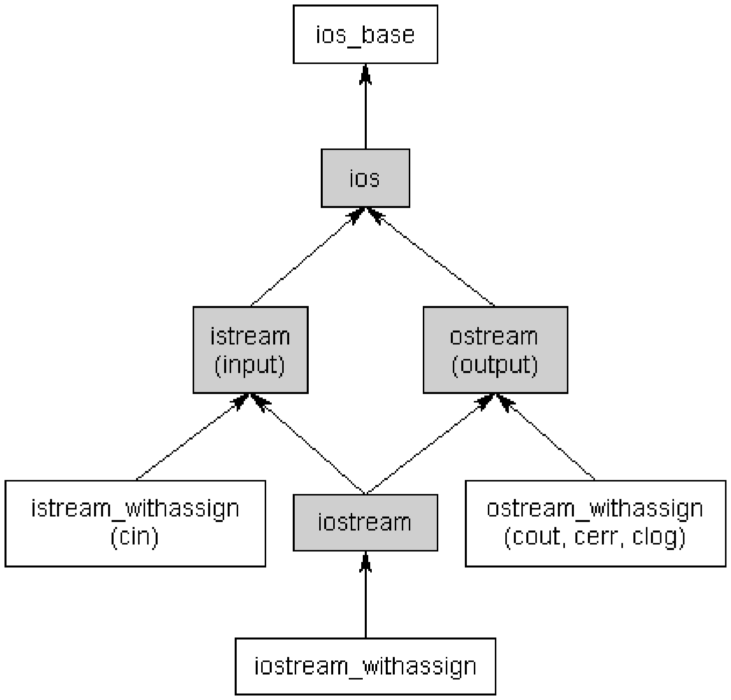
One of the great strengths of C++ is its I/O system, IO Streams. As Bjarne Stroustrup says in his book "The C++

Programming Language", "Designing and implementing a general input/output facility for a programming language is notoriously difficult". He did an excellent job, and the C++ IOstreams library is part of the reason for C++'s success. IO streams provide an incredibly flexible yet simple way to design the input/output routines of any application.

Input and output functionality is not defined as part of the core C++ language, but rather is provided through the C++ standard library (and thus resides in the std namespace).

The iostream library

When you include the iostream header, you gain access to a whole hierarchy of classes responsible for providing I/O functionality (including one class that is actually named iostream). The class hierarchy for the non-file-I/O classes looks like this:



The first thing you may notice about this hierarchy is that it uses multiple inheritance (that thing we told you to avoid if at all possible). However, the iostream library has been designed and extensively tested in order to avoid any of the typical multiple inheritance problems, so you can use it freely without worrying.

Streams

The second thing you may notice is that the word "stream" is used an awful lot. At its most basic, I/O in C++ is implemented with streams. Abstractly, a stream is just a sequence of bytes that can be accessed sequentially. Over time, a stream may produce or consume potentially unlimited amounts of data.

Typically we deal with two different types of streams. Input streams are used to hold input from a data producer, such as a keyboard, a file, or a network. For example, the user may press a key on the keyboard while the program is currently not expecting any input. Rather than ignore the users keypress, the data is put into an input stream, where it will wait until the program is ready for it.

Conversely, output streams are used to hold output for a particular data consumer, such as a monitor, a file, or a printer. When writing data to an output device, the device may not be ready to accept that data yet -- for example, the printer may still be warming up when the program writes data to its output stream. The data will sit in the output stream until the printer begins consuming it.

Some devices, such as files and networks, are capable of being both input and output sources.

The nice thing about streams is the programmer only has to learn how to interact with the streams in order to read and write data to many different kinds of devices. The details about how the stream interfaces with the actual devices they are hooked up to is left up to the environment or operating system.

Input/output in C++

Although the ios class is generally derived from ios_base, ios is typically the most base class you will be working directly with. The ios class defines a bunch of stuff that is common to both input and output streams.

The istream class is the primary class used when dealing with input streams. With input streams, the extraction operator (>>) is used to remove values from the stream. This makes sense: when the user presses a key on the keyboard, the key code is placed in an input stream. Your program then extracts the value from the stream so it can be used.

The ostream class is the primary class used when dealing with output streams. With output streams, the insertion operator (<<) is used to put values in the stream. This also makes sense: you insert your values into the stream, and the data consumer (eg. monitor) uses them.

The iostream class can handle both input and output, allowing bidirectional I/O.

Finally, there are a bunch of classes that end in "_withassign". These stream classes are derived from istream, ostream, and iostream (respectively) with an assignment operator defined, allowing you to assign one stream to another. In most cases, you won't be dealing with these classes directly.

Standard streams in C++

A standard stream is a pre-connected stream provided to a computer program by its environment. C++ comes with four predefined standard stream objects that have already been set up for your use. The first three, you have seen before:

- cin -- an istream_withassign class tied to the standard input (typically the keyboard)
- cout -- an ostream_withassign class tied to the standard output (typically the monitor)
- cerr -- an ostream_withassign class tied to the standard error (typically the monitor), providing unbuffered output
- clog -- an ostream_withassign class tied to the standard error (typically the monitor), providing buffered output

Unbuffered output is typically handled immediately, whereas buffered output is typically stored and written out as a block. Because clog isn't used very often, it is often omitted from the list of standard streams.

IOstreams can be used for a wide variety of data manipulations thanks to the following features:

- A 'stream' is internally nothing but a series of characters. The characters may be either normal characters (char) or wide characters (wchar_t). Streams provide you with a universal character-based interface to any type of storage medium (for example, a file), without requiring you to know the details of how to write to the storage medium. Any object that can be written to one type of stream, can be written to all types of streams. In other words, as long as an object has a stream representation, any storage medium can accept objects with that stream representation.
- Streams work with built-in data types, and you can make user-defined types work with streams by overloading the insertion operator (<<) to put objects into streams, and the extraction operator (>>) to read objects from streams.
- The stream library's unified approach makes it very friendly to use. Using a consistent interface for outputting to the screen and sending files over a network makes life easier. The programs below will show you what is possible.

7.1 BASIC INPUT/OUTPUT

C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen, the keyboard or a file. A *stream* is an entity where a program can either insert or extract characters to/from. There is no need to know details about the media associated to the stream or any of its internal specifications. All we need to know is that streams are a source/destination of characters, and that these characters are provided/accepted sequentially (i.e., one after another).

The standard library defines a handful of stream objects that can be used to access what are considered the standard sources and destinations of characters by the environment where the program runs:

stream	**description**
cin	standard input stream
cout	standard output stream
cerr	standard error (output) stream
clog	standard logging (output) stream

We are going to see in more detail only cout and cin (the standard output and input streams); cerr and clog are also output streams, so they essentially work like cout, with the only difference being that they identify streams for specific purposes: error messages and logging; which, in many cases, in most environment setups, they actually do the exact same thing: they print on screen, although they can also be individually redirected.

7.1.1 Standard output (cout)

On most program environments, the standard output by default is the screen, and the C++ stream object defined to access it is cout.

For formatted output operations, cout is used together with the *insertion operator*, which is written as << (i.e., two "less than" signs).

```
cout << "Output sentence"; // prints Output sentence on screen
cout << 120;               // prints number 120 on screen
cout << x;                 // prints the value of x on screen
```

The << operator inserts the data that follows it into the stream that precedes it. In the examples above, it inserted the literal string Output sentence, the number 120, and the value of variable x into the standard output stream cout. Notice that the sentence in the first statement is enclosed in double quotes (") because it is a string literal, while in the last one, x is not. The double quoting is what makes the difference; when the text is enclosed between them, the text is printed literally; when they are not, the text is interpreted as the identifier of a variable, and its value is printed instead. For example, these two sentences have very different results:

```
cout << "Hello";  // prints Hello
cout << Hello;    // prints the content of variable Hello
```

Multiple insertion operations (<<) may be chained in a single statement:

```
cout << "This " << " is a " << "single C++ statement";
```

This last statement would print the text This is a single C++ statement. Chaining insertions is especially useful to mix literals and variables in a single statement:

```
cout << "I am " << age << " years old and my zipcode is " << zipcode;
```

Assuming the *age* variable contains the value 24 and the *zipcode* variable contains 90064, the output of the previous statement would be:

I am 24 years old and my zipcode is 90064

What cout does not do automatically is add line breaks at the end, unless instructed to do so. For example, take the following two statements inserting into cout:

cout << “This is a sentence.”;

cout << “This is another sentence.”;

The output would be in a single line, without any line breaks in between. Something like:

This is a sentence.This is another sentence.

To insert a line break, a new-line character shall be inserted at the exact position the line should be broken. In C++, a new-line character can be specified as \n (i.e., a backslash character followed by a lowercase n). For example:

```
cout << "First sentence.\n";
cout << "Second sentence.\nThird sentence.";
```

This produces the following output:

First sentence.

Second sentence.

Third sentence.

Alternatively, the endl manipulator can also be used to break lines. For example:

```
cout << "First sentence." << endl;
cout << "Second sentence." << endl;
```

This would print:

First sentence.

Second sentence.

The endl manipulator produces a newline character, exactly as the insertion of '\n' does; but it also has an additional behavior: the stream's buffer (if any) is flushed, which means that the output is requested to be physically written to the device, if it wasn't already. This affects mainly *fully buffered* streams, and cout is (generally) not a *fully buffered* stream. Still, it is generally a good idea to use endl only when flushing the stream would be a feature and '\n' when it would not. Bear in mind that a flushing operation incurs a certain overhead, and on some devices it may produce a delay.

Did You Know?

The iostream library is an object-oriented library that provides input and output functionality using streams.

7.1.2 Standard input (cin)

In most program environments, the standard input by default is the keyboard, and the C++ stream object defined to access it is cin.

For formatted input operations, cin is used together with the extraction operator, which is written as >> (i.e., two "greater than" signs). This operator is then followed by the variable where the extracted data is stored. For example:

```
int age;
cin >> age;
```

The first statement declares a variable of type int called age, and the second extracts from cin a value to be stored in it. This operation makes the program wait for input from cin; generally, this means that the program will wait for the user to enter some sequence with the keyboard. In this case, note that the characters introduced using the keyboard are only transmitted to the program when the ENTER (or RETURN) key is pressed. Once the statement with the extraction operation on cin is reached, the program will wait for as long as needed until some input is introduced.

The extraction operation on cin uses the type of the variable after the >> operator to determine how it interprets the characters read from the input; if it is an integer, the format expected is a series of digits, if a string a sequence of characters, etc.

```cpp
// i/o example

#include <iostream>
using namespace std;

int main ()
{
  int i;
  cout << "Please enter an integer value: ";
  cin >> i;
  cout << "The value you entered is " << i;
  cout << " and its double is " << i*2 << ".\n";
  return 0;
}
```

```
Please enter an integer value: 702
The value you entered is 702 and its double is 1404.
```

As you can see, extracting from cin seems to make the task of getting input from the standard input pretty simple and straightforward. But this method also has a big drawback. What happens in the example above if the user enters something else that cannot be interpreted as an integer? Well, in this case, the extraction operation fails. And this, by default, lets the program continue without setting a value for variable i, producing undetermined results if the value of i is used later.

This is very poor program behavior. Most programs are expected to behave in an expected manner no matter what the user types, handling invalid values appropriately. Only very simple programs should rely on values extracted directly from cin without further checking. A little later we will see how *stringstreams* can be used to have better control over user input.

Extractions on cin can also be chained to request more than one datum in a single statement:

```cpp
cin >> a >> b;
```

This is equivalent to:

```cpp
cin >> a;
cin >> b;
```

In both cases, the user is expected to introduce two values, one for variable a, and another for variable b. Any kind of space is used to separate two consecutive input operations; this may either be a space, a tab, or a new-line character.

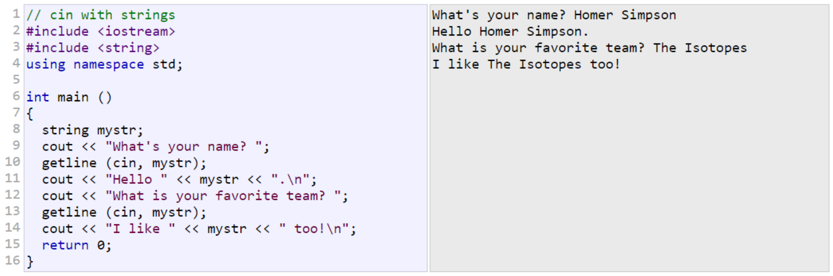
7.1.3 cin and strings

The extraction operator can be used on cin to get strings of characters in the same way as with fundamental data types:

```cpp
string mystring;
cin >> mystring;
```

However, cin extraction always considers spaces (whitespaces, tabs, new-line...) as terminating the value being extracted, and thus extracting a string means to always extract a single word, not a phrase or an entire sentence.

To get an entire line from cin, there exists a function, called **getline**, that takes the stream (cin) as first argument, and the string variable as second. For example:



```
// cin with strings
#include <iostream>
#include <string>
using namespace std;

int main ()
{
  string mystr;
  cout << "What's your name? ";
  getline (cin, mystr);
  cout << "Hello " << mystr << ".\n";
  cout << "What is your favorite team? ";
  getline (cin, mystr);
  cout << "I like " << mystr << " too!\n";
  return 0;
}
```

```
What's your name? Homer Simpson
Hello Homer Simpson.
What is your favorite team? The Isotopes
I like The Isotopes too!
```

Keyword

getline() is a standard library function that is used to read a string or a line from an input stream.

Notice how in both calls to getline, we used the same string identifier (mystr). What the program does in the second call is simply replace the previous content with the new one that is introduced.

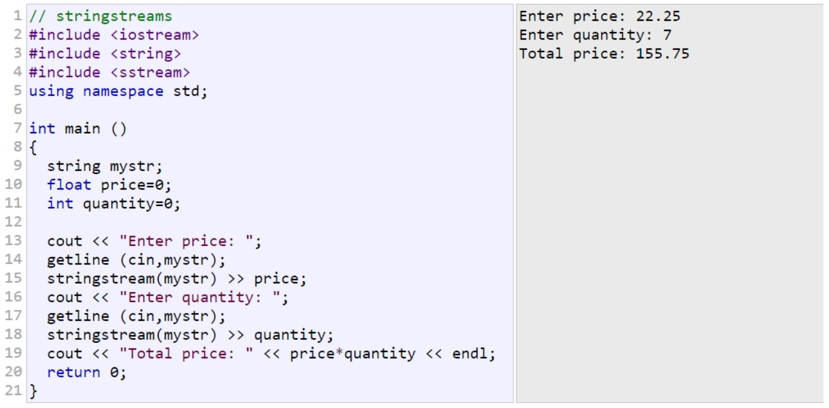
The standard behavior that most users expect from a console program is that each time the program queries the user for input, the user introduces the field, and then presses ENTER (or RETURN). That is to say, input is generally expected to happen in terms of lines on console programs, and this can be achieved by using getline to obtain input from the user. Therefore, unless you have a strong reason not to, you should always use getline to get input in your console programs instead of extracting from cin.

7.1.4 stringstream

The standard header <sstream> defines a type called stringstream that allows a string to be treated as a stream, and thus allowing extraction or insertion operations from/to strings in the same way as they are performed on cin and cout. This feature is most useful to convert strings to numerical values and vice versa. For example, in order to extract an integer from a string we can write:

```
string mystr ("1204");
int myint;
stringstream(mystr) >> myint;
```

This declares a string with initialized to a value of “1204”, and a variable of type int. Then, the third line uses this variable to extract from a stringstream constructed from the string. This piece of code stores the numerical value 1204 in the variable called myint.



```
// stringstreams
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main ()
{
  string mystr;
  float price=0;
  int quantity=0;

  cout << "Enter price: ";
  getline (cin,mystr);
  stringstream(mystr) >> price;
  cout << "Enter quantity: ";
  getline (cin,mystr);
  stringstream(mystr) >> quantity;
  cout << "Total price: " << price*quantity << endl;
  return 0;
}
```

```
Enter price: 22.25
Enter quantity: 7
Total price: 155.75
```

In this example, we acquire numeric values from the *standard input* indirectly: Instead of extracting numeric values directly from cin, we get lines from it into a string object (mystr), and then we extract the values from this string into the variables price and quantity. Once these are numerical values, arithmetic operations can be performed on them, such as multiplying them to obtain a total price.

Remember

Do not confuse between stream extraction operator(>>) and stream insertion operator(<<).

With this approach of getting entire lines and extracting their contents, we separate the process of getting user input from its interpretation as data, allowing the input process to be what the user expects, and at the same time gaining more control over the transformation of its content into useful data by the program.

7.2 C++ CLASS HIERARCHY

A class hierarchy represents a set of hierarchically organized concepts. Base classes act typically as interfaces. They are two uses for interfaces. One is called implementation inheritance and the other interface inheritance.

7.2.1 Interface inheritance

One use of multiple inheritance that is not controversial pertains to *interface inheritance*. In C++, all inheritance is *implementation inheritance*, because everything in a base class, interface and implementation, becomes part of a derived class. It is not possible to inherit only part of a class (the interface alone, say). Private and protected inheritance make it possible to restrict access to members inherited from base classes when used by clients of a derived class object, but this doesn t affect the derived class; it still contains all base class data and can access all non-private base class members.

Interface inheritance, on the other hand, only adds member function *declarations* to a derived class interface and is not directly supported in C++. The usual technique to simulate interface inheritance in C++ is to derive from an *interface class*, which is a class that contains only declarations (no data or function bodies). These declarations will be pure virtual functions, except for the destructor. Here is an example:

```
//: C09:Interfaces.cpp
// Multiple interface inheritance.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class Printable {
public:
virtual ~Printable() {}
virtual void print(ostream&) const = 0;
};

class Intable {
public:
virtual ~Intable() {}
virtual int toInt() const = 0;
};

class Stringable {
public:
virtual ~Stringable() {}
virtual string toString() const = 0;
};

class Able : public Printable, public Intable,
public Stringable {
int myData;
public:
Able(int x) { myData = x; }
void print(ostream& os) const { os << myData; }
int toInt() const { return myData; }
string toString() const {
ostringstream os;
os << myData;
return os.str();
}
};

void testPrintable(const Printable& p) {
p.print(cout);
cout << endl;
}

void testIntable(const Intable& n) {
cout << n.toInt() + 1 << endl;
}

void testStringable(const Stringable& s) {
cout << s.toString() + "th" << endl;
}

int main() {
Able a(7);
testPrintable(a);
```

```
testIntable(a);
testStringable(a);
} ///:~
```

The class Able implements the interfaces Printable, Intable, and Stringable because it provides implementations for the functions they declare. Because Able derives from all three classes, Able objects have multiple is-a relationships. For example, the object a can act as a Printable object because its class, Able, derives publicly from Printable and provides an implementation for print( ). The test functions have no need to know the most-derived type of their parameter; they just need an object that is substitutable for their parameter s type.

As usual, a template solution is more compact:

```
//: C09:Interfaces2.cpp
// Implicit interface inheritance via templates.
#include <iostream>
#include <sstream>
#include <string>
using namespace std;

class Able {
int myData;
public:
Able(int x) { myData = x; }
void print(ostream& os) const { os << myData; }
int toInt() const { return myData; }
string toString() const {
ostringstream os;
os << myData;
return os.str();
}
};

template<class Printable>
void testPrintable(const Printable& p) {
```

```
p.print(cout);
cout << endl;
}

template<class Intable>
void testIntable(const Intable& n) {
cout << n.toInt() + 1 << endl;
}

template<class Stringable>
void testStringable(const Stringable& s) {
cout << s.toString() + “th” << endl;
}

int main() {
Able a(7);
testPrintable(a);
testIntable(a);
testStringable(a);
} ///:~
```

The names Printable, Intable, and Stringable are now just template parameters that assume the existence of the operations indicated in their respective contexts. In other words, the test functions can accept arguments of any type that provides a member function definition with the correct signature and return type; deriving from a common base class in not necessary. Some people are more comfortable with the first version because the type names guarantee by **inheritance** that the expected interfaces are implemented. Others are content with the fact that if the operations required by the test functions are not satisfied by their template type arguments, the error is still caught at compile time. The latter approach is technically a weaker form of type checking than the former (inheritance) approach, but the effect on the programmer (and the program) is the same. This is one form of weak typing that is acceptable to many of today s C++ programmers.

7.2.2 Implementation inheritance

As we stated earlier, C++ provides only implementation inheritance, meaning that you always inherit *everything* from your base classes. This can be good because it frees you from having to implement everything in the derived class, as we had to do with the interface inheritance examples earlier. A common use of multiple inheritance involves using *mixin classes*, which are classes that exist to add capabilities to other classes through inheritance. Mixin classes are not intended to be instantiated by themselves.

Keyword

Inheritance is defined as the capability of a class to derive properties and characteristics from another class .

As an example, suppose we are clients of a class that supports access to a database. In this scenario, you only have a header file available part of the point here is that you don t have access to the source code for the implementation. For illustration, assume the following implementation of a **Database** class:

```
//: C09:Database.h
// A prototypical resource class.
#ifndef DATABASE_H
#define DATABASE_H
#include <iostream>
#include <stdexcept>
#include <string>

struct DatabaseError : std::runtime_error {
DatabaseError(const std::string& msg)
: std::runtime_error(msg) {}
};

class Database {
std::string dbid;
public:
Database(const std::string& dbStr) : dbid(dbStr) {}
virtual ~Database() {}
void open() throw(DatabaseError) {
std::cout << "Connected to " << dbid << std::endl;
}
void close() {
std::cout << dbid << " closed" << std::endl;
}
// Other database functions...
};
#endif // DATABASE_H ///:~
```

We re leaving out actual database functionality (storing, retrieving, and so on), but that s not important here. Using this class requires a database connection string and that you call **Database::open( )** to connect and **Database::close( )** to disconnect:

```
//: C09:UseDatabase.cpp
#include "Database.h"

int main() {
Database db("MyDatabase");
db.open();
// Use other db functions...
db.close();
}
/* Output:
connected to MyDatabase
MyDatabase closed
*/ ///:~
```

In a typical client-server situation, a client will have multiple objects sharing a connection to a database. It is important that the database eventually be closed, but only after access to it is no longer required. It is common to encapsulate this behavior through a class that tracks the number of client entities using the database connection and to automatically terminate the connection when that count goes to zero. To add reference counting to the Database class, we use multiple inheritance to mix a class named Countable into the Database class to create a new class, DBConnection. Here s the Countable mixin class:

```
//: C09:Countable.h
// A "mixin" class.
```

```
#ifndef COUNTABLE_H
#define COUNTABLE_H
#include <cassert>

class Countable {
long count;
protected:
Countable() { count = 0; }
virtual ~Countable() { assert(count == 0); }
public:
long attach() { return ++count; }
long detach() {
return (--count > 0) ? count : (delete this, 0);
}
long refCount() const { return count; }
};
#endif // COUNTABLE_H ///:~
```

It is evident that this is not a standalone class because its constructor is protected; it requires a friend or a derived class to use it. It is important that the destructor is virtual, because it is called only from the delete this statement in detach( ), and we want derived objects to be properly destroyed.[122]

The DBConnection class inherits both Database and Countable and provides a static create( ) function that initializes its Countable subobject. This is an example of the Factory Method design pattern:

```
//: C09:DBConnection.h
// Uses a "mixin" class.
#ifndef DBCONNECTION_H
#define DBCONNECTION_H
#include <cassert>
#include <string>
#include "Countable.h"
#include "Database.h"
using std::string;
```

```
class DBConnection : public Database, public Countable {
DBConnection(const DBConnection&); // Disallow copy
DBConnection& operator=(const DBConnection&);
protected:
DBConnection(const string& dbStr) throw(DatabaseError)
: Database(dbStr) { open(); }
~DBConnection() { close(); }
public:
static DBConnection*
create(const string& dbStr) throw(DatabaseError) {
DBConnection* con = new DBConnection(dbStr);
con->attach();
assert(con->refCount() == 1);
return con;
}
// Other added functionality as desired...
};
#endif // DBCONNECTION_H ///:~
```

We now have a reference-counted database connection without modifying the **Database** class, and we can safely assume that it will not be surreptitiously terminated. The opening and closing is done using the Resource Acquisition Is Initialization (RAII) idiom via the DBConnection constructor and destructor. This makes the DBConnection easy to use:

```
//: C09:UseDatabase2.cpp
// Tests the Countable "mixin" class.
#include <cassert>
#include "DBConnection.h"

class DBClient {
DBConnection* db;
public:
DBClient(DBConnection* dbCon) {
```

```
db = dbCon;
db->attach();
}
~DBClient() { db->detach(); }
// Other database requests using db
};

int main() {
DBConnection* db = DBConnection::create("MyDatabase");
assert(db->refCount() == 1);
DBClient c1(db);
assert(db->refCount() == 2);
DBClient c2(db);
assert(db->refCount() == 3);
// Use database, then release attach from original create
db->detach();
assert(db->refCount() == 2);
} ///:~
```

The call to DBConnection::create( ) calls attach( ), so when we re finished, we must explicitly call detach( ) to release the original hold on the connection. Note that the DBClient class also uses RAII to manage its use of the connection. When the program terminates, the destructors for the two DBClient objects will decrement the reference count (by calling detach( ), which DBConnection inherited from Countable), and the database connection will be closed (because of Countable s virtual destructor) when the count reaches zero after the object c1 is destroyed.

A template approach is commonly used for mixin inheritance, allowing the user to specify at compile time which flavor of mixin is desired. This way you can use different reference-counting approaches without explicitly defining DBConnection twice. Here s how it s done:

```
//: C09:DBConnection2.h
// A parameterized mixin.
#ifndef DBCONNECTION2_H
#define DBCONNECTION2_H
#include <cassert>
```

```
#include <string>
#include "Database.h"
using std::string;

template<class Counter>
class DBConnection : public Database, public Counter {
DBConnection(const DBConnection&); // Disallow copy
DBConnection& operator=(const DBConnection&);
protected:
DBConnection(const string& dbStr) throw(DatabaseError)
: Database(dbStr) { open(); }
~DBConnection() { close(); }
public:
static DBConnection* create(const string& dbStr)
throw(DatabaseError) {
DBConnection* con = new DBConnection(dbStr);
con->attach();
assert(con->refCount() == 1);
return con;
}
// Other added functionality as desired...
};
#endif // DBCONNECTION2_H ///:~
```

Keyword

A **database** is an organized collection of structured information, or data, typically stored electronically in a computer system.

The only change here is the template prefix to the class definition (and renaming **Countable** to **Counter** for clarity). We could also make the **database** class a template parameter (had we multiple database access classes to choose from), but it is not a mixin since it is a standalone class. The following example uses the original Countable as the Counter mixin type, but we could use any type that implements the appropriate interface (attach( ), detach( ), and so on):

```
//: C09:UseDatabase3.cpp
// Tests a parameterized "mixin" class.
#include <cassert>
#include "Countable.h"
#include "DBConnection2.h"

class DBClient {
DBConnection<Countable>* db;
public:
DBClient(DBConnection<Countable>* dbCon) {
db = dbCon;
db->attach();
}
~DBClient() { db->detach(); }
};

int main() {
DBConnection<Countable>* db =
DBConnection<Countable>::create("MyDatabase");
assert(db->refCount() == 1);
DBClient c1(db);
assert(db->refCount() == 2);
DBClient c2(db);
assert(db->refCount() == 3);
db->detach();
assert(db->refCount() == 2);
} ///:~
```

The general pattern for multiple parameterized mixins is simply

```
template<class Mixin1, class Mixin2, , class MixinK>
class Subject : public Mixin1,
public Mixin2,
public MixinK { };
```

7.3 FILE STREAM

File streams in C++ are basically the libraries that are used in the due course of programming. The programmers generally use the iostream standard library in the C++ programming as it provides the cin and cout methods that are used for reading from the input and writing to the output respectively.

In order to read and write from a file, the programmers are generally using the standard C++ library that is known as the fstream.

So far, we have been using the iostream standard library, which provides cin and cout methods for reading from standard input and writing to standard output respectively.

This section will teach you how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types −

Sr.No	Data Type & Description
1	**ofstream** This data type represents the output file stream and is used to create files and to write information to files.
2	**ifstream** This data type represents the input file stream and is used to read information from files.
3	**fstream** This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

7.3.1 Opening a File

A file must be opened before you can read from it or write to it. Either ofstream or fstream object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

void open(const char *filename, ios::openmode mode);

Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Sr.No	Mode Flag & Description
1	**ios::app** Append mode. All output to that file to be appended to the end.
2	**ios::ate** Open a file for output and move the read/write control to the end of the file.
3	**ios::in** Open a file for reading.
4	**ios::out** Open a file for writing.
5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax –

```
ofstream outfile;
outfile.open("file.dat", ios::out | ios::trunc );
```

Similar way, you can open a file for reading and writing purpose as follows –

```
fstream  afile;
afile.open("file.dat", ios::out | ios::in );
```

7.3.2 Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

```
void close();
```

7.3.3 Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an ofstream or fstream object instead of the cout object.

7.3.4 Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an ifstream or fstream object instead of the cin object.

7.3.5 Read and Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen −

```
#include <fstream>
#include <iostream>
using namespace std;

int main () {
   char data[100];

   // open a file in write mode.
   ofstream outfile;
   outfile.open(“afile.dat”);

   cout << “Writing to the file” << endl;
   cout << “Enter your name: “;
   cin.getline(data, 100);

   // write inputted data into the file.
   outfile << data << endl;

   cout << “Enter your age: “;
   cin >> data;
   cin.ignore();

   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();

   return 0;
}
```

When the above code is compiled and executed, it produces the following sample input and output −

```
$./a.out
Writing to the file
Enter your name: Zara
Enter your age: 9
Reading from the file
Zara
9
```

Above examples make use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

7.3.6 File Position Pointers

Both **istream** and ostream provide member functions for repositioning the file-position pointer. These member functions are seekg ("seek get") for istream and seekp ("seek put") for ostream.

Keyword

The **IStream** interface lets you read and write data to stream objects. Stream objects contain the data in a structured storage object, where storages provide the structure.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be ios::beg (the default) for positioning relative to the beginning of a stream, ios::cur for positioning relative to the current position in a stream or ios::end for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are −

```
// position to the nth byte of fileObject (assumes ios::beg)
fileObject.seekg( n );

// position n bytes forward in fileObject
fileObject.seekg( n, ios::cur );

// position n bytes back from end of fileObject
fileObject.seekg( n, ios::end );

// position at end of fileObject
fileObject.seekg( 0, ios::end );
```

7.4 TEXT FILE HANDLING

C++ has the capability of creating, accessing, and editing text files. In this section we will demonstrate how to do this.

7.4.1 FileStream Objects, Header Files, File Access, and Filenames

C++ uses a file stream abstraction for handling text files. A file stream is data that is either going into a text file to be stored or going out of a text file to be loaded into a program. The input (>>) and the output (<<) operators are used to stream data either into a file or into a program.

To perform file input (receiving data from a file), you must include the ifstream header file in your program. To perform file output (send data to a file) you must include the ofstream header file in your program.

There are two primary ways of accessing files from a program — sequential access and random access. When a file is accessed using random access, the program can move back and forth through the file at random. This means a program can read the first record in a file and then immediately move to the fifth record, or the hundredth record.

When a file is accessed sequentially, on the other hand, the program can only move forward through the file, one record at a time. This means that if you want to read the fifth record in the file, you have to read the first through fourth records first.

Sequential file access is easier to work with than random access so we will be working exclusively with sequential files.

Remember

Files are represented on your disk by a file name. Because we are working with text files, the file names should have a .txt extension so the system recognizes them as text files and not some other type of file.

7.4.2 Creating and Storing Data in a Text File

Our first task is to create and store data in a text file. The ofstream library includes a function for creating a new file — open, which takes a string argument consisting of the file name you wan to use.

Files are created in two steps. The first step is to declare the file name and the second step is to call the open function from the new file. Here are the lines needed to create a text file:

```
ofstream outFile;
outFile.open("myfile.txt");
```

The argument to the open function is the full path to the output file you are writing. If you just include the filename, your file will be written in the same file your program resides in. Otherwise, specify a full path before the filename if want the file written somewhere else.

The next step is to write data to the file. Of course, this step will depend on the task you are trying to perform. This next part of the program has the user enter text at a prompt and then writes that data to the file using the file name as the target of the output stream. Here is the code:

```
string quit = "n";
string line;
while (quit != "y") {
  cout << "Enter a line of text: ";
  getline(cin, line);
  outFile << line << endl;
  cout << "Stop entering text (y/n)? ";
  getline(cin, quit);
}
```

The last step, and an important one, is to close the file. While you probably won't lose data if the program ends before the output file is closed, it is a good programming practice to close files after you are finished with them.

Here is the complete program for creating a file and writing data to that file:

```
#include <iostream>
#include <fstream>using namespace std;int main()
{
  ofstream outFile;
  outFile.open("myfile.txt");
  string quit = "n";
  string line;
  while (quit != "y") {
    cout << "Enter a line of text: ";
    getline(cin, line);
    outFile << line << endl;
    cout << "Stop entering text (y/n)? ";
    getline(cin, quit);
  }
  outFile.close();
  return 0;
}
```

Once you've closed a file you are finished with it until you want to open it in order to read the data from it.

7.4.3 Reading Data from a Text File

To read data from a text file, you have to open the file for input. The first thing you need to do is declare an ifstream object to represent an input file. Then you need to open the file by providing the path to the file.

You need to be careful when specifying the file path because if the path has backslashes in it, you have to escape the backslashes so that C++ won't interpret part of the file path as an escaped character.

For example, if the file path includes a \t in the name, C++ will interpret this as the tab character and the system will throw an error. You need to double backslash your paths so that this doesn't happen.

In the example we will show, the file path is:

c:\users\mmcmi\documents\words.txt

Instead, write the path like this:

c:\\users\\mmcmi\\documents\\words.txt.

After the file is open, you can loop through the file and read each line. The file object can act as the condition of the loop, meaning that while there is data in the file to read, the file name returns true (1), and when there is no more data to read, the file name returns false (0).

The last step is to close the file to maintain good software engineering practices, though if you don't do this the operating system will close the file automatically.

Now we're ready to view a complete program that reads a text file from a **hard disk** and displays all the data in the file. The file we are opening and reading is a dictionary of words.

Keyword

A **hard disk** is an electro-mechanical data storage device that stores and retrieves digital data using magnetic storage and one or more rigid rapidly rotating platters coated with magnetic material.

Here is the program:

```
#include <iostream>
#include <fstream>using namespace std;int main()
{
  ifstream inFile;
  inFile.open("c:\\users\\mmcmi\\documents\\words.txt");
  string word;
  while (inFile) {
    getline(inFile, word);
    cout << word << endl;
  }
  inFile.close();
  return 0;
}
```

Here is a partial display of the output from this file:

```
…
bloggers
blogging
blogs
blond
blonde
blood
bloody
bloom
bloomberg
blow
blowing
…
```

A more concise way to write the program above is to combine the file check in the condition with getting the next word from the file. Here's how that looks:

```
int main()
{
  ifstream inFile;
  inFile.open("c:\\users\\mmcmi\\documents\\words.txt");
  string word;
  while (getline(inFile, word)) {
    cout << word << endl;
  }
  inFile.close();
  return 0;
}
```

When the loop reaches the end of the file, getline will essentially return a false value and the loop stops.

7.4.4 Working with Numbers in a Text File

When numbers are stored in a texts file they are stored as strings, so when they are read into a program they have to be converted to the proper numeric data type. Here is an example that reads a set of test grades from a text file and computes the average of the grades:

```
#include <iostream>
#include <fstream>
#include <string>using namespace std;int main()
{
  ifstream inFile;
  inFile.open("grades.txt");
  string strGrade;
  int grade, total, numGrades;
  total = 0;
  numGrades = 0;
  while (getline(inFile, strGrade)) {
    cout << strGrade << " ";
    numGrades++;
    grade = stoi(strGrade);
    total += grade;
  }
  inFile.close();
  double average = static_cast<double>(total) / numGrades;
  cout << endl << endl << "The average test grade is: "
       << average << endl;
  return 0;
}
```

This program uses the stoi function to convert the string grade into an integer.

Working with a file that contains the grades — 82, 91, 77, 84, 91, 63 — the output from this program is:

```
82 91 77 84 91 63The average test grade is: 81.3333
```

7.4.5 Appending Data to a File

Besides opening a file for input or output, you can also open a file in order to append new data to it. You do this by adding the constant ios::app as a second argument to the open function. The following program demonstrates how to append data to an existing file of grades, the same file we used in the immediate example above:

```
int main()
{
  ofstream outFile;
  outFile.open("grades.txt", ios::app);
  int grade1 = 91;
  int grade2 = 87;
  int grade3 = 93;
  outFile << grade1 << endl;
  outFile << grade2 << endl;
  outFile << grade3 << endl;
  outFile.close();
  ifstream inFile;
  inFile.open("grades.txt");
  int grade;
  while (inFile >> grade) {
    cout << grade << " ";
  }
  inFile.close();
  return 0;
}
```

7.5 BINARY FILE HANDLING

Let's now learn about basic operations on files. The operations that you have learnt so far are applicable on both text and binary files. The only difference is that while working with binary files, you do need to prefix file modes with ios::binary at the time of opening the file. You have learnt how to create files (open them in ios::out or ios::app mode which will create a file if it doesn't exist already), how to read records from binary files (call read() function through stream object to which file is attached), how to write records in binary files (call write() function through stream object which file is attached.).

Here, in this section, we will perform the following basic operations on binary files:

- searching
- appending data
- inserting data in sorted file
- deleting a record
- modifying data

Let's start with searching operation.

7.5.1 Searching in C++

We can perform search in a binary file opened in input mode by reading each record then checking whether it is our desired record or not. For instance, if you want to search for a record for a student having rollno as 1 in file marks.dat, you can implement this search process in C++ in two manners :

- with records implemented through structures.
- with records implemented through classes.

When records are implemented through structures, you can perform search as demonstrated in the following examples :

```
struct student
{
    int rollno;
    char name[20];
    char branch[3];
    float marks;
    char grade;
}stud1;
```

```
ifstream fin("marks.dat", ios::in | ios::binary);
:                        // Read rollno to be searched for

while(!fin.eof())
{
   fin.read((char *)&stud1, sizeof(stud1));    // read record
   if(stud1.rollno == rn)                      // if true, record is found
   {
          :              // process desired record here

          found = 'y';       // after processing you may jump from the
          break;             // loop employed form searching purpose
   }
}

if(found == 'n')                    // record not found
{
   :                         // display error message here or process as desired
}
```

Let's take an example for complete understanding on searching operation in C++.

C++ Searching Example

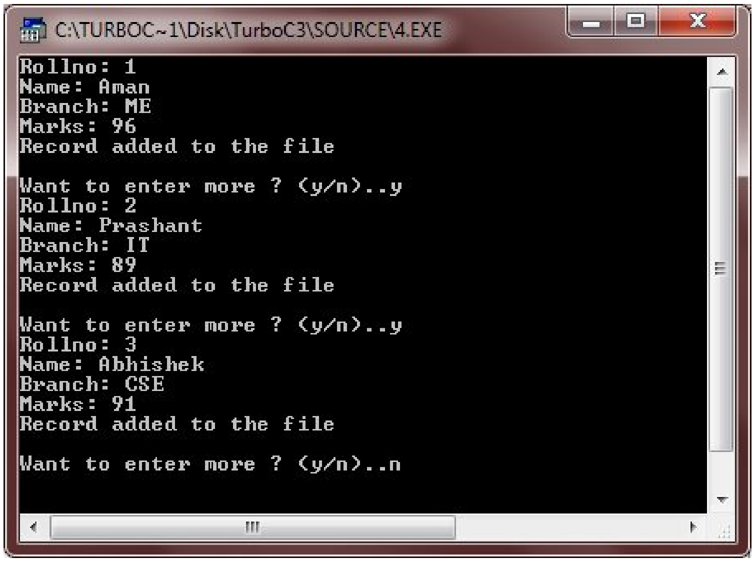
Here is an example program, demonstrating the searching operation on binary files in C++.

```
/* C++ Basic Operations on Binary Files
 * This program demonstrates the searching
 * operation in a C++ program. Here the
 * searching operations performed, on
 * the records implemented through structures
 */

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

```
#include<stdlib.h>

class student
{
    int rollno;
    char name[20];
    char branch[3];
    float marks;
    char grade;

    public:
            void getdata()
            {
                    cout<<"Rollno: ";
                    cin>>rollno;
                    cout<<"Name: ";
                    cin>>name;
                    cout<<"Branch: ";
                    cin>>branch;
                    cout<<"Marks: ";
                    cin>>marks;

                    if(marks>=75)
                    {
                            grade = 'A';
                    }
                    else if(marks>=60)
                    {
                            grade = 'B';
                    }
                    else if(marks>=50)
                    {
                            grade = 'C';
                    }
                    else if(marks>=40)
                    {
                              grade = ‘D’;
                    }
                    else
                    {
                              grade = ‘F’;
                    }
          }

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

          int getrno()
          {
                    return rollno;
          }
}stud1;

void main()
{
    clrscr();

    fstream fio(“marks.dat”, ios::in | ios::out);
    char ans=’y’;
    while(ans==’y’ || ans==’Y’)
    {
          stud1.getdata();
          fio.write((char *)&stud1, sizeof(stud1));
          cout<<"Record added to the file\n";
          cout<<"\nWant to enter more ? (y/n)..";
          cin>>ans;
}

clrscr();
int rno;
long pos;
char found='f';

cout<<"Enter rollno of student to be search for: ";
cin>>rno;

fio.seekg(0);
while(!fio.eof())
{
          pos=fio.tellg();
          fio.read((char *)&stud1, sizeof(stud1));
          if(stud1.getrno() == rno)
          {
                    stud1.putdata();
                    fio.seekg(pos);
                    found='t';
                    break;
          }
}
if(found=='f')
{
          cout<<"\nRecord not found in the file..!!\n";
          cout<<"Press any key to exit...\n";
          getch();
          exit(2);
}
```

```
        fio.close();
        getch();
    }
```

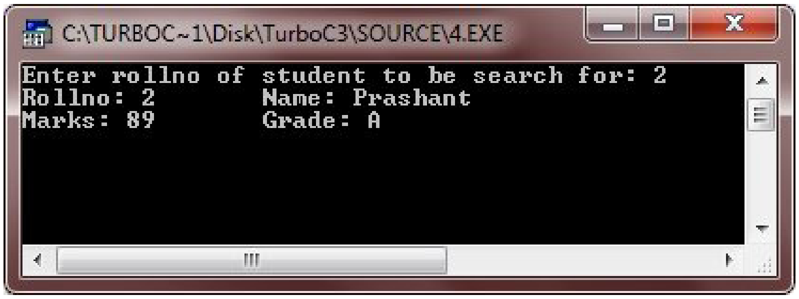
Here are the sample run of the above C++ program:



C:\TURBOC~1\Disk\TurboC3\SOURCE\4.EXE

```
Rollno: 1
Name: Aman
Branch: ME
Marks: 96
Record added to the file

Want to enter more ? (y/n)..y
Rollno: 2
Name: Prashant
Branch: IT
Marks: 89
Record added to the file

Want to enter more ? (y/n)..y
Rollno: 3
Name: Abhishek
Branch: CSE
Marks: 91
Record added to the file

Want to enter more ? (y/n)..n
```

After entering the three records, press n and then ENTER. Now to search for a specific roll number. Enter the roll number of a student and press enter to perform searching operation. Here is the sample run:



C:\TURBOC~1\Disk\TurboC3\SOURCE\4.EXE

```
Enter rollno of student to be search for: 2
Rollno: 2        Name: Prashant
Marks: 89        Grade: A
```

When the records are implemented through classes, then you may need to add an additional accessor function in public: section of the class, that returns the value of the data member to which the user-specified search value is to be compared. With structures, this was not required as all the data members were public by default and hence accessible through structure variable. On the other hand, through an object, private data members can't be accessed. Therefore, to read value of a private data member, a function is added in the public section, that returns this value. The rest of the processing is just similar to that of the structures. Here is an example code fragment demonstrates this:

```
class student
{
    int rollno;
    char name[20];
    char branch[3];
    float marks;
    char grade;

    public:
            void getdata();
            void putdata();
            int getrno()    // this function returns the value of private member rollno
            {
                    return rollno;
            }
}stud1;

ifstream fin("marks.dat", ios::in);
:           // read rollno to be search for

while(!fin.eof())
{
    fin.read((char *)&stud1, sizeof(stud1));
    if(stud1.getrno() == rno)             // retrieve value of private member and
compare
    {
            :                // process desired record here

            found = 't';
            break;
    }
}
if(found=='n')     // if record not found
{
    :                  // display message for not found
}
```

Let's take an example for the complete understanding on the searching operations on the records implemented through classes in C++.

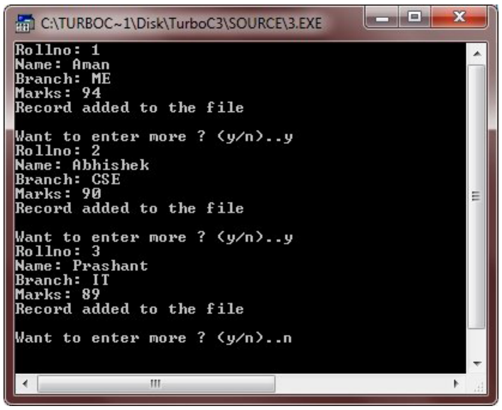
```
/* C++ Basic Operations on Binary Files
 * This program demonstrates the searching
 * operation in a C++ program. Here the
 * searching operations performed, on
 * the records implemented through classes
 */

#include<fstream.h>
#include<conio.h>
#include<stdlib.h>

class student
{
    int rollno;
    char name[20];
    char branch[3];
    float marks;
    char grade;

    public:
            void getdata()
            {
                    cout<<"Rollno: ";
                    cin>>rollno;
                    cout<<"Name: ";
                    cin>>name;
```

```
          cout<<"Branch: ";
          cin>>branch;
          cout<<"Marks: ";
          cin>>marks;

          if(marks>=75)
          {
                    grade = 'A';
          }
          else if(marks>=60)
          {
                    grade = 'B';
          }
          else if(marks>=50)
          {
                    grade = 'C';
          }
          else if(marks>=40)
          {
                    grade = 'D';
          }
          else
          {
                    grade = 'F';
          }
}

void putdata()
{
          cout<<name<<", rollno "<<rollno<<" has ";
          cout<<marks<<"% marks and "<<grade<<" grade."<<"\n";
}
```

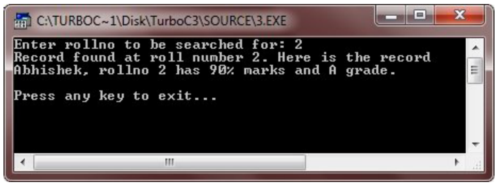
```
            int getrno()
            {
                    return rollno;
            }
}stud1;

void main()
{
    clrscr();

    ofstream fout("marks.dat", ios::out);
    char ans='y';
    while(ans=='y' || ans=='Y')
    {
            stud1.getdata();
            fout.write((char *)&stud1, sizeof(stud1));
            cout<<"Record added to the file\n";
            cout<<"\nWant to enter more ? (y/n)..";
            cin>>ans;
    }
    fout.close();

    clrscr();
    int rno;
    char found;
    ifstream fin("marks.dat", ios::in);

    found = 'n';
    cout<<"Enter rollno to be searched for: ";
    cin>>rno;

    while(!fin.eof())        // end-of-file used here
    {
```

```
                fin.read((char *)&stud1, sizeof(stud1));
                if(stud1.getrno() == rno)
                {
                        cout<<"Record found at roll number "<<rno<<". Here is the
record\n";

                        stud1.putdata();
                        found = 't';
                        break;
                }
        }
        if(found=='n')
        {
                cout<<"\nRecord not found at this roll number..!!\n";
                cout<<"Press any key to exit...\n";
                getch();
                exit(2);
        }

        fin.close();
        cout<<"\nPress any key to exit...\n";
        getch();
    }
```

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



```
Rollno: 1
Name: Aman
Branch: ME
Marks: 94
Record added to the file

Want to enter more ? (y/n)..y
Rollno: 2
Name: Abhishek
Branch: CSE
Marks: 90
Record added to the file

Want to enter more ? (y/n)..y
Rollno: 3
Name: Prashant
Branch: IT
Marks: 89
Record added to the file

Want to enter more ? (y/n)..n
```



```
Enter rollno to be searched for: 2
Record found at roll number 2. Here is the record
Abhishek, rollno 2 has 90% marks and A grade.

Press any key to exit...
```

7.5.2 Appending Data in C++

To append data in a file, the file is opened with the following two specifications :

- file is opened in output mode
- file is opened in ios::app mode

Once the file gets opened in ios::app mode, the previous records/information is retained and new data gets appended to the file.

Let's see the following **code fragment** that appends new records to file marks.dat :

Keyword

Code fragments are pieces of code that occur regularly in programs and for that reason have been collected for reuse.

```
class student
{
        :
}stud1;
ofstream fout;
fout.open("marks.dat", ios::app } ios::binary);
/* Repeat following lines as many times */
{
        stud1.getdata();                        // Read Record
        fout.write((char *) & stud1, sizeof(stud1));    // record
appended
}
```

C++ Appending Data Example

Here is an example program demonstrates, how to append data in a file in C++.

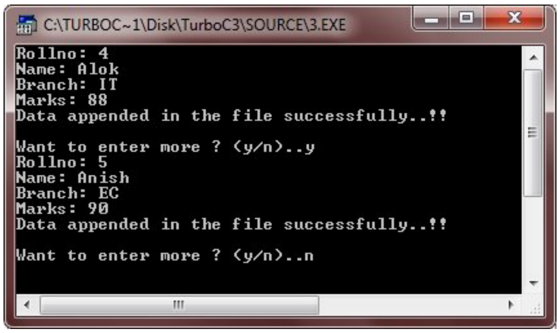
/* C++ Basic Operations on Binary Files

```
 * This program demonstrates, how to
 * append data in a file in C++ */

#include<fstream.h>
#include<conio.h>
#include<stdlib.h>

class student
{
        int rollno;
        char name[20];
        char branch[3];
        float marks;
        char grade;

        public:
                void getdata()
                {
                        cout<<"Rollno: ";
                        cin>>rollno;
                        cout<<"Name: ";
                        cin>>name;
                        cout<<"Branch: ";
                        cin>>branch;
                        cout<<"Marks: ";
                        cin>>marks;

                        if(marks>=75)
                        {
                                grade = 'A';
                        }
                        else if(marks>=60)
                        {
```

```
                grade = 'B';
            }
            else if(marks>=50)
            {
                grade = 'C';
            }
            else if(marks>=40)
            {
                grade = 'D';
            }
            else
            {
                grade = 'F';
            }
        }

        void putdata()
        {
            cout<<name<<", rollno "<<rollno<<" has ";
            cout<<marks<<"% marks and "<<grade<<" grade."<<"\n";
        }

        int getrno()
        {
            return rollno;
        }
}stud1;

void main()
{
    clrscr();

    ofstream fout("marks.dat", ios::app);
```

```
        char ans='y';
        while(ans=='y' || ans=='Y')
        {
                stud1.getdata();
                fout.write((char *)&stud1, sizeof(stud1));
                cout<<"Data appended in the file successfully..!!\n";
                cout<<"\nWant to enter more ? (y/n)..";
                cin>>ans;
        }

        fout.close();
        cout<<"\nPress any key to exit...\n";
        getch();
}
```

Here is the sample run of this C++ program:



```
Rollno: 4
Name: Alok
Branch: IT
Marks: 88
Data appended in the file successfully..!!

Want to enter more ? (y/n)..y
Rollno: 5
Name: Anish
Branch: EC
Marks: 90
Data appended in the file successfully..!!

Want to enter more ? (y/n)..n
```

7.5.3 C++ Inserting Data in Sorted File

To insert data in a sorted file, firstly, its appropriated position is determined and then records in the file prior to this determined position are copied to temporary file, followed by the new record to be inserted and then the rest of the records from the file are also copied.

For example, records in marks.dat are sorted in ascending order on the basis of rollno. Assuming that there are about 10 records in the file marks.dat. Now a new record with rollno 5 is to be inserted. It will be accomplished as follows :

(i) Determining the appropriate position. If the rollno of new record say NREC in which rollno 5 is lesser than the rollno of very first record, then the position is 1, where the new record is to be inserted as it will be inserted in the beginning of the file to maintain the sorted order.

If the rollno of new record (5 here) satisfies following condition for two consecutive records say (pth and (p + 1)th)

if prev.getrno() <= NREC.getrno() && (NREC.getrno() <= next.getrno())

then the appropriate position will be position of prev + 1 i.e, p + 1.

And if the rollno of the new record is more than the rollno of last record (say nth record) then the appropriate position will be n+1.

(ii) Copy the records prior to determined position to a temporary file say temp.dat.

(iii) Append the new record in the temporary file temp.dat.

(iv) Now append the rest of the records in temporary file temp.dat.

(v) Delete the file marks.dat by using the following code.

remove("marks.dat");

(vi) Now, rename the file temp.dat as marks.dat as follows :

rename("temp.dat", "marks.dat");

Let's take an example for the complete understanding on inserting data in a sorted file in C++.

C++ Inserting Data in Sorted File Example

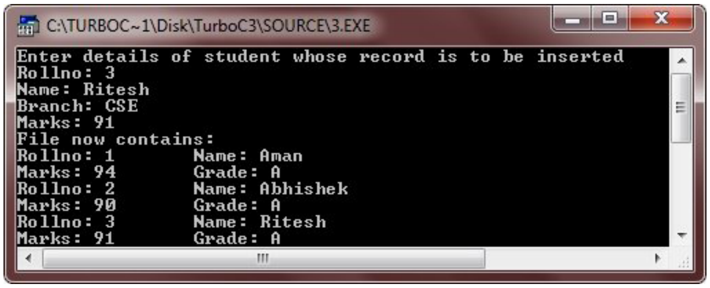
Here is an example program demonstrating, how to insert data in a sorted file in C++

```
/* C++ Basic Operations on Binary Files
 * This program demonstrates how to insert
 * data in a sorted file in C++ */

#include<fstream.h>
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>

class student
{
    int rollno;
```

```
char name[20];
char branch[3];
float marks;
char grade;

public:
        void getdata()
        {
                cout<<"Rollno: ";
                cin>>rollno;
                cout<<"Name: ";
                cin>>name;
                cout<<"Branch: ";
                cin>>branch;
                cout<<"Marks: ";
                cin>>marks;

                if(marks>=75)
                {
                        grade = 'A';
                }
                else if(marks>=60)
                {
                        grade = 'B';
                }
                else if(marks>=50)
                {
                        grade = 'C';
                }
                else if(marks>=40)
                {
                        grade = 'D';
                }
                else
                {
                        grade = 'F';
                }
        }

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

        int getrno()
        {
                return rollno;
        }
}stud1, stud;

void main()
{
    clrscr();
    ifstream fin("marks.dat", ios::in);
    ofstream fout("temp.dat", ios::out);
    char last='y';
    cout<<"Enter details of student whose record is to be inserted\n";
    stud1.getdata();
    while(!fin.eof())
    {
        fin.read((char *)&stud, sizeof(stud));
        if(stud1.getrno()<=stud.getrno())
        {
                fout.write((char *)&stud1, sizeof(stud1));
                last = 'n';
                break;
            }
            else
            {
                fout.write((char *)&stud, sizeof(stud));
            }
    }
    if(last == 'y')
    {
            fout.write((char *)&stud1, sizeof(stud1));
    }
    else if(!fin.eof())
    {
            while(!fin.eof())
            {
                fin.read((char *)&stud, sizeof(stud));
                fout.write((char *)&stud, sizeof(stud));
            }
    }
    fin.close();
    fout.close();
    remove("marks.dat");
    rename("temp.dat", "marks.dat");
    fin.open("marks.dat", ios::in);
    cout<<"File now contains:\n";
    while(!fin.eof())
    {
            fin.read((char *)&stud, sizeof(stud));
            if(fin.eof())
            {
                break;
            }
            stud.putdata();
```

```
        }
        fin.close();
        getch();
    }
```

Here is the sample run of the above C++ program:



C:\TURBOC~1\Disk\TurboC3\SOURCE\3.EXE

```
Enter details of student whose record is to be inserted
Rollno: 3
Name: Ritesh
Branch: CSE
Marks: 91
File now contains:
Rollno: 1               Name: Aman
Marks: 94               Grade: A
Rollno: 2               Name: Abhishek
Marks: 90               Grade: A
Rollno: 3               Name: Ritesh
Marks: 91               Grade: A
```

7.5.4 Deleting a Record in C++

To delete a record, following procedure is carried out :

(i) Firstly, determine the position of the record to be deleted, by performing a search in the file.

(ii) Keep copying the records other than the record to be delete in a temporary file say temp.dat.

(iii) Do not copy the record to be deleted to temporary file, temp.dat.

(iv) Copy rest of the records to temp.dat.

(v) Delete original file say marks.dat as :

remove("marks.dat");

(vi) Rename temp.dat as marks.dat as :

rename("temp.dat", "marks.dat");

For example program, just copy the above code, and make changes as told in the above steps to process the deletion of record.

7.5.5 Modifying Data in C++

To modify a record, the file is opened in I/O mode and an important step is performed that gives the beginning address of record being modified. After the record is modified in memory, the file pointer is once again placed at the beginning position of this record and then record is rewritten. Following code illustrates it :

```
class student
{
          :
          void modify();
}stud1;

fstream fio("marks.dat", ios::in | ios::out);
/* Read rollno whose data is to be modified */
long pos;
while(!fio.eof())
{
          pos = fio.tellg()          // determine the beginning position of record
          fio.read((char *) & stud1, sizeof(stud1));
          if(stud1.getrno() == rn)        // this is the record to be modified
          {
                    stud1.modify();          // get the new data
                    fio.seekg(pos);      // place file pointer at the beginning record position
                    fio.write((char *) & stud1, sizeof(stud1));      // now write the modified
record
          }
}
```

For example program, just copy the above code, and make changes, as shown in the above code fragment to process the modification of data in a file.

7.6 ERROR HANDLING DURING FILE OPERATIONS

It's quite common that errors may occur during file operations. There may have different reasons for arising errors while working with files. The following are the common problems that lead to errors during file operations.

- When trying to open a file for reading might not exist.
- When trying to read from a file beyond its total number of characters.
- When trying to perform a read operation from a file that has opened in write mode.
- When trying to perform a write operation on a file that has opened in reading mode.

- When trying to operate on a file that has not been open.

During the file operations in C++, the status of the current file stream stores in an integer flag defined in ios class. The following are the file stream flag states with meaning.

Flag Bit	Meaning
badbit	1 when a fatal I/O error has occurred, 0 otherwise.
failbit	1 when a non-fatal I/O error has occurred, 0 otherwise
goodbit	1 when no error has occurred, 0 otherwise
eofbit	1 when end-of-file is encountered, 0 otherwise.

We use the above flag bits to handle the errors during the file operations.

The C++ programming language provides several built-in functions to handle errors during file operations.

The following are the built-in functions to handle file errors.

Function	Return Value
int bad()	It returns a non-zero (true) value if an invalid operation is attempted or an unrecoverable error has occurred. Returns zero if it may be possible to recover from any other error reported and continue operations.
int fail( )	It returns a non-zero (true) value when an input or output operation has failed.
int good()	It returns a non-zero (true) value when no error has occurred; otherwise returns zero (false).
int eof( )	It returns a non-zero (true) value when end-of-file is encountered while reading; otherwise returns zero (false).

7.6.1 int bad( )

The **bad( )** function returns a non-zero (true) value if an invalid operation is attempted or an unrecoverable error has occurred. Returns zero if it may be possible to recover from any other error reported and continue operations.

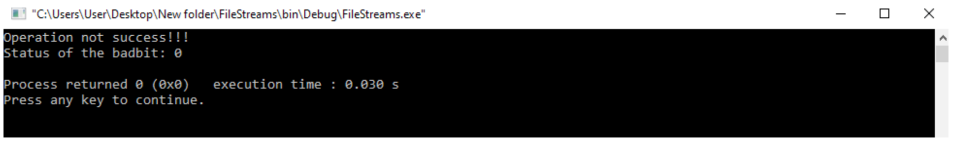
Let's look at the following code.

Example - Code to illustrate the bad( ) function

```
#include <iostream>
#include <fstream>

using namespace std;
```

```
int main()
{
    fstream file;
    file.open("my_file.txt", ios::out);

    string data;

    file >> data;

    if(!file.bad()){
        cout << "Operation not success!!!" << endl;
        cout << "Status of the badbit: " << file.bad() << endl;
    }
    else {
        cout << "Data read from file - " << data << endl;
    }

    return 0;
}
```

Output



```
Operation not success!!!
Status of the badbit: 0

Process returned 0 (0x0)   execution time : 0.030 s
Press any key to continue.
```

7.6.2 int fail( )

The **fail( )** function returns a non-zero value when an input or output operation has failed.

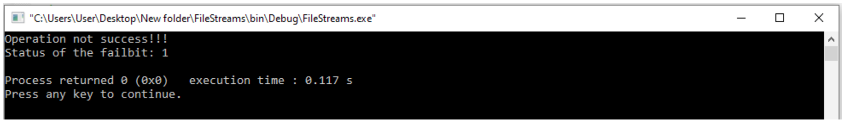
Let's look at the following code.

Example - Code to illustrate the fail( ) function

```
#include <iostream>
#include <fstream>
```

```
using namespace std;

int main()
{
    fstream file;
    file.open("my_file.txt", ios::out);

    string data;

    file >> data;

    if(file.fail()){
        cout << "Operation not success!!!" << endl;
        cout << "Status of the failbit: " << file.fail() << endl;
    }
    else {
        cout << "Data read from file - " << data << endl;
    }

    return 0;
}
```

Output



```
"C:\Users\User\Desktop\New folder\FileStreams\bin\Debug\FileStreams.exe"
Operation not success!!!
Status of the failbit: 1

Process returned 0 (0x0)   execution time : 0.117 s
Press any key to continue.
```

7.6.3 int eof( )

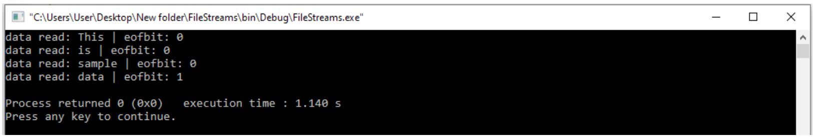
The **eof( )** function returns a non-zero (true) value when end-of-file is encountered while reading; otherwise returns zero (false).

Let's look at the following code.

Example - Code to illustrate the eof( ) function

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    fstream file;
    file.open("my_file.txt", ios::in);

    string data;

    while(!file.eof()){
        file >> data;
        cout << "data read: " << data << " | eofbit: " << file.eof() << endl;
    }

    return 0;
}
```

Output



"C:\Users\User\Desktop\New folder\FileStreams\bin\Debug\FileStreams.exe"

```
data read: This | eofbit: 0
data read: is | eofbit: 0
data read: sample | eofbit: 0
data read: data | eofbit: 1

Process returned 0 (0x0)   execution time : 1.140 s
Press any key to continue.
```

7.6.4 int good( )

The **good( )** function returns a non-zero (true) value when no error has occurred; otherwise returns zero (false).

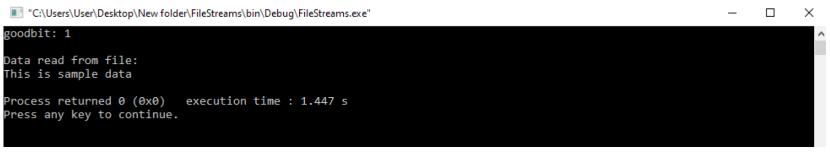
Let's look at the following code.

Example - Code to illustrate the good( ) function

```
#include <iostream>
#include <fstream>
```

```
using namespace std;

int main()
{
    fstream file;
    file.open("my_file.txt", ios::in);

    cout << "goodbit: " << file.good() << endl;

    string data;

    cout << endl << "Data read from file:" << endl;
    while(!file.eof()){
        file >> data;
        cout << data << " ";
    }
    cout << endl;

    return 0;
}
```

Output



"C:\Users\User\Desktop\New folder\FileStreams\bin\Debug\FileStreams.exe"

```
goodbit: 1

Data read from file:
This is sample data

Process returned 0 (0x0)   execution time : 1.447 s
Press any key to continue.
```

7.6.5 int clear( )

The **clear( )** function used to reset the error state so that further operations can be attempted.

- The above functions can be summarized as eof() returns true if eofbit is set; bad() returns true if badbit is set. The fail() function returns true if failbit is set; the good() returns true there are no errors. Otherwise, they return false.

- All the built-in function returns either non-zero to indicate true or zero to indicate false.

7.7 OVERLOADING << AND >> OPERATORS

C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<. The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.

Here, it is important to make operator overloading function a friend of the class because it would be called without creating an object.

Following example explains how extraction operator >> and insertion operator <<.

```
#include <iostream>
using namespace std;

class Distance {
   private:
      int feet;              // 0 to infinite
      int inches;             // 0 to 12

   public:
      // required constructors
      Distance() {
         feet = 0;
         inches = 0;
      }
      Distance(int f, int i) {
         feet = f;
         inches = i;
      }
        friend ostream &operator<<( ostream &output, const
Distance &D ) {
```

Remember

To check for such errors and to ensure smooth processing, C++ file streams inherit 'stream-state' members from the ios class that store the information on the status of a file that is being currently used.

```
            output << "F : " << D.feet << " I : " << D.inches;
            return output;
        }

        friend istream &operator>>( istream  &input, Distance &D ) {
            input >> D.feet >> D.inches;
            return input;
        }
};

int main() {
    Distance D1(11, 10), D2(5, 11), D3;

    cout << "Enter the value of object : " << endl;
    cin >> D3;
    cout << "First Distance : " << D1 << endl;
    cout << "Second Distance :" << D2 << endl;
    cout << "Third Distance :" << D3 << endl;

    return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
$./a.out
Enter the value of object :
70
10
First Distance : F : 11 I : 10
Second Distance :F : 5 I : 11
Third Distance :F : 70 I : 10
```

ROLE MODEL

BJARNE STROUSTRUP: COMPUTER WIZARD AND CREATOR OF C++ LANGUAGE

Best known for designing and implementing the C++ programming language which is widely in use today, this computer scientist is also managing director at Morgan Stanley, NY.

Born in a working class family in Aarhus, Denmark, on December 30, 1950, Stroustrup received early education in local schools. In 1969, he joined Aarhus University and graduated with master's degree in mathematics and computer science in 1975. Interested in microprogramming and machine architecture, he learned the fundamentals of object-oriented programming from its inventor, Norwegian computer scientist Kristen Nygaard. In 1979, he received PhD in computer science from the University of Cambridge, UK. He is an honorary professor at Aarhus University and an honorary fellow of Churchill College.

Career

In 1979, Stroustrup began his career with the Computer Science Research Center of Bell Labs in Murray Hill, New Jersey, USA, where he began work on C++ and programming techniques.

He headed AT&T Bell Labs' large-scale programming research department since its creation until late 2002. In 1993, he was made a Bell Lab's fellow and in 1996, an AT&T Fellow. He is also associated with AT&T Labs – Research and is a member of its Information and Systems Software Research Lab. He works as a visiting faculty in the Computer Science Department of Columbia University and is also a member of the US National Academy of Engineering (NAE), and fellow of the Institute of Electrical and Electronics Engineers (IEEE) and Association for Computing Machinery (ACM).

C++

In 1979, he began developing C++ (initially called C with Classes). C++ is a language for defining and using light-weight abstractions. It has significant strengths in areas where hardware must be handled effectively and there is considerable complexity to cope with. C++ was made available in 1985. In the same year, he also published a textbook titled The C++ Programming Language.

Stroustrup documented his principles guiding the design of C++ and evolution of the language in his 1994 book The Design and Evolution of C++ and two papers for ACM's History of Programming Languages conferences.

Personal life and legacy

Stroustrup lives in New York with his wife. Their daughter is a medical doctor and son a research professor in systems biology.

Apart from research, he is interested in light literature, general history, travelling, music, photography, and hiking and running. He was elected as the member of the NAE in 2004.

Awards

Stroustrup has received numerous honours including the William Procter Prize for Scientific Achievement in 2005, Computer Entrepreneur Award by IEEE in 2004, ACM's Grace Murray Hopper Award (1993), the 2018 Charles Stark Draper Prize from US National Academy of Engineering (NAE), the Grace Murray Hopper Award (1993), the 2017 IET Faraday Medal and Dr. Dobb's Excellence in Programming award (2008).

He received fellowships from the ACM and IEEE and was elected member of the NAE. In 2013, he was elected to the Electronic Design Hall of Fame.

Stroustrup has authored a large number of books such as A Tour of C++, Programming: Principles and Practice Using C++, The C++ Programming Language, and The Design and Evolution of C++.

He was awarded an honorary doctorate from the University Carlos III, Spain in 2019.

Interesting Facts

- In 2013, he received the Golden Abacus Award from Upsilon Pi Epsilon that is given to one who has gained professional fame and provided support and leadership in computing and information disciplines.
- In 1990, Bjarne Stroustrup was named one of America's 12 top young scientists by Fortune Magazine. Its recipients include Claude Shannon, mathematician, engineer and cryptographer.
- He applied programming techniques in areas like general systems programming, switching, simulation, graphics, user-interfaces, embedded systems and scientific computation.
- Stroustrup was made a Fellow of the Computer History Museum in 2015. His book titled The C++ Programming Language was translated into 19 languages and is one of the most widely read books of its kind.
- In 2018, he received the John Scott Legacy Medal and Premium from The Franklin Institute and the City Council of Philadelphia. It is the second oldest US award for scientific accomplishments.

SUMMARY

- C++ uses a convenient abstraction called *streams* to perform input and output operations in sequential media such as the screen, the keyboard or a file. A *stream* is an entity where a program can either insert or extract characters to/from.
- On most program environments, the standard output by default is the screen, and the C++ stream object defined to access it is cout.
- In most program environments, the standard input by default is the keyboard, and the C++ stream object defined to access it is cin.
- To get an entire line from cin, there exists a function, called getline, that takes the stream (cin) as first argument, and the string variable as second.
- The standard header <sstream> defines a type called stringstream that allows a string to be treated as a stream, and thus allowing extraction or insertion operations from/to strings in the same way as they are performed on cin and cout.
- A class hierarchy represents a set of hierarchically organized concepts. Base classes act typically as interfaces.
- File streams in C++ are basically the libraries that are used in the due course of programming. The programmers generally use the iostream standard library in the C++ programming as it provides the cin and cout methods that are used for reading from the input and writing to the output respectively.
- C++ has the capability of creating, accessing, and editing text files.
- C++ uses a file stream abstraction for handling text files. A file stream is data that is either going into a text file to be stored or going out of a text file to be loaded into a program. The input (>>) and the output (<<) operators are used to stream data either into a file or into a program.
- It's quite common that errors may occur during file operations. There may have different reasons for arising errors while working with files.
- The C++ programming language provides several built-in functions to handle errors during file operations.
- C++ is able to input and output the built-in data types using the stream extraction operator >> and the stream insertion operator <<. The stream insertion and stream extraction operators also can be overloaded to perform input and output for user-defined types like an object.

KNOWLEDGE CHECK

1. **Which header file is required to use file I/O operations?**
    a. <ifstream>
    b. <ostream>
    c. <fstream>
    d. <iostream>
2. **Which stream class is to only write on files?**
    a. ofstream
    b. ifstream
    c. iostream
    d. fstream
3. **Which of the following is used to create an output stream?**
    a. ofstream
    b. ifstream
    c. iostream
    d. fstream
4. **Which of the following is used to create a stream that performs both input and output operations?**
    a. ofstream
    b. ifstream
    c. iostream
    d. fstream
5. **Which of the following is not used as a file opening mode?**
    a. ios::trunc
    b. ios::binary
    c. ios::in
    d. ios::ate
6. **Which of the following statements are correct?**
    1) It is not possible to combine two or more file opening mode in open() method.
    2) It is possible to combine two or more file opening mode in open() method.
    3) ios::in and ios::out are input and output file opening mode respectively.
    a. 1, 3
    b. 2, 3

   c. 3 only
   d. 1, 2
7. **By default, all the files in C++ are opened in _________ mode.**
   a. Text
   b. Binary
   c. ISCII
   d. VTC
8. **What is the use of ios::trunc mode?**
   a. To open a file in input mode
   b. To open a file in output mode
   c. To truncate an existing file to half
   d. To truncate an existing file to zero
9. **Which of the following is the default mode of the opening using the ofstream class?**
   a. ios::in
   b. ios::out
   c. ios::app
   d. ios::trunc
10. **What is the return type open() method?**
   a. int
   b. char
   c. bool
   d. float

REVIEW QUESTIONS

1. What is meant by I O streams?
2. What are the types of I O streams?
3. What is class hierarchy example?
4. What is class hierarchy explain how inheritance helps in building class hierarchy?
5. What is stream in C++ with example?
6. Why fstream is used in C++?

7. How do text files work in C++?
8. How we read and write a binary file with example?
9. What are the error handling functions during I O operations?

Check Your Result

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

REFERENCES

1. A. Alexandrescu. “Traits on Steroids” (C++ Report, 12(6), June 2000).
2. A. Alexandrescu. “Traits: The else-if-then of Types” (C++ Report, 12(4), April 2000).
3. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. “FormatGuard: Automatic Protection From printf Format String Vulnerabilities” (Proceedings of the 2001 USENIX Security Symposium, August 2001, Washington, D.C.).
4. Fowler, Glenn S.; Korn, David G.; Vo, Kiem-Phong (2000). *Extended Formatting with Sfio*. Proc. Summer USENIX Conf.
5. Holzner, Steven (2001). C++ : Black Book. Scottsdale, Ariz.: Coriolis Group. p. 584. ISBN 1-57610-777-9. ...endl, which flushes the output buffer and sends a newline to the standard output stream.
6. S. Dewhurst. C++ Gotchas (Addison-Wesley, 2003).

CHAPTER 8

CONTROL FLOW

"Always think about how a piece of code should be used: good interfaces are the essence of good code. You can hide all kinds of clever and dirty code behind a good interface if you really need such code."

– Bjarne Stroustrup

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

1. Explain branching or conditional structure
2. Describe iterative or looping structure
3. Focus on sequential control flow structure

INTRODUCTION

When a program is run, the CPU begins execution at the top of main(), executes some number of statements (in sequential order by default), and then the program terminates at the end of main(). The specific sequence of statements that the CPU executes is called the program's execution path (or path, for short).

Consider the following program:

```
#include <iostream>

int main()
{
    std::cout << "Enter an integer: ";

    int x{};
    std::cin >> x;

    std::cout << "You entered " << x;

    return 0;
}
COPY
```

The execution path of this program includes lines 5, 7, 8, 10, and 12, in that order. This is an example of a straight-line program. Straight-line programs take the same path (execute the same statements in the same order) every time they are run.

However, often this is not what we desire. For example, if we ask the user for input, and the user enters something invalid, ideally we'd like to ask the user to make another choice. This is not possible in a straight-line program. In fact, the user may repeatedly enter invalid input, so the number of times we might need to ask them to make another selection isn't knowable until runtime.

Fortunately, C++ provides a number of different control flow statements (also called flow control statements), which are statements that allow the programmer to change the normal path of execution through the program. You've already seen an example of this with if statements that let us execute a statement only if a conditional expression is true.

When a control flow statement causes point of execution to change to a non-sequential statement, this is called branching.

8.1 BRANCHING OR CONDITIONAL STRUCTURE

In the normal state, execution of the program is gradual execution. When the Sequential Execution of the program is blocked by the selected Statements, it is called Branches Execution.

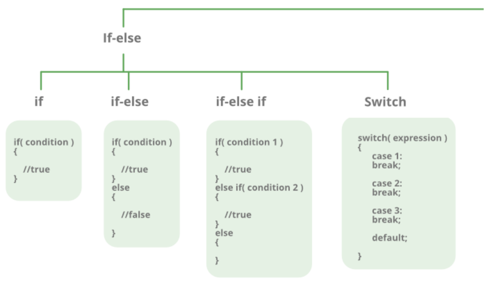
Control of the **program** requires a condition to block selected statements. That is, the control of the program is done by blocking the selected statements based on a condition, then the statement used in it is called Conditional Statement.

Keyword

Program, a set of instructions that describes how to perform a specific task to a computer.

C ++ consists of three Conditional Statements –

Branching or Conditional Structure



```
if( condition )
{

   //true
}
```

```
if( condition )
{

   //true
}
else
{

   //false

}
```

```
if( condition 1 )
{

   //true
}
else if( condition 2 )
{

   //true
}
else
{

}
```

```
switch( expression )
{
   case 1:
   break;

   case 2:
   break;

   case 3:
   break;

   default;

}
```

8.1.1 if statement

The if statement is a control statement that is used to test a particular condition. In this, the condition is executed only once when the condition is true.

If the condition is true in the statement then the statement is Execute.

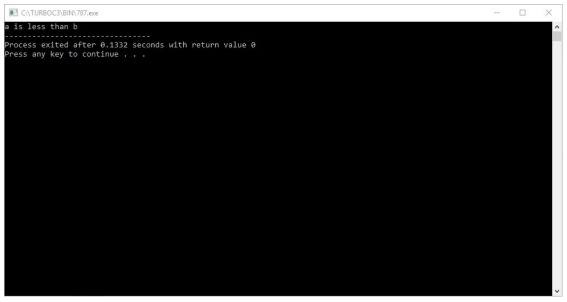
Syntax

```
if(condition)
{
statements
}
```

Example

```
#include <iostream>
using namespace std;

int main(){

int a=100, b=200;
  if (a < b)
  {
  cout<<"a is less than b";
  }
return 0;
}
note – this program run in c++ IDE compiler
```

Output



```
C:\TURBOC3\BIN\787.exe
a is less than b
--------------------------------
Process exited after 0.1332 seconds with return value 0
Press any key to continue . . .
```

Keyword

An **else statement** is an alternative statement that is executed if the result of a previous test condition evaluates to false.

8.1.2 if else statements

The if-else statement is used to test a particular condition. If the condition is true then the if statement is executed if the condition is false then the **else statement** is executed.

Syntax

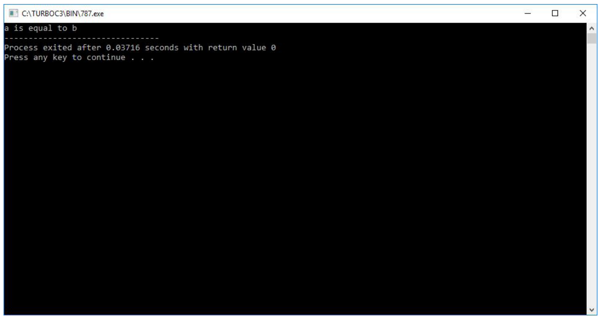
```
if(condition)
{
statement
}
else
{
statement
}
```

Example

```
#include <iostream>
using namespace std;

int main(){

int a=10, b=10;

if( a == b )
{

    cout<<"a is equal to b";
}
else
{
    cout<<"a is not equal to b";
}
return 0;
```

note – this program run in c++ IDE compiler

Output



```
a is equal to b
--------------------------------
Process exited after 0.03716 seconds with return value 0
Press any key to continue . . .
```

8.1.3 switch statements

This statement is also a selection statement that defines various paths for the execution of a program.

This serves as an alternative to the if-else statement.

Switch case statement has expression and some cases related to it. The case which matches that expression or declares variable is printed in the output.

If no case matches the expression then it will print the default statement in the output. You have to break after every statement, that means it will print only the statement before it.

If you do not break, then it will print both the first and the second statement. Do not break after the default case.

Syntax

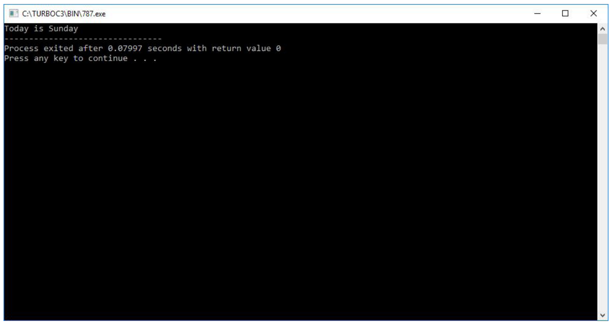
```
switch(variable)
{
case constant 1;
statements(s);
break:
case constant 2;
statement(s);
break;
case constant 3;
statement(s);
break;
————–
default
statement(s);
}
```

Example

```
#include <iostream>
using namespace std;

int main(){

char Day='A';

switch(Day){

        case 'A' :
        cout<<"Today is Sunday";
        break;

        case 'B' :
        cout<<"Today is Monday";
        break;

        case 'C' :
        cout<<"Today is Tuesday";
        break;

        case 'D' :

        case 'E' :
        cout<<"Today is Wednesday";
        break;
```

```
          case 'F' :
          cout<<"Today is Thurday";
          break;

          case 'G' :
          cout<<"Today is Friday";
          break;

          case 'H' :
          cout<<"Today is Saturday";
          break;

          default :
         cout<<"Day is Not Found";
}
return 0;
}
```

Output



C:\TURBOC3\BIN\787.exe

```
Today is Sunday
--------------------------------
Process exited after 0.07997 seconds with return value 0
Press any key to continue . . .
```

8.1.4 else if Statement

The condition of if is true in else_if statement then statement of if is executed.
The condition of if is false, then it goes to the next condition and checks.
condition is true, then it executes its statement.
no condition is true, then it executes the statement of else.

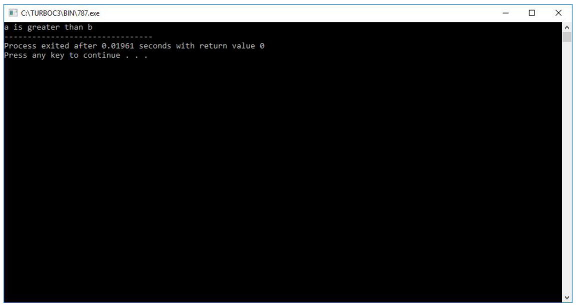
Syntax for else if Statement

```
if(condition){
 statement(s);
}else if(condition){
statement(s);
}else{
statement(s);
}
```

Example

```
#include<iostream>
using namespace std;

int main(){

int a=100, b=20;

if( a < b ){
    cout<<"a is less than b";
}else if( a > b ){
    cout<<"a is greater than b";
}else{
    cout<<"a is equal to b";
}
return 0;
}
```

Output


C:\TURBOC3\BIN\787.exe

```
a is greater than b
--------------------------------
Process exited after 0.01961 seconds with return value 0
Press any key to continue . . .
```

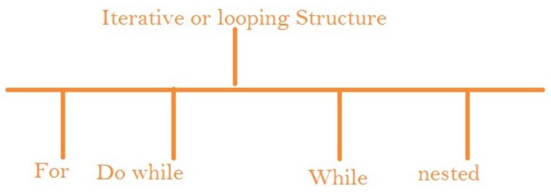
Keyword

Iteration statements cause statements (or compound statements) to be executed zero or more times, subject to some loop-termination criteria. When these statements are compound statements, they are executed in order, except when either the break statement or the continue statement is encountered.

8.2 ITERATIVE OR LOOPING STRUCTURE

The statements that cause a set of statements to be executed repeatedly either for a specific number of times or until some condition is satisfied are known as **iteration statements**. That is, as long as the condition evaluates to True, the set of statement(s) is executed. The various iteration statements used in C++ *are for loop, while loop* and *do while loop.*

Through loops, we can execute anyone's statement or many abstract statements more than once until the condition is achieved.



8.2.1 The for Loop

The for loop is one of the most widely used loops in C++. The for loop is a deterministic loop in nature, that is, the number of times the body of the loop is executed is known in advance.

The syntax of the for loop is

```
for(initialize; condition; update) {
  //body of the for loop
}
```

Note that initialize, condition and update are optional expressions and are always specified in parentheses. All the three expressions are separated by semicolons. The semicolons are mandatory and hence cannot be excluded even if all the three expressions are omitted.

To understand the concept of the for statement, consider this example.

Example: A program to display a countdown using for loop

```
#include<iostream>
using namespace std;
int main() {
  int n;
  for(n=l; n<=10; n++)
    cout<<n<<" "; // body of the loop
    cout<<"\n This is an example of for loop!! !";
    //next statement in sequence
return 0;
}
```

The output of this program is

```
1 2 3 4 5 6 7 8 9 10

This is an example of for loop!! !
```

for loop using comma operator: for loop allows multiple variables to control the loop using comma operator. That is, two or more variables can be used in the initialize and the update parts of the loop. For example, consider this statement.

```
for (i=1,j=50 ;i<10;i++, j- -)
```

This statement initializes two variables, namely i and j and updates them. Note that for loop can have only one condition.

- Scenario 1: Copying (Cloning)

```
// Rust
let v = vec!["a".to_string(), "b".to_string(), "c".to_string()];
for e.clone() in &v {
    println!("{}", e);
}
// C++
#include <iostream>
#include <vector>
std::vector<std::string> a{"a", "b", "c"};
for (auto s : a) // s has type `std::string`
{
    std::cout << s << std::endl;
}
```

Did You Know?

A procedure for solving a problem in terms of the actions to be executed and the order in which these actions should be executed is called an algorithm.

Note how Rust makes it crystal clear that copying during iteration and using String for instead of &str for static strings are anti-patterns and how C++ makes it easy to write such inefficient code.

- Scenario 2: As Reference (Borrowing)

```
// Rust
let v = vec![1, 2, 3, 4, 5];
for e in &v {
    println!("{}", *e);
}
// C++
#include <iostream>
#include <vector>
std::vector<int> a{1, 2, 3, 4, 5};
for (auto& num : a)
{
    printf("%d ", num); // no asterisk!
}
```

- Scenario 3: Mutation

```
// Rust
```

```
let mut v = vec![1, 2, 3, 4, 5];
for e in &mut v {
    *e = *e + 1;
}
assert_eq!(v, vec![2, 3, 4, 5, 6]);
// C++
#include <vector>
#include <cassert>
std::vector<int> a{1, 2, 3, 4, 5};
for (auto &num : a)
{
    num++;
}
assert(a == (std::vector<int>{2, 3, 4, 5, 6}));
```

Note the parentheses surrounding the second argument of assert, without which we would get an error. This is because assert is a so called #define macro, which simply parses its arguments as comma-separated identifiers and does text replacement. Since std::vector<int>{2, 3, 4, 5, 6} contains commas, it has to be escaped with parentheses.[1] This macro is actually defined in assert.h which is part of C's std, and its more or less copied verbatim into C++'s cassert.

8.2.2 The while Loop

The while loop is used to perform looping operations in situations where the number of iterations is not known in advance. That is, unlike the for loop, the while loop is non deterministic in nature.

The syntax of the while loop is

```
while(condition) {
  // body of while loop
}
```

These points should be noted about the while loop.

- Unlike for loops where explicit initialize and update expressions are specified, while loops do not specify any explicit initialize and update expressions. This implies that the control variable must be declared and initialized before the while loop and needs to be updated within the body of the while loop .

- The while loop executes as long as condition evaluates to True. If condition evaluates to False in the first iteration, then the body of while loop never executes.
- while loop can have more than one expression in its condition. However, such multiple expressions must be separated by commas and are executed in the order of their appearance.

To understand the concept of the while loop, consider this example.

Example : A program to determine the sum of first n consecutive positive integers

```
#include<iostream>
using namespace std;
int main() {
  int n,i,sum; // i is the control variable
  cout<<" Enter the number of consecutive positive"<<
  "\n integers(starting from 1): ";
  cin>>n;
  sum=0;
  i=l; // initialize expression
  while (i<=n) {
   sum+=i;
   ++i; //update expression
}
cout<<"\nThe sum is "<<sum;
return 0;
}
```

The output of the program is

```
Enter the number of consecutive positive integers(starting from 1): 9
The sum is : 45
```

8.2.3 The do-while loop

In a while loop, the condition is evaluated at the beginning of the loop and if the condition evaluates to False, the body of the loop is not executed even once. However, if the body of the loop is to be executed at least once, no matter whether the initial state of the condition is True or False, the do-while loop is used. This loop places the condition to be evaluated at the end of the loop.

The syntax of the do-while loops is given here.

```
do {
 //body of do while loop
}while(condition) ;
```

To understand the concept of do-while loop, consider this example.

Example : A program to calculate the sum of an Arithmetic Progression (AP)

```
#include<iostream>
using namespace std;
int main () {
  int a,d,n,sum,term=0; /*a is the first term , d is
  the common difference, n is the number of terms to be summed */
  cout<<"Enter the first term, common difference,"
  <<"and the number of terms to be summed"
  <<"respectively:\n";
  cin>>a>>d>>n;
  sum=0;
  int i=1;
  cout<<"\nThe terms are ";
  do //do-while loop {
   term= a+ (i-1)*d;
   sum+=term; //Adding each term to 'sum'
   cout<<term<<" ";
   ++i;
}
while (i<=n) ;
cout<<"\nThe sum of A.P. is "<<sum;
return 0;
}
```

The output of the program is

```
Enter the first term, common difference, and the number of terms to be summed respectively:
3
6
5
The terms are 3 9 15 21 27
The sum of A.P. is 75
```

Remember

Control instructions make your code powerful and versatile.

Note that, all the three loops (for, while and do-while) can be nested within the body of another loop

8.3 SEQUENTIAL CONTROL FLOW STRUCTURE

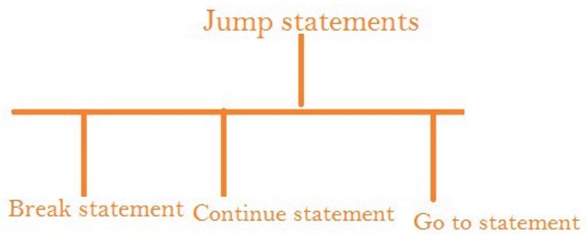
The statements written in the program are implemented one after the other.

The statements are executed in the order in which the compiler is received.

This execution of the program is called serial execution.

8.3.1 Jump statements

Jump statements are used to interrupt the normal flow of program.



8.3.2 Break statement

This statement is used to end a sequence of statements in a switch statement and to immediately exit a loop.

The execution of the loops and switch cases of the Break Statement Program stops at any condition.

syntax:- break;

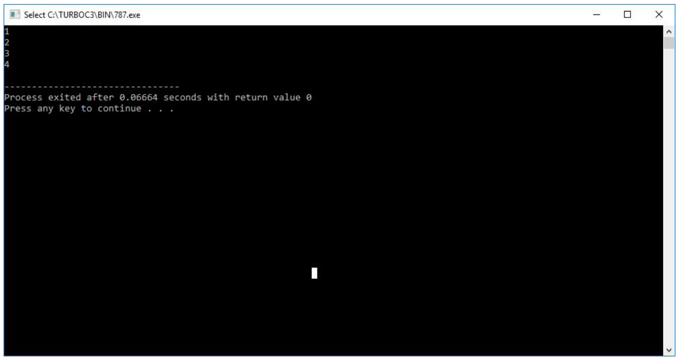
Example

```
#include <iostream>
using namespace std;
int main() {
      for (int i = 1; i <= 10; i++)
          {
              if (i == 5)
              {
                  break;
              }
        cout<<i<<"\n";
          }
}
```

Remember

The break statement, when executed in one of the repetition structures (for, while and do/while), causes immediate exit from the structure.

Output



```
1
2
3
4

--------------------------------
Process exited after 0.06664 seconds with return value 0
Press any key to continue . . .
```

8.3.3 Continue statement

The continue statement is used when we want to run the loop continues with the next iteration and skip other statements in the loop for the current iteration.

Depending on the condition of the loops of the Continue Statement program, skip the middle statements, and execute the subsequent statements.

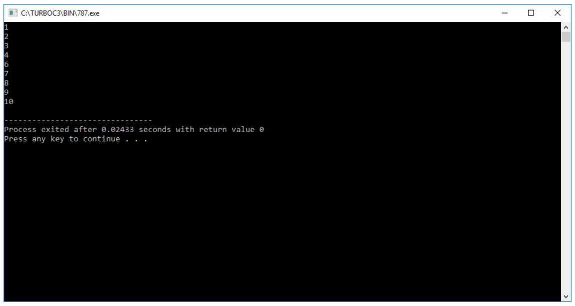
syntax:- continue;

Example

```
#include <iostream>

using namespace std;

int main()

{

    for(int i=1;i<=10;i++){

          if(i==5){

              continue;

          }

          cout<<i<<"\n";

       }

}
```

note – this program run in c++ IDE compiler

Output



```
1
2
3
4
6
7
8
9
10

--------------------------------
Process exited after 0.02433 seconds with return value 0
Press any key to continue . . .
```

8.3.4 Go to statement

Go to is the statement of C ++ Programming. Labels are used in this.

There are two types of Go to Statements.

Forward

Backward

When a goto statement executes its next statement except for some statement, it is called Forward goto statement and goes to its previous label to execute any previous or executed statement again, it is called Backward goto statement.

Remember

The continue statement, when executed in one of the repetition structures (for, while and do/while), skips any remaining statements in the body of the structure and proceeds with the next iteration of the loop.

Syntax for Forward and Backward goto Statement

Syntax for Forward

goto label ;

statement ;

————

label ;

Syntax for Backward

label ;

statement ;

————

goto label ;

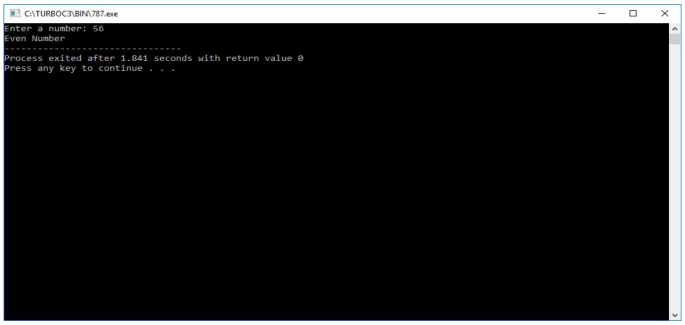
Example

```
#include <iostream>
using namespace std;
int main()
{
   int num; cout<<"Enter a number: ";
    cin>>num;
   if (num % 2==0){
      goto print;
   }
   else {
      cout<<"Odd Number";
   }

   print:
   cout<<"Even Number";
   return 0;
}
```

Remember

Any nonzero value implicitly converts to true; 0 (zero) implicitly converts to false.

Output



```
Enter a number: 56
Even Number
--------------------------------
Process exited after 1.841 seconds with return value 0
Press any key to continue . . .
```

ROLE MODEL

JOSEPH MARIE JACQUARD

Innovator of the loom that bears his name, Joseph Marie Jacquard (1752-1834) developed the first loom to weave designs into cloth. It was also recognized as the first machine to employ the punch-card technology, that would eventually program the computer of the mid-twentieth century.

In the 1700s, the European textile industry and specifically weaving, had not changed for hundreds of years. Using a loom, a weaver created woven fabrics by interlacing two sets of threads—taut lengthwise or "warp" threads that were crosswise, and "weft" or "filling" threads, at right angles. To create wide finished textiles, such as those used for window coverings, narrow lengths of fabric had to be woven by hand. Warp threads were then tautly stretched across the loom's frame, and raised and lowered by the loom's harness, to allow the weft threads to be woven between them. These intricately textured patterns, as well as multi-colored designs were time-consuming. Even so, with its generations of skilled weavers, by the mid-1800s, France was known around the world for the quality of its woven silks.

As ever-larger mechanized looms replaced skilled hand weavers in the 1790s, an explosion of woven goods appeared in European and American trade markets. These goods were inexpensive due to being mass-produced. However, these new, mechanized looms could not compete with the skilled manual labor required to create fabrics containing anything other than a plain or simple, woven pattern, such as a check or stripe.

It would be the invention of a Frenchman named Joseph Marie Jacquard that would spread mass production to these more complicated, and costly, textile designs, allowing even intricate patterns to be automatic ally woven into the cloth at much the same rate as a plain length of fabric could be generated.

Son Of A Silk Weaver

Born July 7, 1752, in the southern French city of Lyon, Jacquard spent much of his life in the silk textile industry. Like his parents had before him, young Joseph went to work at a silk mill in Lyon. Along with many young boys of his generation and economic status, he grew up working 10-hour days within the factory. His first task as a young worker was to serve as a draw-boy.

Sitting on a perch above the heavy, massive loom and working quickly in advance of each passage of the flying shuttle carrying the weft thread, he would lift and re-position warp threads of various colors in different spots to create the pattern desired by the Master weaver who operated the loom. This tedious and sometimes dangerous task was given to children because their smaller fingers were more capable of setting the fine silk, wool, or cotton threads used.

The Industrial Revolution heralded what would be a long, gradual shift from a farming economy, to an industrial, trade-based economy. As fewer peasants made their living off the land, they migrated to the cities, where factories sought workers in response to foreign demands for their trade goods. Throughout France, the textile industry flourished.

Poverty Leads To Revolution

Unfortunately, this new economic growth and the growth of a new entrepreneurial class came at some expense. The citizens of Lyon, as well as other industrial cities, were overworked, yet still poor and lacking food. The "curse" of the Industrial Revolution was that the upper middle-class factory owners profited from the rise in foreign trade, while the lower classes suffered crowded living conditions and little pay.

By the time Jacquard had entered adulthood, France was entering one of the most tumultuous periods of its history: the French Revolution. And in Lyons, one of the country's most densely inhabited cities, this unrest— particularly that caused by the shift in political power from the wealthy nobility into the hands of the masses—was felt by all. Changes in the status quo were happening on all levels, including political, social, economic, and technological areas.

As early as 1775, French Controller-General Anne-Robert Turgot had encouraged free trade by inhibiting the restrictive guild system and subsidizing innovations in those industries he believed would one day make France an economic rival with her nemesis, Great Britain. Following the execution of Turgot's employer, King Louis XVI, and the rise of a revolutionary government, innovations among the French citizenry continued to be encouraged and the inventive spirit was rewarded with government grants. This trend would continue following the Revolution, as Emperor Napoleon Bonaparte himself encouraged technological advances in his every-growing republic.

This encouragement by the government drew the interest of young men such as Jacquard, who had grown up and advanced to the position of mill mechanic in Lyon. Reflecting on his childhood job, Jacquard set about to find an alternative to the position of draw-boy in the silk industry.

A concept developed by fellow Frenchman Jacques de Vaucanson in 1745, that utilized a perforated roll of paper to control the weaving process, served as Jacquard's starting point. Given one of Vaucanson's looms to restore, Jacquard set to work on correcting Vaucanson's unworkable design. Absorbed by his project for several years, Jacquard created an operative prototype of his loom by 1790.

By 1793, the Revolution was in full swing, forcing Jacquard to abandon his project; instead he joined the republican lower classes in mounting their historic attack on the French nobility. After fighting alongside his fellow citizens in defense of the new French republic, Jacquard resumed his work in 1801, shortly after Napoleon's rise to power. His improved draw-loom, displayed that same year at an industrial exhibition in the Louvre in Paris, earned Jacquard a bronze medal.

Three years later, in the fall of 1803, the inventor was again summoned to Paris, this time to demonstrate a second version of his original loom design. This version had attached to the top of its frame the "Jacquard mechanism" or "Jacquard attachment," which was a device connecting the wooden loom to an interchangeable continuous roll of connected punch cards. This remarkably innovative method of "programming" a machine allowed the Jacquard loom to produce tapestries, brocades, damasks, and other intricately woven silk fabrics far more quickly than had the manual technology of the past.

The Technology Of Jacquard Weaving

The innovation underlying Jacquard's loom was the use of encoded punch cards to control the action of the weaving process, allowing any desired pattern to be reproduced automatically. The required design is encoded onto a series of connected pasteboard cards as a group of punched holes, each card containing a single line of holes representing a single row of weave. Each series of rectangular cards, when connected, creates a grid of rows and columns.

Jacquard's mechanism allowed each warp thread to operate independently, much like a player piano, where each note is sounded by a hole on a music roll as it passes over a certain opening. In the Jacquard mechanism, a specific combination of holes punched in a row through an individual card allowed selected sprung rods or needles to pass through the card and pick up certain threads. The connected cards create a continuous loop allowing for repeated patterns; when all the cards have been used, the sequence begins again.

Combining any number of connected cards in a loop, Jacquard's loom was able to weave patterns of great complexity, and these became popular for tablecloths and bed coverings. In addition to textile designs featuring smallscale, repeated patterns, Jacquard became known for intricate representational coverlets featuring a single large design, woven in a variety of colors.

One remarkable example of his craft that still exists is a black-and-white silk portrait of Jacquard himself, which was woven using a strip of ten thousand cards. Also important is the course his technology would take. Jacquard's open hole/closed hole system was the first use of the binary system that would be translated into a basic computer over a century later. In addition, computer operators would refer to his concept of sequencing individual cards in a specific order to create a specific pattern, as sequencing commands to create a "program."

Innovation Gave Rise To Computer

Jacquard's invention was immediately recognized as something that would revolutionize the French textile industry. Ironically, the impoverished factory mechanic, who had also risked his life in defense of his country, would earn no money directly from his invention. Instead, in an agreement with the city of Lyon, the patent for his Jacquard mechanism reverted to the city, which declared his invention public property in 1806. Fortunately, Jacquard was awarded a state pension by Emperor Napoleon that allowed him to profit from his innovation; in addition he received royalties on each loom sold and put into operation.

Perhaps more significant that its revolution of the textile industry, Jacquard's innovative use of the punched card mechanism greatly influenced other inventors. English inventor Charles Babbage used Jacquard's technology in his development of the analytical engine, a simple form of a calculator. American statistician Herman Hollerith adopted punchcards as a means of entering data into his census collator. His collator, developed in 1890, was used through the 1960s to tabulate results of the United States census.

Repercussions Of Progress

Like many labor-saving developments that occurred during the Industrial Revolution, Jacquard's technology was not immediately embraced by silk weavers and others in weaving trades. They saw it as a threat to their jobs and protested its use. As early as 1801, riots broke out in Lyon over changes to the traditional loom. In 1804, after Jacquard's revised loom was introduced, the violence escalated. In addition to trying to destroy any Jacquard looms that were in use in Lyon, attempts were made on Jacquard's life.

However, the advantages of his looms eventually won out over the opposition. In 1800, only 3,500 working looms were in use in Lyon's silk industry. Within a decade, the number of working looms in the city reached 11,000. One textile mill owner even had thousands of workers on his payroll.

By 1810, France had become competitive with its longstanding rival, Great Britain, in the textile industry. In 1819, Jacquard was awarded the Legion of Honor Cross, as well as a gold medal, for his role in his nation's economic success. During the 1820s, his name became known worldwide as use of the Jacquard loom spread to England.

Jacquard died in Oullins, France on August 7, 1834. Over 160 years later, the technology that bears his name is still in use around the world.

CASE STUDY

THINKINGABOUTOBJECTS:IDENTIFYINGTHECLASSESIN A PROBLEM

Now we begin our optional, object-oriented design/implementation case study. This case study will provide you with a substantial, carefully paced, complete design and implementation experience. We present this case study in a fully solved format. This is not an exercise; rather it is an endto-end learning experience that concludes with a detailed walkthrough of the C++ code. We have provided this case study so you can become accustomed to the kinds of substantial problems that are attacked in industry. We hope you enjoy this experience.

Problem Statement

A company intends to build a two-floor office building and equip it with an elevator. The company wants you to develop an object-oriented software simulator in C++ that models the operation of the elevator to determine whether or not it will meet their needs.

Your simulator should include a clock that begins with its time, in seconds, set to zero. The clock ticks (increments the time by one) every second; it does not keep track of hours and minutes. Your simulator should also include a scheduler that begins the day by randomly scheduling two times: the time when a person will step onto floor 1 and press the button on the floor to summon the elevator, and the time when a person will step onto floor 2 and press the button on the floor to summon the elevator. Each of these times is a random integer in the range of 5 to 20, inclusive (i.e., 5, 6, 7, ..., 20). When the clock time equals the earlier of these two times, the scheduler creates a person, who then walks onto the appropriate floor and presses the floor button. [Note: It is possible that these two randomly scheduled times will be identical, in which case people will step onto both floors and press both floor buttons at the same time.] The floor button illuminates, indicating that it has been pressed. [Note: The illumination of the floor button occurs automatically when the button is pressed and needs no programming; the light built into the button turns off automatically when the button is reset.] The elevator starts the day waiting with its door closed on floor 1. To conserve energy, the elevator moves only when necessary. The elevator alternates directions between moving up and moving down.

For simplicity, the elevator and each of the floors have a capacity of one person. The scheduler first verifies that a floor is unoccupied before creating a person to walk onto that floor. If the floor is occupied, the scheduler delays creating the person by one second (thus giving the elevator an opportunity to pick up the person and clear the floor). After a person walks onto a floor, the scheduler creates the next random time (between 5 and 20 seconds into the future) for a person to walk onto that floor and press the floor button.

When the elevator arrives at a floor, it resets the elevator button and sounds the elevator bell (which is inside the elevator). The elevator then signals its arrival to the floor. The floor, in response, resets the floor button and turns on the floor's elevator arrival light. The elevator then opens its door. [Note: The door on the floor opens automatically with the elevator door and needs no programming.] The elevator's passenger, if there is one, exits the elevator, and a person, if there is one waiting on that floor, enters the elevator. Although each floor has a capacity of one person, assume there is enough room on each floor for a person to wait on that floor while the elevator's passenger, if there is one, exits.

A person entering the elevator presses the elevator button, which illuminates (automatically, without programming) when pressed and turns off when the elevator arrives on the floor and resets the elevator button. [Note: Because there are only two floors, only one elevator button is necessary; this button simply tells the elevator to move to the other floor.] Next, the elevator closes its door and begins moving to the other floor. When the elevator arrives at a floor, if a person does not enter the elevator and the floor button on the other floor has not been pressed, the elevator closes its door and remains on that floor until a button on a floor is pressed.

For simplicity, assume that all the activities that happen once the elevator reaches a floor, and until the elevator closes its door, take zero time. [Note: Although these activities take zero time, they still occur sequentially, e.g., the elevator door must open before the passenger exits the elevator.] The elevator takes five seconds to move from either floor to the other. Once per second, the simulator provides the time to the scheduler and to the elevator. The scheduler and elevator use the time to determine what actions each needs to take at that particular time, e.g., the scheduler may determine that it is time to create a person; and the elevator, if moving, may determine that it is time to arrive at its destination floor.

The simulator should display messages on the screen describing the activities that occur in the system. These include a person pressing a floor button, the elevator arriving on a floor, the clock ticking, a person entering the elevator, etc. The output should resemble the following:

```
Enter run time: 30
(scheduler schedules next person for floor 1 at time 5)
(scheduler schedules next person for floor 2 at time 17)
*** ELEVATOR SIMULATION BEGINS ***
TIME: 1
elevator at rest on floor 1
TIME: 2
```

```
elevator at rest on floor 1
TIME: 3
elevator at rest on floor 1
TIME: 4
elevator at rest on floor 1
TIME: 5
scheduler creates person 1
person 1 steps onto floor 1
person 1 presses floor button on floor 1
floor 1 button summons elevator
(scheduler schedules next person for floor 1 at time 20)
elevator resets its button
elevator rings its bell
floor 1 resets its button
floor 1 turns on its light
elevator opens its door on floor 1
person 1 enters elevator from floor 1
person 1 presses elevator button
elevator button tells elevator to prepare to leave
floor 1 turns off its light
elevator closes its door on floor 1
elevator begins moving up to floor 2 (arrives at time 10)
TIME: 6
elevator moving up
TIME: 7
elevator moving up
TIME: 8
elevator moving up
TIME: 9
elevator moving up
TIME: 10
elevator arrives on floor 2
elevator resets its button
elevator rings its bell
floor 2 resets its button
floor 2 turns on its light
elevator opens its door on floor 2
person 1 exits elevator on floor 2
floor 2 turns off its light
elevator closes its door on floor 2
elevator at rest on floor 2
TIME: 11
elevator at rest on floor 2
TIME: 12
elevator at rest on floor 2
TIME: 13
elevator at rest on floor 2
TIME: 14
elevator at rest on floor 2
TIME: 15
elevator at rest on floor 2
TIME: 16
elevator at rest on floor 2
TIME: 17
scheduler creates person 2
person 2 steps onto floor 2
person 2 presses floor button on floor 2
floor 2 button summons elevator
(scheduler schedules next person for floor 2 at time 34)
elevator resets its button
elevator rings its bell
floor 2 resets its button
floor 2 turns on its light
elevator opens its door on floor 2
person 2 enters elevator from floor 2
person 2 presses elevator button
```

```
elevator button tells elevator to prepare to leave
floor 2 turns off its light
elevator closes its door on floor 2
elevator begins moving down to floor 1 (arrives at time 22)
TIME: 18
elevator moving down
TIME: 19
elevator moving down
TIME: 20
scheduler creates person 3
person 3 steps onto floor 1
person 3 presses floor button on floor 1
floor 1 button summons elevator
(scheduler schedules next person for floor 1 at time 26)
elevator moving down
TIME: 21
elevator moving down
TIME: 22
elevator arrives on floor 1
elevator resets its button
elevator rings its bell
floor 1 resets its button
floor 1 turns on its light
elevator opens its door on floor 1
person 2 exits elevator on floor 1
person 3 enters elevator from floor 1
person 3 presses elevator button
elevator button tells elevator to prepare to leave
floor 1 turns off its light
elevator closes its door on floor 1
elevator begins moving up to floor 2 (arrives at time 27)
TIME: 23
elevator moving up
```

```
TIME: 24
elevator moving up
TIME: 25
elevator moving up
TIME: 26
scheduler creates person 4
person 4 steps onto floor 1
person 4 presses floor button on floor 1
floor 1 button summons elevator
(scheduler schedules next person for floor 1 at time 35)
elevator moving up
TIME: 27
elevator arrives on floor 2
elevator resets its button
elevator rings its bell
floor 2 resets its button
floor 2 turns on its light
elevator opens its door on floor 2
person 3 exits elevator on floor 2
floor 2 turns off its light
elevator closes its door on floor 2
elevator begins moving down to floor 1 (arrives at time 32)
TIME: 28
elevator moving down
TIME: 29
elevator moving down
TIME: 30
elevator moving down
*** ELEVATOR SIMULATION ENDS ***
```

Our goal is to implement a working software simulator that models the operation of the elevator for the number of seconds entered by simulator user.

Analyzing and Designing the System

In this and the next several "Thinking About Objects" sections, we perform the steps of an object-oriented design process for the elevator system. The UML is designed for use with any OOAD process—many such processes exist. One popular method is the Rational Unified Process™ developed by Rational Software Corporation. For this case study, we present our own simplified design process for your first OOD/UML experience.

Before we begin, we must examine the nature of simulations. A simulation consists of two portions. One contains all the elements that belong to the world we want to simulate. These elements include the elevator, the floors, the buttons, the lights, etc. Let us call this the world portion. The other portion contains all the elements needed to simulate this world. These elements include the clock and the scheduler. We call this the controller portion. We will keep these two portions in mind as we design our system.

Use Case Diagrams

When developers begin a project, they rarely start with a detailed problem statement, such as the one we have provided at the beginning of this section. This document and others are usually the result of the object-oriented analysis (OOA) phase. In this phase you interview the people who want you to build the system and the people who will eventually use the system. You use the information gained in these interviews to compile a list of system requirements. These requirements guide you and your fellow developers as you design the system. In our case study, the problem statement contains the system requirements for the elevator system. The output of the analysis phase is intended to specify clearly what the system is supposed to do. The output of the design phase is intended to clearly specify how the system should be constructed to do what is needed.

The UML provides the use case diagram to facilitate the process of requirements gathering. The use case diagram models the interactions between the system's external clients and the use cases of the system. Each use case represents a different capability that the system provides the client. For example, an automated teller machine has several use cases, including "Deposit," "Withdraw" and "Transfer Funds."

Figure 1 shows the use case diagram for the elevator system. The stick figure represents an actor. Actors are any external entities such as people, robots, other systems, etc., that use the system. The only actors in our system are the people who want to ride the elevator. We therefore model one actor called "Person." The actor's "name" appears underneath the stick figure.

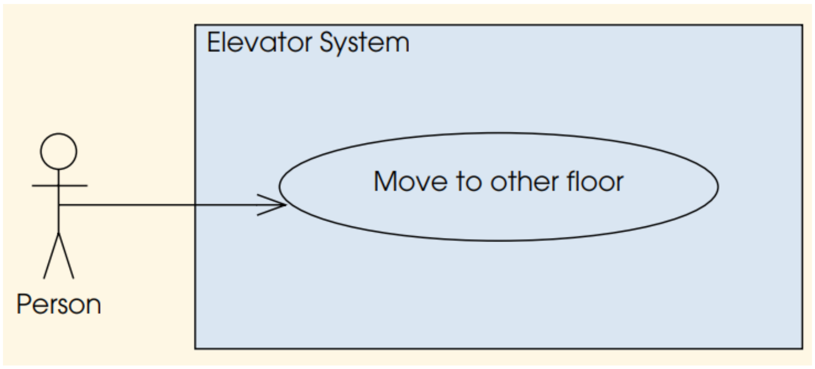


Figure 1: Use case diagram for elevator system.

The system box (i.e., the enclosing rectangle in the figure) contains the use cases for the system. Notice that the box is labeled "Elevator System." This title shows that this use case model focuses on the behaviors of the system we want to simulate (i.e., elevator transporting people), as opposed to the behaviors of the simulation (i.e., creating people and scheduling arrivals). The UML models each use case as an oval. In our simple system, actors use the elevator for only one purpose: to move to another floor. The system provides only one capability to its users; therefore, "Move to other floor" is the only use case in our elevator system.

As you build your system, you rely on the use case diagram to ensure that all the clients' needs are met. Our case study contains only one use case. In larger systems, use case diagrams are indispensable tools that help systems designers remain focused on satisfying the users' needs. The goal of the use case diagram is to show the kinds of interactions users have with a system without providing the details of those interactions.

Identifying the Classes in a System

The next step of our OOD process is to identify the classes in our problem. We will eventually describe these classes in a formal way and implement them in C++. First we review the problem statement and locate all the nouns; with high likelihood, these represent most of the classes (or instances of classes) necessary to implement the elevator simulator. Figure 2 is a list of these nouns.

List of nouns in the problem statement
company
building
elevator
simulator
clock
time
scheduler
person
floor 1
floor button
floor 2
elevator door
energy
capacity
elevator button
elevator bell
floor's elevator arrival light
person waiting on a floor
elevator's passenger

Figure 2: List of nouns in problem statement

We choose only the nouns that perform important duties in our system. For this reason we omit the following:

- company
- simulator
- time
- energy
- capacity

We do not need to model “company” as a class, because the company is not part of the simulation; the company simply wants us to model the elevator. The “simulator” is our entire C++ program, not an individual class. The “time” is a property of the clock, not an entity itself. We do not model “energy” in our simulation (although electric, gas or oil companies might certainly be interested in doing so in their simulation programs) and, finally, “capacity” is a property of the elevator and of the floor—not a separate entity itself.

We determine the classes for our system by filtering the remaining nouns into categories. Each remaining noun from Fig. 2 refers to one or more of the following categories:

- building
- elevator
- clock
- scheduler
- person (person waiting on a floor, elevator’s passenger)
- floor (floor 1, floor 2)
- floor button
- elevator button
- bell
- light
- door

These categories are likely to be the classes we will need to implement for our system. Notice that we create one category for the buttons on the floors and one category for the button on the elevator. The two types of buttons perform different duties in our simulation—the buttons on the floors summon the elevator, and the button in the elevator tells the elevator to begin moving to the other floor.

We can now model the classes in our system based on the categories we derived. By convention, we will capitalize class names. If the name of a class contains more than one word, we run the words together and capitalize each word (e.g., MultipleWordName). Using this convention, we create classes Elevator, Clock, Scheduler, Person, Floor, Door, Building, FloorButton, ElevatorButton, Bell and Light. We construct our system using all of these classes as building blocks. Before we begin building the system, however, we must gain a better understanding of how the classes relate to one another.

Class Diagrams

The UML enables us to model the classes in the elevator system and their relationships via the class diagram. Figure 3 shows how to represent a class using the UML. Here, we model class Elevator. In a class diagram, each class is modeled as a rectangle. This rectangle can then be divided into three parts. The top part contains the name of the class. The middle part contains the class's attributes. The bottom contains the class's operations. Classes relate to one another via associations. Figure 4 shows how our classes Building, Elevator and Floor relate to one another. Notice that the rectangles in this diagram are not subdivided into three sections. The UML allows the truncation of class symbols in this manner in order to create more readable diagrams.

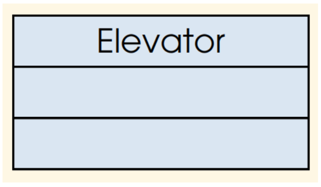


Figure 3: Representing a class in the UML.

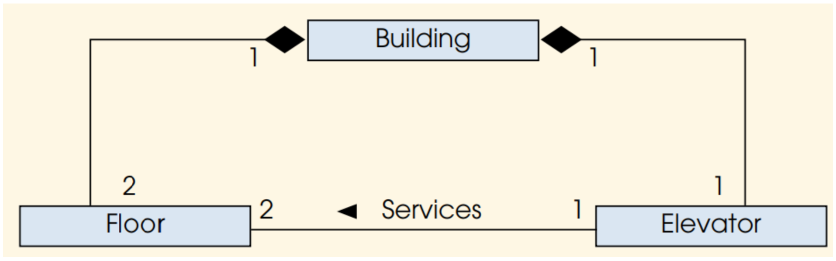


Figure 4: Associations between classes in a class diagram.

In this class diagram, a solid line that connects classes represents an association. An association is a relationship between classes. The numbers near the lines express multiplicity values. Multiplicity values indicate how many objects of a class participate in the association. From the diagram, we see that two objects of class Floor participate in the association with one object of class Building. Therefore, class Building has a one-totwo relationship with class Floor; we can also say that class Floor has a two-to-one relationship with class Building. From the diagram, you can see that class Building has a one-to-one relationship with class Elevator and vice versa. Using the UML, we can model many types of multiplicity. Figure 5 shows the multiplicity types and how to represent them.

An association can be named. For example, the word "Services" above the line connecting classes Floor and Elevator indicates the name of that association—the arrow shows the direction of the association. This part of the diagram reads: "one object of class Elevator services two objects of class Floor."

The solid diamond attached to the association lines of class Building indicates that class Building has a composition relationship with classes Floor and Elevator. Composition implies a whole/part relationship. The class that has the composition symbol (the solid diamond) on its end of the association line is the whole (in this case, Building), and the class on the other end of the association line is the part (i.e., Floor and Elevator).

Symbol	Meaning
0	None.
1	One.
m	An integer value.
0..1	Zero or one.
m..n	At least *m*, but not more than *n*.
*	Any non-negative integer.
0..*	Zero or more
1..*	One or more

Figure 5: Multiplicity table.

Figure 6 shows the full class diagram for the elevator system. All the classes we created are modeled, as well as the associations between these classes.

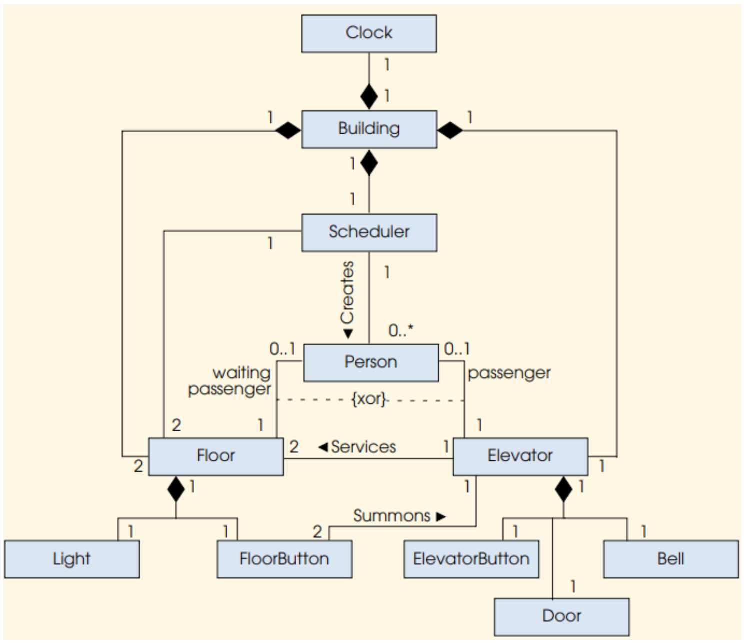


Figure 6: Full class diagram for elevator simulation.

Class Building is represented near the top of the diagram and is composed of four classes, including Clock and Scheduler. These two classes make up the controller portion of the simulation. Class Building is also composed of class Elevator and class Floor (notice the one-to-two relationship between class Building and class Floor). Classes Floor and Elevator are modeled near the bottom of the diagram. Class Floor is composed of one object each of classes Light and FloorButton. Class Elevator is composed of one object each of classes ElevatorButton, class Door and class Bell. The classes involved in an association can also have roles. Roles help clarify the relationship between two classes. For example, class Person plays the "waiting passenger" role in its association with class Floor (because the person is waiting for the elevator.) Class Person plays the passenger role in its association with class Elevator. In a class diagram, the name of a class's role is placed on either side of the association line, near the class's rectangle. Each class in an association can play a different role.

The association between class Floor and class Person indicates that an object of class Floor can relate to zero or one objects of class Person. Class Elevator also relates to zero or one objects of class Person. The dashed line that bridges these two association lines indicates a constraint on the relationship between classes Person, Floor and Elevator. The constraint says that an object of class Person can participate in a relationship with an object of class Floor or with an object of class Elevator, but not both objects at the same time. The notation for this relationship is the word "xor" (which stands for "exclusive or") placed inside braces. The association between class

Scheduler and class Person states that one object of class Scheduler creates zero or more objects of class Person.

Object Diagrams

The UML also defines object diagrams, which are similar to class diagrams, except that they model objects and links—links are relationships between objects. Like class diagrams, object diagrams model the structure of the system. Object diagrams present a snapshot of the structure while the system is running—this provides information about which objects are participating in the system at a specific point in time.

Figure 7 models a snapshot of the system when no one is in the building (i.e., no objects of class Person exist in the system at this point in time). Object names are usually written in the form: objectName : ClassName. The first word in an object name is not capitalized, but subsequent words are. All object names in an object diagram are underlined. We omit the object name for some of the objects in the diagram (e.g., objects of class FloorButton). In large systems, many names of objects will be used in the model. This can cause cluttered, hard-to-read diagrams. If the name of a particular object is unknown or if it is not necessary to include the name (i.e., we only care about the type of the object), we can leave the object name out. In this instance, we simply display the colon and the class name.

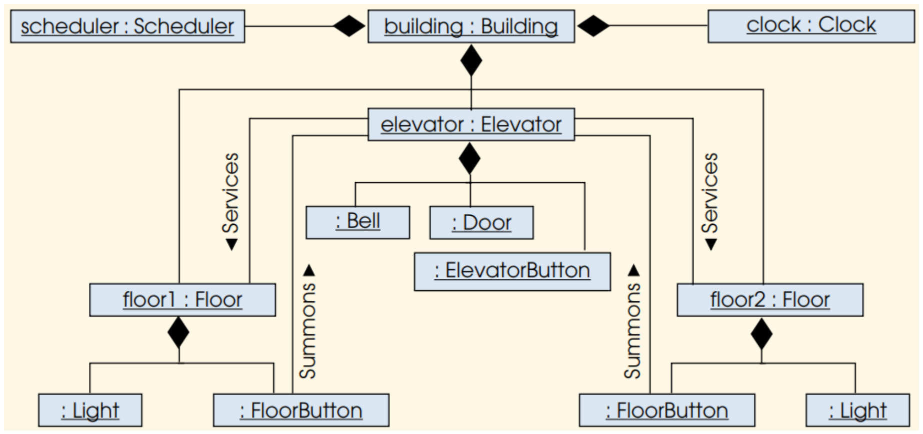


Figure 7: Object diagram of empty building.

Now we have identified the classes for this system (although we may discover others in later phases of the design process). We have also examined the system's use case.

Questions

1. How might you decide whether the elevator is able to handle the anticipated traffic volume?
2. Why might it be more complicated to implement a three-story (or taller) building?
3. It is common for large buildings to have many elevators. Once we have created one elevator object, it is easy to create as many as we want. What problems or opportunities do you foresee in having several elevators, each of which may pick up and discharge passengers at every floor in a large building?
4. For simplicity, we have given our elevator and each floor a capacity of one passenger. What problems or opportunities do you foresee in being able to increase these capacities?

SUMMARY

- In the normal state, execution of the program is gradual execution. When the Sequential Execution of the program is blocked by the selected Statements, it is called Branches Execution.
- The if statement is a control statement that is used to test a particular condition. In this, the condition is executed only once when the condition is true.
- The if-else statement is used to test a particular condition. If the condition is true then the if statement is executed if the condition is false then the else statement is executed.
- Switch case statement has expression and some cases related to it. The case which matches that expression or declares variable is printed in the output.
- The condition of if is true in else_if statement then statement of if is executed.
- The statements that cause a set of statements to be executed repeatedly either for a specific number of times or until some condition is satisfied are known as **iteration statements**. That is, as long as the condition evaluates to True, the set of statement(s) is executed. The various iteration statements used in C++ *are for loop, while loop* and *do while loop.*
- The for loop is one of the most widely used loops in C++. The for loop is a deterministic loop in nature, that is, the number of times the body of the loop is executed is known in advance.
- The while loop is used to perform looping operations in situations where the number of iterations is not known in advance. That is, unlike the for loop, the while loop is non deterministic in nature.
- The statements written in the program are implemented one after the other.
- Jump statements are used to interrupt the normal flow of program.
- The execution of the loops and switch cases of the Break Statement Program stops at any condition.
- The continue statement is used when we want to run the loop continues with the next iteration and skip other statements in the loop for the current iteration.
- When a goto statement executes its next statement except for some statement, it is called Forward goto statement and goes to its previous label to execute any previous or executed statement again, it is called Backward goto statement.

KNOWLEDGE CHECK

1. **Which of the following can replace a simple if-else construct?**
   a. Ternary operator
   b. while loop
   c. do-while loop
   d. for loop
2. **Which of the following is an entry-controlled loop?**
   a. do-while loop
   b. while loop
   c. for loop
   d. Both (B) and (C)
3. **Which of the following is most suitable for a menu-driven program?**
   a. do-while loop
   b. while loop
   c. for loop
   d. All of these
4. **Consider the following loop :**
   for(int i=0; i<5; i++) ;
   What will be the value of i after this loop?
   a. It will give compilation error.
   b. 5
   c. 6
   d. Some Garbage value
5. **A switch construct can be used with which of the following types of variable?**
   a. int
   b. int, char
   c. int, float, char
   d. Any basic datatype
6. **Which of the following must be present in switch construct?**
   a. Expression in ( ) after switch
   b. default
   c. case followed by value
   d. All of these

7. **What is the effect of writing a break statement inside a loop?**
   a. It cancels remaining iterations.
   b. It skips a particular iteration.
   c. The program terminates immediately.
   d. Loop counter is reset.
8. **What is the effect of writing a continue statement inside a loop?**
   a. It cancels remaining iterations.
   b. It skips execution of statements which are written below it.
   c. The program terminates immediately.
   d. Loop counter is reset.
9. **If the variable count exceeds 100, a single statement that prints "Too many" is**
   a. if (count<100) cout << "Too many";
   b. if (count>100) cout >> "Too many";
   c. if (count>100) cout << "Too many";
   d. None of these.
10. **The break statement causes an exit**
   a. from the innermost loop only.
   b. only from the innermost switch.
   c. from all loops & switches.
   d. from the innermost loop or switch.

REVIEW QUESTIONS

1. What is control flow explain with example?
2. How many types of control flow are there in CPP?
3. What is the difference between looping and iteration?
4. What are iterative or looping statements?
5. What is meant by iterative structure?
6. What is sequential control structure in C++?

Check Your Result

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

REFERENCES

1. David Anthony Watt; William Findlay (2004). Programming language design concepts. John Wiley & Sons. p. 228. ISBN 978-0-470-85320-7.
2. Kozen, Dexter (2008). "The Böhm–Jacopini Theorem Is False, Propositionally". Mathematics of Program Construction (PDF). Lecture Notes in Computer Science. 5133. pp. 177–192. CiteSeerX 10.1.1.218.9241. doi:10.1007/978-3-540-70594-9_11. ISBN 978-3-540-70593-2.

Index

Level: Beginner to Advanced
Subject: Computer and Information Science

Basic Computer Coding: C++

2nd Edition

C++ is an object-oriented programming language. It is an extension to C programming. C++ is a general purpose, case-sensitive, free-form programming language that supports object-oriented, procedural and generic programming. It has imperative, object-oriented and generic programming features. C++ runs on lots of platform like Windows, Linux, Unix, Mac etc. It is also popular in communications and gaming. It is used in many other industries: health care, finances, and even defense. Any program that was developed in the original C language can easily be moved into C++ without any major modifications. Because C++ offers flexibility, programmers are able to create powerful constructs and introduce new conceptual objects and abstract applications. As a result, C++ allows programmers to directly control and manipulate hardware resources to produce high functioning programs.

This edition is systematically divided into eight chapters. This book is designed to equip with C++ programming fundamentals including objects and classes, language reference, inheritance, templates, exceptions, and static class members.

ISBN 978-1-98467-608-5

9 781984 676085 00000

BIBLIOTEX
Digital Library

www.3ge-learning.com
email: info@3ge-learning.com