Learner Guide

# Faculty of Information Technology

## PROGRAMMING C++ oops 621

Year 2 Semester 1

RICHFIELD

richfield.ac.za

# LEARNER GUIDE

**MODULES: TECHNICAL PROGRAMMING (OOP with C++)**

## SECTION A: PREFACE

### 1. WELCOME

Welcome to the Faculty of Media, Information & Communication Technology at WL/ ICL9[5
We trust you will find the contents and learning outcomes of this module both interesting and insightful as you begin your academic journey and eventually your career in the business world.

This section of the study guide is intended to orientate you to the module before the commencement of formal lectures.
Please note that this study guide covers the content of various academic programmes at different levels of the NQF and HEQF. Your lecturers will provide further guidance and additional study materials covering

parts of the syllabi that may have been omitted from this study guide. Learners who are undertaking the Higher Certificate or Diploma Qualification may use the non-compulsory material supplied as additional reading. This will however not be directly examinable.

The following lectures will focus on the common study units described:

| **SECTION A: WELCOME & ORIENTATION** | |
|---|---|
| **Study unit 1: Orientation Programme**<br><br>Introducing academic staff to the learners by academic head. Introduction of institution policies. | **Lecture 1** |
| **Study unit 2: Orientation of Learners to Library and Students Facilities**<br><br>Introducing learners to physical structures | **Lecture 2** |
| **Study unit 3: Distribution and Orientation of Programming 621 Learner Guides, Textbooks and Prescribed Materials** | **Lecture 3** |
| **Study unit 4: Discussion on the Objectives and Outcomes of Programming OOPS USING C++ 621** | **Lecture 4** |
| **Study unit 5: Orientation and guidelines to completing Assignments**<br><br>Review and Recap of Study units 1-4 | **Lecture 5** |

**2. <u>TITLE OF MODULES, COURSE, CODE, NQF LEVEL, CREDITS & MODE OF DELIVERY</u>**

| 1st Semester | |
|---|---|
| **Title Of Module:**<br>**Code:**<br>**NQF Level:**<br>**Credits:**<br>**Mode of Delivery:** | OOPS using C++<br>PROG 621 C++<br>6<br>10<br>Contact |

**3. <u>PURPOSE OF MODULE</u>**

**3.1 <u>C++ 621</u>**

The purpose of this module is to introduce the learners with programming concepts of C++.

**3.2 <u>DATA STRUCTURES USING C++ 621 (1st Semester)</u>**

To provide learners with an informed understanding of C++ programming concepts.

**3.3 <u>OOPS USING C++ 621 (1st Semester)</u>**

To introduce learners to sorting algorithms and to present additional material on abstract classes and also further demonstrate the use of OOP.

**4. <u>LEARNING OUTCOMES</u>**

On completion of these modules the student will be able to:

- Become familiar with different types of programming languages and understand a typical C++ program development environment
- Understand basic problem solving techniques and to create new functions and to understand the mechanisms used to pass information between functions
- Use pointers to pass arguments to functions by call-by-reference and understand the close relationship among pointers, arrays and strings
- A sound understanding of how to use C++ object-oriented stream input/ output, format inputs and outputs and tie output streams to input streams.
- A sound understanding of standard template library (STL) containers, container adapters and "near containers" and become familiar with the STL resources available on the Internet and the World Wide Web.
- A sound understanding of Standard C++ Languages Additions
- Fundamental knowledge of a Data structure - primitive and composite Data Types, Asymptotic notations, Arrays, Operations on Arrays, Order lists.
- A solid knowledge base of stack, Queues - Operations on Queues, Queue Applications and Circular Queue.
- A solid knowledge base of Linked lists, Doubly Linked List - Operations, Applications
- A solid knowledge base of Binary trees and Graphs
- A sound understanding of Sort and Search Algorithms - Definition - Examples Bubble sort, Insert sort, heap sort- Binary Search - Maximum and Minimum - Merge Sort.

- Use function templates to create a group of related (overload) functions, understand the relationships among templates, friends, inheritance and static members and how to overload template functions
- Use **try, throw** and **catch** to watch for, indicate and handle exceptions respectively, use **auto-ptr** to prevent memory leaks and understand the standard exception hierarchy
- Become familiar with different types of programming languages and understand a typical C++ program development environment.
- Fundamental knowledge of Object Oriented Programming (OOP), Benefits of OOP and Applications of OOP
- Informed understanding of the important rules of Tokens, Keywords, Identifiers, Variables, Operators, Manipulators, Expressions and Control Structures
- Understand basic problem solving techniques and to create new functions and to understand the mechanisms used to pass information between functions.
- Understand the software engineering concepts of encapsulation and data hiding, the notions of data abstraction and abstract data types (adts) and create C ++ adts, namely classes.
- Informed understanding of file manipulations including error handling techniques for file operations.
- Understand the purpose of friend functions and friend classes, understand the concept of a container class, understand the notion of iterator classes that walk through the elements of container classes and understand the use of the "this pointer".
- Understand how to redefine (overload) operators to work with new types, understand how to convert objects from one class to another class and learn when to, and when not to, overload operators.
- Create new classes by inheriting from existing classes, understand how inheritance promotes software reusability, and use multiple inheritance to derive a class from several base classes
- Understand the notion of polymorphism, how to declare and use virtual functions to affect polymorphism and understand how C++ implements virtual functions and dynamic binding "under the hood".
- Sound knowledge of working with files
- Use class **string** from the C++ standard library to treat **strings** as full-fledged objects and perform input from and output to **strings** in memory.

## 5. METHOD OF STUDY

The sections that have to be studied are indicated under each topic. These form the basis for tests, assignments and examination. To be able to do the activities and assignments for this module, and to achieve the learning outcomes and ultimately to be successful in the tests and examination, you will need an in-depth understanding of the content of these sections in the learning guide and prescribed book.

In order to master the learning material, you must accept responsibility for your own studies. Learning is not the same as memorising. You are expected to show that you understand and are able to apply the information. Use will also be made of lectures, tutorials, case studies and group discussions to present this module.

## 6. LECTURES AND TUTORIALS

Learners must refer to the notice boards on their respective campuses for details of the lecture and tutorial time tables. The lecturer assigned to the module will also inform you of the number of lecture periods and tutorials allocated to a particular module. Prior preparation is required for each lecture and tutorial. Learners are encouraged to actively participate in lectures and tutorials in order to ensure success in tests, assignments and examinations.

## 7. NOTICES

All information pertaining to this module such as tests dates, lecture and tutorial time tables, assignments, examinations etc. will be displayed on the notice board located on your campus. Learners must check the notice board on a daily basis. Should you require any clarity, please consult your lecturer, or programme manager, or administrator on your respective campus.

## 8. PRESCRIBED & RECOMMENDED MATERIAL

### 8.1 Prescribed Materials:

8.1.м C++ How to program мл th Edition by H.M. Deitel and P.J. Deitel, Prentice Hall, 20мс

8.1.2 Data Structures using C++ by D.S. Malik, Course Technology, 2008, ISBN 978-0-619-15907-8

8.1.3 C++ programming 4th Edition by D. S. Malik, Course Technology, 2009, ISBN-13:978-1- 4239-0209-6

8.1.п E. Horowitz and S. Shani Fundamentals of Data Structures in C++, Galgotia Pub. 1999.

The purchasing of prescribed books is for the learners own account and are compulsory for all learners. This guide will have limited value if not accompanied by the prescribed text books.

### 8.2 Recommended Materials:

8.н.м C++ How to program óŜl·₪fè⌀ƻōʔŜŎú⌀ǾŜ₪ăƻƴŭɸth Edition by H.M. Deitel and P.J. Deitel, Prentice Hall, 20мо

8.2.2 Stair R, Reynolds G, Chesney T: (2008) **Principles of Business Information Systems.** Course Technology

**NB:** Learners please note that there will be a limited number of copies of the recommended texts and reference material that will be made available at your campus library. Learners are advised to make copies or take notes of the relevant information, as the content matter is examinable.

### 8.3 Independent Research:

The student is encouraged to undertake independent research

### 8.4 Library Infrastructure

The following services are available to you:

**8.4.1** Each campus keeps a limited quantity of the recommended reading titles and a larger variety of similar titles which you may borrow. Please note that learners are required to purchase the prescribed materials.

**8.4.2** Arrangements have been made with municipal, state and other libraries to stock our recommended reading and similar titles. You may use these on their premises or borrow them if available. It is your responsibility to safe keeps all library books.

**8.4.3** PCT&BC has also allocated one library period per week as to assist you with your formal research under professional supervision.

**8.4.4** The computers laboratories, when not in use for academic purposes, may also be used for research purposes. Booking is essential for all electronic library usage.

## 9. ASSESSMENT

Final Assessment for this module will comprise two Continuous Assessment tests, an assignment and an examination. Your lecturer will inform you of the dates, times and the venues for each of these. You may also refer to the notice board on your campus or the Academic Calendar which is displayed in all lecture rooms.

### 9.1 Continuous Assessment Tests

There are two compulsory tests for each module (in each semester).

### 9.2 Assignment

There is one compulsory assignment for each module in each semester. Your lecturer will inform you of the Assessment questions at the commencement of this module.

### 9.3 Examination

There is one two hour examination for each module. Make sure that you diarize the correct date, time and venue. The examinations department will notify you of your results once all administrative matters are cleared and fees are paid up.
The examination may consist of multiple choice questions, short questions and essay type questions. This requires you to be thoroughly prepared as all the content matter of lectures, tutorials, all references to the prescribed text and any other additional documentation/reference materials is examinable in both your tests and the examinations.

The examination department will make available to you the details of the examination (date, time and venue) in due course. You must be seated in the examination room 15 minutes before the commencement of the examination. If you arrive late, you will not be allowed any extra time. Your learner registration card must be in your possession at all times.

## 9.4 Final Assessment

The final assessment for this module will be weighted as follows:

| | | | |
|---|---|---|---|
| Continuous Assessment Test 1 | | | |
| Continuous Assessment Test 2 | | 40 % | |
| | | | Assignment 1 |
| Total Continuous Assessment | 40% | | |
| Semester Examinations | | 60% | |
| Total | | 100% | |

## 9.5 Key Concepts in Assignments and Examinations

In assignment and examination questions you will notice certain key concepts (i.e. words/verbs) which tell you what is expected of you. For example, you may be asked in a question to list, describe, illustrate, demonstrate, compare, construct, relate, criticize, recommend or design particular information / aspects / factors /situations. To help you to know exactly what these key concepts or verbs mean so that you will know exactly what is expected of you, we present the following taxonomy by Bloom, explaining the concepts and stating the level of cognitive thinking that theses refer to.

| ***COMPETENCE*** | ***SKILLS DEMONSTRATED*** |
|---|---|
| **Knowledge** | observation and recall of information<br>knowledge of dates, events, places knowledge<br>of major ideas |
| | mastery of subject matter<br>***Question Cues*** list, define, tell, describe, identify, show, label, collect, examine, tabulate, quote, name, who, when, where, etc. |
| **Comprehension** | understanding information grasp<br>meaning translate knowledge into<br>new context interpret facts,<br>compare, contrast order, group,<br>infer causes predict consequences<br>***Question Cues***<br>summarize, describe, interpret, contrast, predict, associate, distinguish, estimate, differentiate, discuss, extend |
| **Application** | use information<br>use methods, concepts, theories in new situations solve<br>problems using required skills or knowledge<br>***Questions Cues***<br>apply, demonstrate, calculate, complete, illustrate, show, solve, examine, modify, relate, change, classify, experiment, discover |

| | |
|---|---|
| **Analysis** | seeing patterns organization of<br>parts recognition of hidden<br>meanings identification of<br>components<br>***Question Cues***<br>analyze, separate, order, explain, connect, classify, arrange, divide, compare, select, explain, infer |
| **Synthesis** | use old ideas to create new ones<br>generalize from given facts relate<br>knowledge from several areas<br>predict, draw conclusions<br>***Question Cues***<br>combine, integrate, modify, rearrange, substitute, plan, create, design, invent, what if?, compose, formulate, prepare, generalize, rewrite |
| **Evaluation** | compare and discriminate between ideas<br>assess value of theories, presentations make<br>choices based on reasoned argument verify<br>value of evidence recognize subjectivity<br>***Question Cues***<br>assess, decide, rank, grade, test, measure, recommend, convince, select, judge, explain, discriminate, support, conclude, compare, summarize |

## 10. SPECIMEN ASSIGNMENT COVER SHEET

**C++ 621 (1st Semester)**
**Assignment Cover Sheet**
**(To be attached to all Assignments – hand written or typed)**

**Name of Learner:………………………… Student No: …………………………..**
**Module:…………………………………… Date: …………………………………. ICAS Number ………………………… ID Number …………………………..**

***The purpose of an assignment is to ensure that one is able to:***

- Interpret, convert and evaluate text.
- Have sound understanding of key fields viz principles and theories, rules, concepts and awareness of how to cognate areas.
- Solve unfamiliar problems using correct procedures and corrective actions.
- Investigate and critically analyze information and report thereof.
- Present information using Information Technology.
- Present and communicate information reliably and coherently.
- Develop information retrieval skills.
- Use methods of enquiry and research in a disciplined field.

**ASSESSMENT CRITERIA**
***(NB: The allocation of marks below may not apply to certain modules like EUC and Accounting)***
**A. Content- Relevance.**

| Question Number | Mark Allocation | Examiner's Mark | Moderator's Marks | Remarks |
|---|---|---|---|---|
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| 6 | | | | |
| 7 | | | | |
| 8 | | | | |
| 9 | | | | |
| 10 | | | | |
| **Sub Total** | **70 Marks** | | | |
| **B. Research (**A minimum of "**TEN SOURCES**" is recommended) | | | | |
| Library, EBSCO, Emerald Journals, Internet, Newspapers, Journals, Text Books, Harvard method of referencing | | | | |
| **Sub Total** | **15 Marks** | | | |
| **C. Presentation** | | | | |
| Introduction, Body, Conclusion, Paragraphs, Neatness, Integration, Grammar / Spelling, Margins on every page, Page Numbering, Diagrams, Tables, Graphs, Bibliography | | | | |
| **Sub Total** | **15 Marks** | | | |

|  |  |  |  |  |
|---|---|---|---|---|
| **Grand Total** | **100Marks** |  |  |  |

***NB: All Assignments are compulsory as it forms part of continuous assessment that goes towards the final mark.***

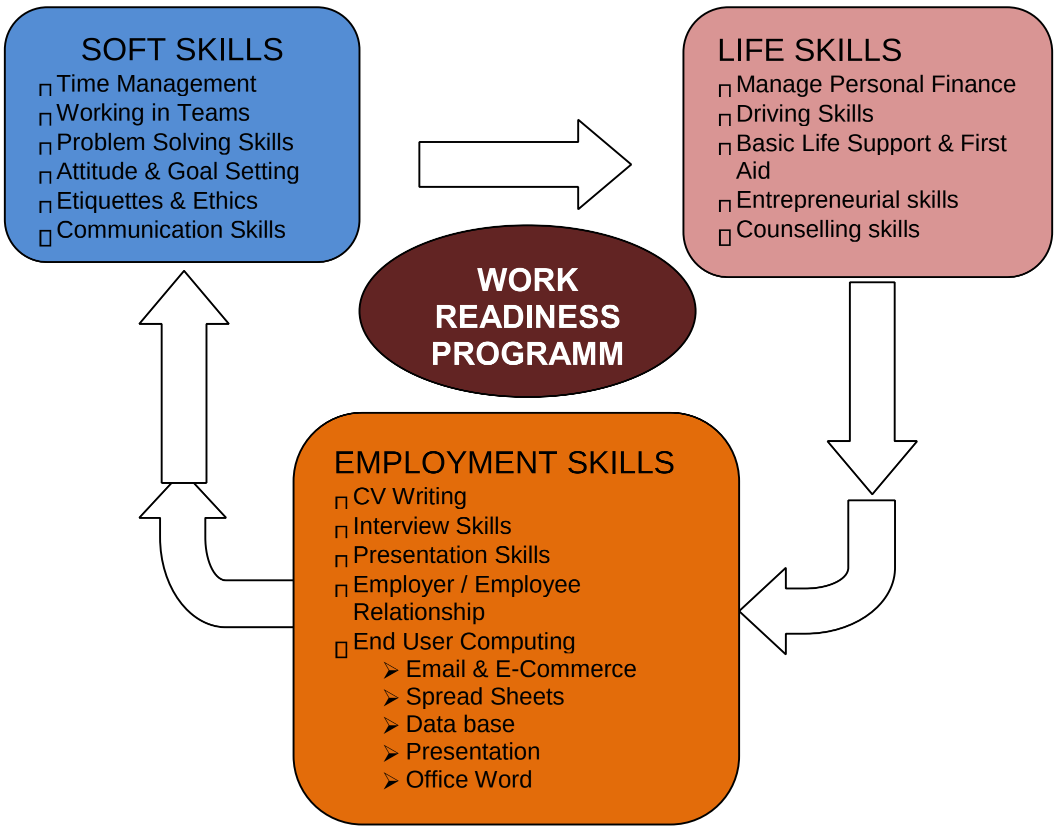
## 11. <u>WORK READINESS PROGRAMME (WRP)</u>

In order to prepare learners for the world of work, a series of interventions over and above the formal curriculum, are concurrently implemented to prepare learners.

**These include:**

- Soft skills
- Employment skills
- Life skills
- End –User Computing (if not included in your curriculum)

The illustration below outlines some of the key concepts for Work Readiness that will be included in your timetable.



It is in your interest to attend these workshops, complete the Work Readiness Log Book and prepare for the Working World.

## 12. WORK INTEGRATED LEARNING (WIL)

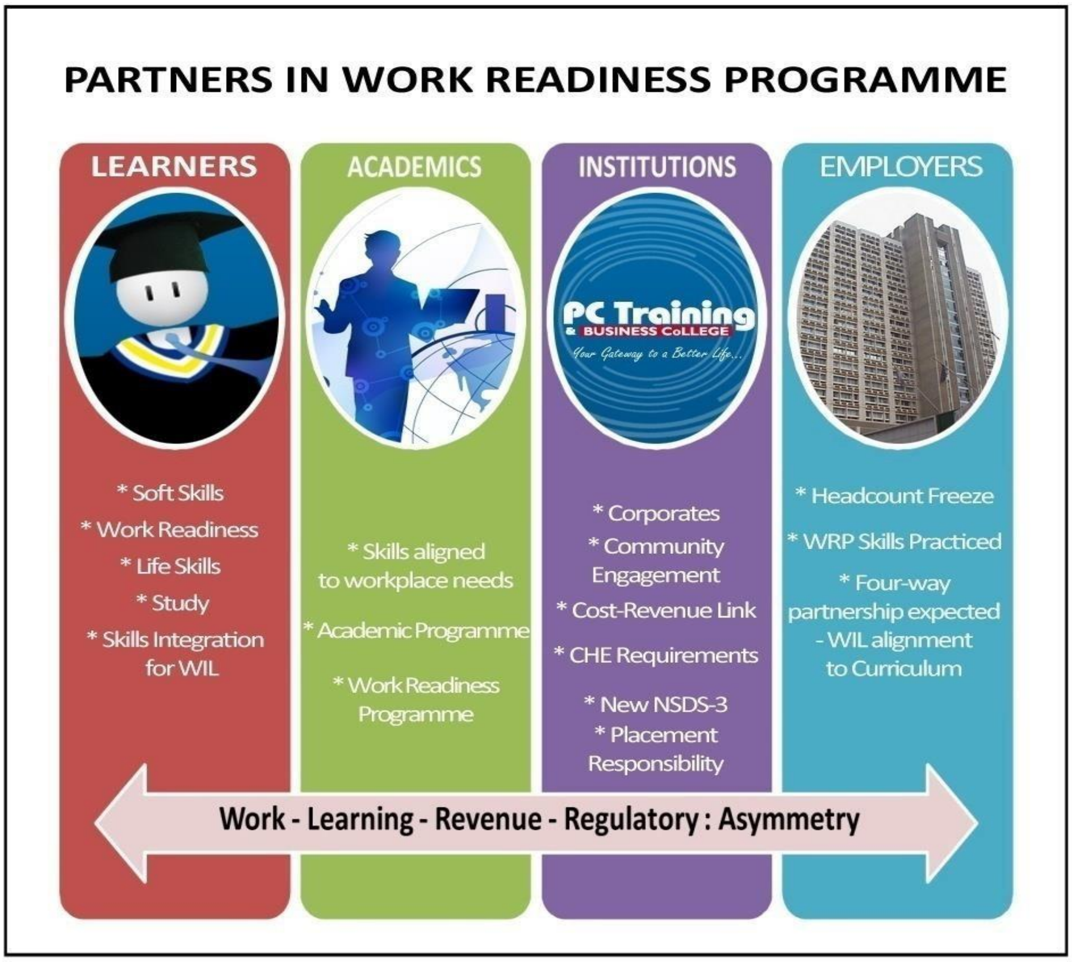
Work Integrated Learning forms a core component of the curriculum for the completion of this programme. All modules making of the Bachelor of Science in Information Technology will be assessed in an integrated manner towards the end of the programme or after completion of all other modules.

**Prerequisites for placement with employers will include:**

- Completion of all tests & assignment
- Success in examination      Payment of all arrear fees
- Return of library books, etc.
- Completion of the Work Readiness Programme.

Learners will be fully inducted on the Work Integrated Learning Module, the Workbooks & assessment requirements before placement with employers.

The partners in Work Readiness Programme (WRP) include:


PARTNERS IN WORK READINESS PROGRAMME
LEARNERS
ACADEMICS
INSTITUTIONS
EMPLOYERS
PC Training & BUSINESS CoLLEGE
Your Gateway to a Better Life...
* Soft Skills
* Work Readiness
* Life Skills
* Study
* Skills Integration for WIL
* Skills aligned to workplace needs
* Academic Programme
* Work Readiness Programme
* Corporates
* Community Engagement
* Cost-Revenue Link
* CHE Requirements
* New NSDS-3
* Placement Responsibility
* Headcount Freeze
* WRP Skills Practiced
* Four-way partnership expected - WIL alignment to Curriculum
Work - Learning - Revenue - Regulatory : Asymmetry

***Good luck with your studies…***

**Isaka Reddy Academic Manager: Faculty of Media, Information and Communication Technology**

## SECTION B

# FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

# LEARNER GUIDE

## MODULES: OBJECT ORIENTED PROGRAMMING USING C++ 621 (1st SEMESTER)

# TOPIC 1

# 1. PRINCIPLES OF OBJECT ORIENTED PROGRAMMING (OOP)

**LEARNING OUTCOMES**

***After Studying this topic you should be able to:***

- Fundamental knowledge of Object Oriented Programming (OOP)
- Become familiar with different types of programming languages
- Benefits of OOP
- Applications of OOP

**Software Crisis:**

Developments in software technology continue to be dynamic. New tools and techniques are announced in quick succession. This has forced the software industry and software engineers to continuously look for new approaches to software design and development, which is becoming more and more critical in view of the increasing complexity of software systems as well as the highly competitive nature of the industry. The following issues needed to be resolved to overcome this crisis:

- How to represent real-life entities of problems in system-design?
- How to design systems with open interfaces?
- How to ensure reusability and extensibility of modules?
- How to develop modules that are tolerant to any change in future?
- How to improve software productivity and decrease software cost?
- How to manage time schedules?
- How to improve the quality of software?
- How to industrialize the software development process?

Problems occur when software products are either not finished, not used or are delivered with errors. Changes in user requirements have always been a major problem. The reports on software implementation suggest that software products should be evaluated carefully for their quality before they are delivered and implemented. Some of the quality issues that must be considered for critical evaluation are:

- Correctness
- Maintainability
- Reusability
- Openness and interpretability
- Portability
- Security
- Integrity
- User friendliness

## 1.1 Software Evolution

Ernest Tello, a well-known writer in the field of artificial intelligence, compares the evolution of software technology to the growth of a tree. Like a tree, the software evolution has had distinct phases or "layers" of growth. These layers were built up by one over the last four decades as shown in figure 1, with each layer representing an improvement over the previous one. However, the analogy fails if we consider life of these layers. In software systems, each of the layers continues to be functional, whereas in the case of trees, only the uppermost layer is functional.

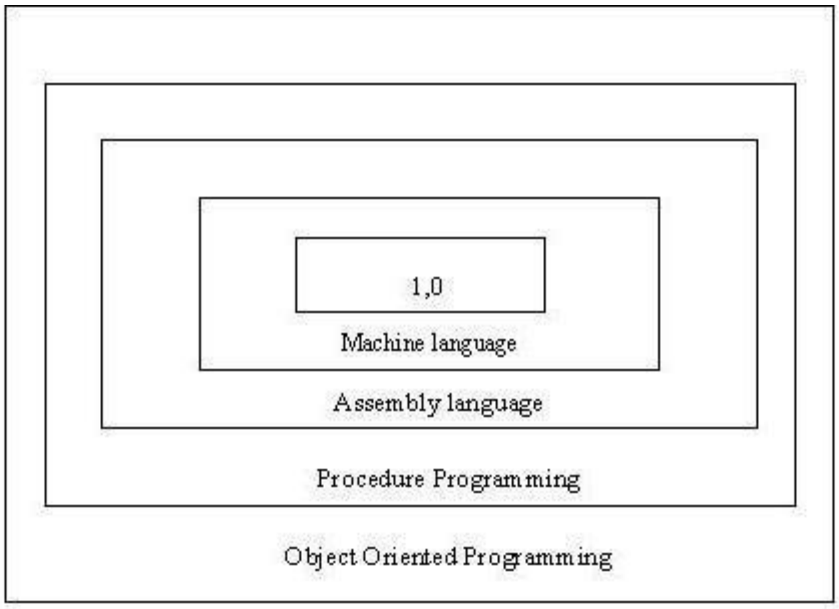


Figure 1

Objected-oriented programming (OOP) is a new way of approaching the job of programming. Approaches to programming have changed dramatically since the invention of the computer, primarily to accommodate the increasing complexity of program. Assembly language was invented so that a program could deal with longer, increasingly complex programs using symbolic representations of the machine instructions. As programs continued to grow, high level languages were introduced that gave the programmer more tools with which to handle complexity. The first widespread language was FORTRAN. Although FORTRAN was a very impressive first step, it is hardly a language that encourages clear, easyto–understand programming.

The 1960 gave birth to structured programming. This is the method encouraged by languages such as C and Pascal. They are of structured languages, which made it possible to write moderately complex programs fairly easily. However, even using structured programming methods, a project becomes uncontrollable once it reaches a certain size i.e. once its complexity exceeds that which a program can manage.

At each milestone in the development of programming, methods were created to allow the programmer to deal with greater complexity. Each step of the way, the new approach uses the best elements of the previous method and moved forward. To solve this problem, object oriented programming was invented.

Before going into details about object oriented programming, have a look at procedure oriented programming. In the procedure-oriented approach, the problem is viewed as a sequence of things to be done, such as reading, calculating and printing. A number of functions are written to accomplish these tasks. The primary focus is on functions. A typical program structure for procedural programming is shown in Fig 2. The technique of hierarchical decomposition has been used to specify the tasks to be completed in order to solve a problem.

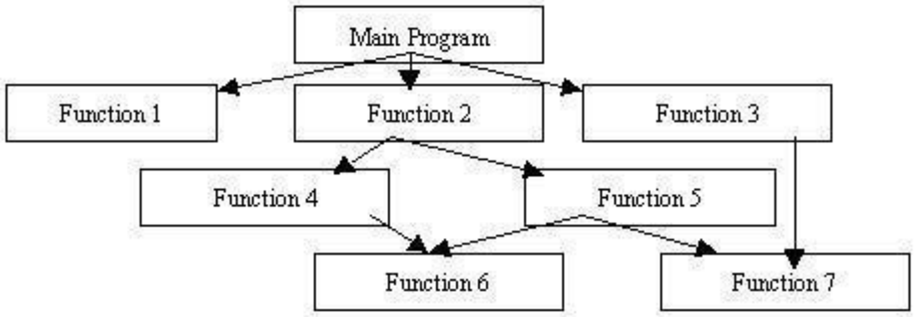


**Fig 2**

Procedure-oriented programming basically consists of writing a list of instructions for the computer to follow, and organizing these instructions into groups known as functions. We normally use a flowchart to organize these actions and represent the flow of control from one action to another.

While we concentrate on the development of functions, very little attention is given to the data that are being used by various functions. What happens to data? How are they affected by the functions that work on them?

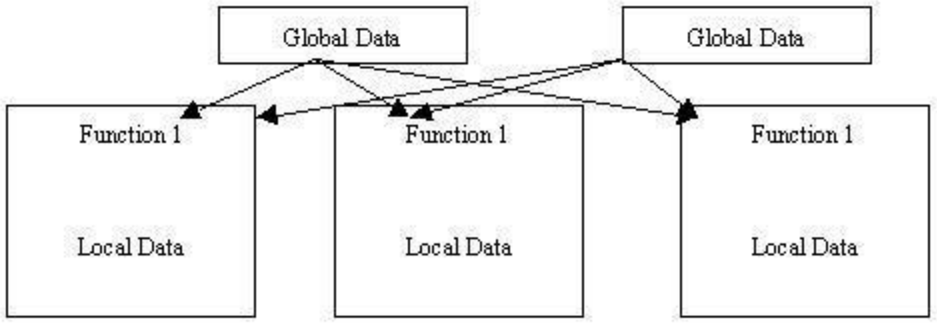


**Fig 3**

In multi-function programming, many important data items are places as global so that they may be accessed by all the functions. Each function may have its own local data. Fig 3 shows the relationship of data and functions in a procedure.

In a multi-function program, many important data items are placed as global so that they may be accessed by all the functions. Each function-may have its own local data. Global data are more vulnerable to an inadvertent change by a function. In a large program it is very difficult to identify what data is used by which function. In case we need to revise an external data structure, we should also revise all functions that access the data. This provides an opportunity for bugs to creep in.

Another serious drawback with the procedural approach is that it does not model real world problems very well. This is because functions are action-oriented and do not really corresponding to the elements of the problem.

Some characteristics exhibited by procedure-oriented programming are:

- Emphasis is on doing things (algorithms).
- Large programs are divided into smaller programs known as functions.

- Most of the functions share global data.
- Data move openly around the system from function to function. Functions transforms data from one form to another

Employs top-down approach in program design.

## 1.2 Object-oriented programming paradigm

The major motivating factor in the invention of object-oriented approach is to salvage some of the flaws encountered in the procedural approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the functions that operate on it and protects it from accidental modification from outside functions. OOP allows us to decompose a problem into a number of entities called objects and then builds data and functions around these entities. The organization of data and functions in object-oriented programs is shown here.

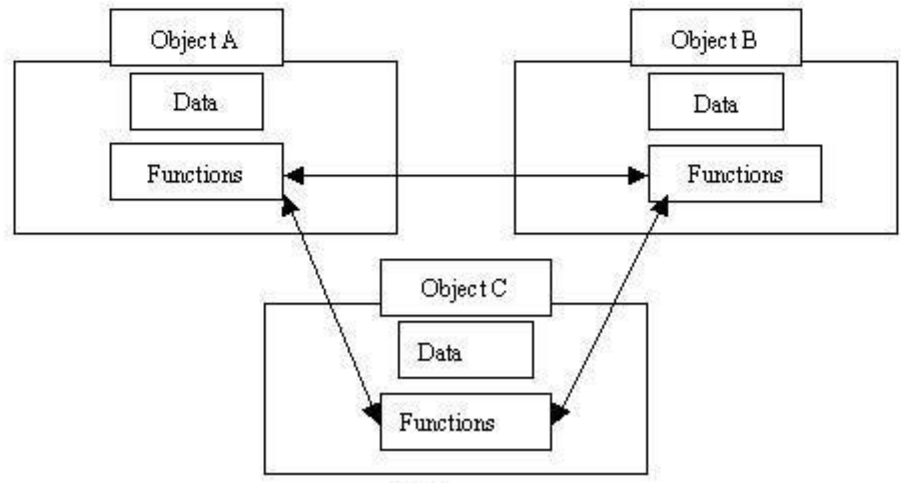


**Fig 4**

The data of an object can access the functions of other objects.

However, functions of one object can access the functions of other objects.

Some of the striking features of object-oriented programming are:
- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterise the objects.
- Functions that operate on the data of an object are tied together in the data structure.
- Data is hidden and cannot be accessed by external functions.
- Objects may communicate with each other through functions.
- New data and functions can be easily added whenever necessary.
- Follows bottom-up approach in program design.

Object-oriented programming is the most recent concept among programming paradigms; it means different things to different people. It is therefore important to have a working definite definition of object-oriented programming before we proceed further. Our definition of object-oriented programming is as follows “Object-oriented programming is an approach that provides a way of moulding programs by creating partitioned memory area for both data, and functions that can be used as templates for creating copies of such modules on demand."

That is, an object is considered to be a partitioned area of computer memory that stores data and set of operations that can access that data. Since the memory partitions are independent, the objects are used in a variety of different programs without modifications.

**What is object oriented programming?**

Object oriented programming (OOP) have taken the best ideas of structured programming and combined them with several powerful new concept that encourage you to approach the task of programming in a new way. In general when programming in an object oriented fashion you break down a problem into subgroups of related parts that take into account both code and data related to each group. Also, you organise this subgroup into a hierarchy for all intents and purpose, an object in a variable of area-defined type. It may seem strong at first to think of an object, which lines both code and data on variable. However, in object oriented programming, this is precisely the case. When you define an object, you are implicitly creating a new data type.

## 1.3 Basic concepts of object-oriented programming

'Object-Oriented' remains a term, which is interpreted differently by different people. It is therefore necessary to understand some of the concepts used extensively in object-oriented programming. We shall discuss in this section the following general concepts:

1.3.1 Objects
1.3.2. Classes
1.3.3. Data abstraction
1.3.4. Inheritance
1.3.5. Dynamic binding
1.3.6. Data encapsulation
1.3.7. Polymorphism
1.3.8. Message passing

### 1.3.1. Objects

Objects are the basic run-time entities in an object-oriented system. They may represent a person, i.e. a bank account, a table of data or any item that the program must handle. They may also resent userdefined data such as vectors, time and lists. A programming problem is analysed in terms of objects and the nature of communication between them. Program objects should be chosen such that they match closely with the real-world objects. As pointed out earlier, objects take up space in the memory and have an associated address like a record in Pascal, or a structure in C.

When a program is executed, the objects interact by sending messages to one another. For example if "customer" and "account" are two objects in a program, then the customer object may send a message to the account object requesting for the bank balance. Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code. It is sufficient to know the type of message accepted and the type of response returned by the objects, although different authors represent them differently.

### 1.3.2. Classes

We just mentioned that objects contain data and code to manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a class. In fact, objects are variables of type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created. A class is thus a collection of objects of similar type. For example, mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language. For example, the syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit has been defined as a class, then the statement fruit mango; will create an object mango belonging to the class fruit.

### 1.3.3. Data abstraction

The wrapping up of data and functions into a single unit (called class) is known as encapsulation. Data encapsulation is the most striking feature of a class. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program is called 'data hiding'.

Abstraction refers to the act of representing essential features without including the background details or explanations. Classes use the concept of abstraction and are defined as a list of abstract attributes such as size, weight and cost, and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. Since the classes use the concept of data abstraction, they are known as Abstract Data Types (ADT). **1.3.4. Inheritance**

Inheritance is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification. For example, the bird robin is a part of the class flying bird which is again a part of the class bird. As illustrated in Fig. 5, the principle behind this sort of division is that each derived class shares common characteristics with the class from which it is derived.

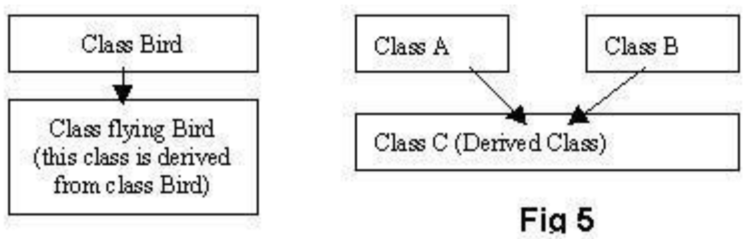


Fig 5

In OOP, the concept of inheritance provides the idea of reusability. This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes. The real appeal and power of the inheritance mechanism is that it allows the programmer to reuse a class that is almost, but not exactly, what he wants, and to tailor the class in such a way that it does not introduce any undesirable side effects into the rest of the classes.

Note that each sub-class defines only those features that are unique to it. Without the use of classification, each class would have to explicitly include all of its features.

### 1.3.5. Dynamic Binding

Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic binding means that the code associated with a given procedure call is not known until the time of the call at run-time. It is associated with polymorphism and inheritance. A function call associated with a polymorphic reference depends on the dynamic type of that reference.

Consider the procedure "draw" in Fig. 6. By inheritance, every object will have this procedure. Its algorithm is, however, unique to each object and so the draw procedure will be redefined in each class that defines the object. At run-time, the code matching the object under current reference will be called.

### 1.3.6. Encapsulation

Encapsulation is the mechanism that binds together code and data and that keeps both safe from outside interference or misuse. It also allows the creation of an object. More simply, an object is a logical entity that encapsulate both data and the code that manipulators that data.

Within an object, some of the code and/ or data may be private to the objected and is inaccessible to anything outside the object. In this way and object provides a significant level of protection against some other unrelated part of the program accidentally modifying or incorrectly using the private parts of the object.

### 1.3.7. Polymorphism

Polymorphism is another important OOP concept. Polymorphism means the ability to take more than one form. For example, an operation may exhibit different behaviour in different instances. The behaviour depends upon the types of data, used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings, then the operation would produce a third string by concatenation.

The picture below illustrates how a single function name can be used to handle different number and different types of arguments. This is something similar to a particular word having several different meanings depending on the context.

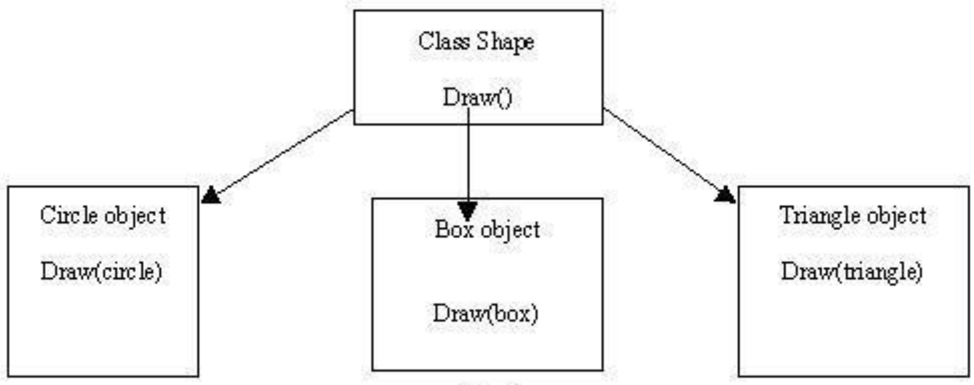


**Fig 6**

Polymorphism plays an important role in allowing objects having different internal structures to share the same external interface. This means that a general class of operations may be accessed in the same manner even through specific actions associated with each operation may differ. Polymorphism is extensively used in implementing inheritance.

Object oriented programming languages support polymorphism, which in characterised by the phase “on interface multiple method”. In simple terms, polymorphism in an attribute that allows one interface to be used with a general class of actions. Polymorphism helps in reducing complexity by allowing the same interface to specify a general class of action. It is the compiler’s job to select the “specific action” it applies to each situation. The programmers don’t need to make this selection manually, operator overloading, function overloading and overlooking are examples of polymorphism structures. Finally you translate these subgroups self-contained units called object.

### 1.3.8. Message Communication

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

1. Creating classes that define objects and their behaviour.
2. Creating objects from class definitions.
3. Establishing communication among objects.

Objects communicate with one another by sending and receiving information much the same way as people pass messages to one another. The concept of message passing makes it easier to talk about building systems that directly model or simulate their real-world counterparts.

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

Example:

Objects have a life cycle. They can be created and destroyed. Communication with an object is feasible as long as it is alive.

## 1.4 <u>Benefits of OOP</u>

OOP offers several benefits to both the program designer and the user. Object-orientation contribution to the solution of many problems associated with the development and quality of software products. The new technology promises greater programmer productivity, better quality of software and lesser maintenance cost. The principal advantages are:

1. Through inheritance, we can eliminate redundant code and extend the use of existing classes.
2. We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity.
3. The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.
4. It is possible to have multiple instances of an object to co-exist without any interference.
5. It is possible to map objects in the problem domain to those objects in the program.

6. It is easy to partition the work in a project based on objects.
7. The data-centered design approach enables us to capture more details of a model in implementable form.
8. Object-oriented systems can be easily upgraded from small to large systems.
9. Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.
10. Software complexity can be easily managed.

While it is possible to incorporate all these features in an object-oriented system, their importance depends on the type of the project and the preference of the programmer. There are a number of issues that need to be tackled to get some of the benefits stated above. For instance, object libraries must be available for reuse. The technology is still developing and current products may be superseded quickly. Strict controls and protocols need to be developed if reuse is not to be compromised.

Developing software that is easy to use makes it hard to build. It is hoped that the object-oriented programming tools would help manage this problem.

## 1.5 Object-oriented languages

Object-oriented programming is not the right of any particular language. Like structured programming, OOP concepts can be implemented using languages such as C and Pascal. However, programming becomes clumsy and may generate confusion when the programs grow large. A language that is specially designed to support the OOP concepts makes it easier to implement them.

The languages should support several of the OOP concepts to claim that they are object-oriented. Depending upon the features they support, they can be classified into the following two categories:

1. Object-based programming languages
2. Object-oriented programming languages

Object-based programming is the style of programming that primarily supports encapsulation and object identity.

Major features that are required for object-based programming are:

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

Languages that support programming with objects are said to be object-based programming languages. They do not support inheritance and dynamic binding.

Ada is a typical object-based programming language.
Object-oriented programming incorporates all of object-based programming features along with two additional features, namely, inheritance and dynamic binding.

## 1.6 <u>Application of OOP</u>

The Promising areas for application of oops include:

1. Real-time systems
2. Simulation and Modelling
3. Object-oriented databases
4. Hypertext, hypermedia and expertext
5. AI and expert systems
6. Neural networks and parallel programming
7. Decision support and office automation systems
8. CIM/CAM/CAD systems

# TOPIC 2

# 2. INTRODUCTION TO C++

**LEARNING OUTCOMES**

***After Studying this topic you should be able to:***

- Informed understanding of the important rules of Tokens, Keywords, Identifiers, Variables, Operators, Manipulators, Expressions and Control Structures
- Understand basic problem solving techniques and to create new functions and to understand the mechanisms used to pass information between functions.
- Understand a typical C++ program development environment.
- Understand Dynamic memory

## 2.1 C++ Program Structure

Let us look at a simple code that would print the words Hello World.

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

// main() is where program execution begins.

int main()
{
  cout << "Hello World"; // prints Hello World
return 0;
}
```

Let us look various parts of the above program:

1. The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header <iostream> is needed.
2. The line using namespace std; tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.
3. The next line // main() is where program execution begins. is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
4. The line int main() is the main function where program execution begins.
5. The next line cout << "This is my first C++ program."; causes the message "This is my first C++ program" to be displayed on the screen.
6. The next line return 0; terminates main( )function and causes it to return the value 0 to the calling process.

### 2.1.1 Semicolons & Blocks

In C++, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity. For example, following are three different statements:

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

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

```
{
  cout << "Hello World"; // prints Hello World
return 0;
}
```

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

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

Is the same as

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

### 2.1.2 Tokens

A token is the smallest element of a C++ program that is meaningful to the compiler. The C++ parser recognizes these kinds of tokens: identifiers, keywords, literals, operators, punctuators, and other separators. A stream of these tokens makes up a translation unit.

Tokens are usually separated by "white space." White space can be one or more: 
Blanks

- Horizontal or vertical tabs
- New lines
- Formfeeds
- Comments

### 2.1.3 Comments

A comment is text that the compiler ignores but that is useful for programmers. Comments are normally used to annotate code for future reference. The compiler treats them as white space. You can use comments in testing to make certain lines of code inactive.

A C++ comment is written in one of the following ways:

- The /* (slash, asterisk) characters, followed by any sequence of characters (including new lines), followed by the */ characters. This syntax is the same as ANSI C.
- The // (two slashes) characters, followed by any sequence of characters. A new line not immediately preceded by a backslash terminates this form of comment. Therefore, it is commonly called a "single-line comment."

The comment characters (/*, */, and //) have no special meaning within a character constant, string literal, or comment. Comments using the first syntax, therefore, cannot be nested.

### 2.1.4 Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).

C++ does not allow punctuation characters such as @, $, and % within identifiers. C++ is a case sensitive programming language. Thus Manpower and manpower are two different identifiers in C++.

Here are some examples of acceptable identifiers:

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

### 2.1.5 Keywords:

Keywords are predefined reserved identifiers that have special meanings.The following list shows the reserved words in C++. These reserved words may not be used as constant or variable or any other identifier names.

| | | | |
|---|---|---|---|
| Asm | else | new | this |
| Auto | enum | operator | throw |
| Bool | explicit | private | true |
| Break | export | protected | try |
| Case | extern | public | typedef |
| Catch | false | register | typeid |
| Char | float | reinterpret_cast | typename |
| Class | for | return | union |
| Const | friend | short | unsigned |
| const_cast | goto | signed | using |
| Continue | if | sizeof | virtual |

| | | | |
|---|---|---|---|
| Default | inline | static | void |
| Delete | int | static_cast | volatile |
| Do | long | struct | wchar_t |
| Double | mutable | switch | while |
| dynamic_cast | namespace | template | |

### 2.1.6 Data types

When programming, we store the variables in our computer's memory, but the computer must know what we want to store in them since storing a simple number, a letter or a large number is not going to occupy the same space in memory.

Our computer's memory is organized in bytes. A byte is the minimum amount of memory that we can manage. A byte can store a relatively small amount of data, usually an integer between 0 and 255 or one single character. But in addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or numbers with decimals. Next you have a list of the existing fundamental data types in C++, as well as the range of values that can be represented with each one of them:

| Name | Bytes* | Description | Range* |
|---|---|---|---|
| **Char** | 1 | character or integer 8 bits length. | **signed:** -128 to 127 **unsigned:** 0 to 255 |
| **Short** | 2 | integer 16 bits length. | **signed:** -32768 to 32767 **unsigned:** 0 to 65535 |
| **Long** | 4 | integer 32 bits length. | **signed:**-2147483648 to 2147483647 **unsigned:** 0 to 4294967295 |
| **Int** | * | Integer. Its length traditionally depends on the length of the system's Word **type**, thus in MSDOS it is 16 bits long, whereas in 32 bit systems (like Windows 9x/2000/NT and systems that work under protected mode in x86 systems) it is 32 bits long (4 bytes). | See **short**, **long** |
| **Float** | 4 | floating point number. | 3.4e + / - 38 (7 digits) |
| **Double** | 8 | double precision floating point number. | 1.7e + / - 308 (15 digits) |
| **long double** | 10 | long double precision floating point number. | 1.2e + / - 4932 (19 digits) |

| | | | |
|---|---|---|---|
| **Bool** | 1 | Boolean value. It can take one of two values: true or false NOTE: this is a type recently added by the ANSI-C++ standard. Not all compilers support it. Consult section bool type for compatibility information. | **true** or **false** |
| **wchar_t** | 2 | Wide character. It is designed as a type to store international characters of a two-byte character set. NOTE: this is a type recently added by the ANSI-C++ standard. Not all compilers support it. | wide characters |

* Values of columns Bytes and Range may vary depending on your system. The values included here are the most commonly accepted and used by almost all compilers. In addition to these fundamental data types, there also exist the pointers and the void parameter type specification that we will see later.

Primitive data types are data structures built into the C++ language. They include basic numeric data, such as integers and floating point numbers, as well as character and Boolean variables.

1. **Primitive data types:**
   ○ void: generic identifier that does not imply
   type ○ int: basic integer variable; 2 or 4 bytes ○
   float: basic floating point variable; 4 bytes
   ○ double: double precision floating point
   variable; 8 bytes ○ char: basic ASCII
   character variable; 1 byte ○ bool: basic
   Boolean variable; 1 byte
2. **Modifiers:** All modifiers default to int if a base type is not specified.
   ○ unsigned: does not use a sign bit
   ○ long: may have double number of bytes as base type; implementation dependent ○
   short: may have half the number of bytes as base type; implementation dependent
3. **Arrays:** Arrays are blocks of consecutive data structures. Arrays are declared with a specific size in the following way: int a[size];
   A specific part of an array is referenced using an index**,** which is an integer value from 0 to size - 1.
4. **Enumeration:** A data type that holds a set of values defined by the user. An enumeration can be named, and if so, the name is also the type. Members of an enumeration are integral types that have values associated with them. The user declares the values in a list enclosed by brackets:
   enum cards {CCLUBS, DIAMONDS, HEARTS, SPADES}
   The enumeration type is cards, and the integral types are assigned to the enumerator values as follows:
   0 = = CLUBS, 1 = = DIAMONDS, 2 = = HEARTS, 3 = = SPADES

A composite data type is any data type which can be constructed in a program using its programming language's primitive data types and other composite types. The act of constructing a composite type is known as composition.

### 2.1.6.1 Fundamental types

C++ provides the following fundamental built-in data types: Boolean, character, integer and floatingpoint. It also enables us to create our own user-defined data types using enumerations and classes. For each of the fundamental data types the range of values and the operations that can be performed on variables of that data type are determined by the compiler. Each compiler should provide the same operations for a particular data type but the range of values may vary between different compilers.

#### 2.1.6.1.1 Boolean Type

The Boolean type can have the value true or false. For example:

```
bool isEven = false;
bool keyFound = true;
```

If a Boolean value is converted to an integer value true becomes 1 and false becomes 0.If an integer value is converted to a Boolean value 0 becomes false and non-zero becomes true.

#### 2.1.6.1.2 Character Type

The character type is used to store characters - typically ASCII characters but not always. For example:

```
char menuSelection = 'q';  char
userInput = '3';
```

Note how a character is enclosed within single quotes. We can also assign numeric values to variables of character type:

```
char chNumber = 26;
```

We can declare signed and unsigned characters, where signed characters can have positive and negative values, and unsigned characters can only contain positive values.

```
signed char myChar = 100;  signed
char newChar = -43;  unsigned
char yourChar = 200;
```

Note that if we use a plain char, neither signed nor unsigned:

```
char dataValue = 27;
```

it may differ between compilers as to whether it behaves as a signed or unsigned character type. On some compilers it may accept positive and negative values; on others it may only accept positive values. Refer to your compiler documentation to see which applies.

A char is guaranteed to be at least 8 bits in size. C++ also provides the data type wchar_t, a wide character type typically used for large character sets.

An array of characters can be used to contain a C-style string in C++. For example:

```
char aString[] = "This is a C-style string";
```

Note that C++ also provides a string class that has advantages over the use of character arrays.

### 2.1.6.1.3 Integer Types

The integer type is used for storing whole numbers. We can use signed, unsigned or plain integer values as follows:

```
signed int index = 41982;
signed int temperature = -
32;  unsigned int count = 0;
int height = 100;
int balance = -67;
```

Like characters, signed integers can hold positive or negative values, and unsigned integers can hold only positive values. However, plain integer can always hold positive or negative values, they're always signed. You can declare signed and unsigned integer values in a shortened form, without the int keyword:

```
signed index = 41982;
unsigned count = 0;
```

Integer values come in three sizes, plain int, short int and long int.

```
int normal = 1000;   short
int smallValue = 100;
long int bigValue = 10000;
```

The range of values for these types will be defined by your compiler. Typically a plain int can hold a greater range than a short int, a long int can hold a greater range than a plain int, although this may not always be true. What we can be sure of is that plain int will be at least as big as short int and may be greater, and long int will be at least as big as plain int and may be greater. A short integer is guaranteed to be at least 16 bits and a long integer at least 32 bits.
You can declare short and long integer values in a shortened form, without the int keyword:

```
short smallValue = 100;
long bigValue = 10000;
```

You can have long and short signed and unsigned integers, for example:

```
unsigned long bigPositiveValue = 12345;  signed
short smallSignedValue= -7;
```

### 2.1.6.1.4 Floating-Point Types

Floating point types can contain decimal numbers, for example 1.23, -.087. There are three sizes, float (single-precision), double (double-precision) and long double (extended-precision). Some examples:

```
float celsius = 37.623;  double
fahrenheit = 98.415;
long double accountBalance = 1897.23;
```

The range of values that can be stored in each of these is defined by your compiler. Typically double will hold a greater range than float and long double will hold a greater range than double but this may not always be true. However, we can be sure that double will be at least as great as float and may be greater, and long double will be at least as great as double and may be greater.

### 2.1.6.2 Enumeration Type

An enumeration type is a user defined type that enables the user to define the range of values for the type. Named constants are used to represent the values of an enumeration, for example:

```
enum weekday {monday, tuesday, wednesday, thursday, friday, saturday, sunday};
weekday currentDay = wednesday;  if(currentDay==tuesday)
{
// do something
}
```

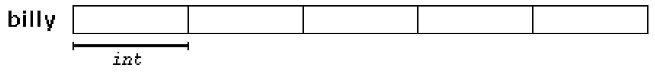
The default values assigned to the enumeration constants are zero-based, so in our example above monday == 0, tuesday == 1, and so on.

The user can assign a different value to one or more of the enumeration constants, and subsequent values that are not assigned a value will be incremented. For example: enum fruit {apple=3, banana=7, orange, kiwi}; Here, orange will have the value 8 and kiwi 9.

### 2.1.6.3 Arrays

Arrays are a series of elements (variables) of the same type placed consecutively in memory that can be individually referenced by adding an index to a unique name. That means that, for example, we can store 5 values of type int without having to declare 5 different variables each with a different identifier. Instead, using an array we can store 5 different values of the same type, int for example, with a unique identifier.

For example, an array to contain 5 integer values of type int called billy could be represented this way:



where each blank panel represents an element of the array, that in this case are integer values of type int. These are numbered from 0 to 4 since in arrays the first index is always 0, independently of its length. Like any other variable, an array must be declared before it is used. A typical declaration for an array in C++ is:

```
type name [elements];
```

where type is a valid object type (int, float...), name is a valid variable identifier and the elements field, that is enclosed within brackets [], specifies how many of these elements the array contains. Therefore, to declare billy as shown above it is as simple as the following sentence:

```
int billy [5];
```

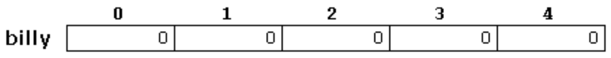
The elements field within brackets [] when declaring an array must be a constant value, since arrays are blocks of static memory of a given size and the compiler must be able to determine exactly how much memory it must assign to the array before any instruction is considered.

### 2.1.6.3.1 Initialising Arrays

When declaring an array of local scope (within a function), if we do not specify otherwise, it will not be initialized, so its content is undetermined until we store some values in it. If we declare a global array (outside any function) its content will be initialized with all its elements filled with zeros. Thus, if in the global scope we declare:

```
int billy [5];
```

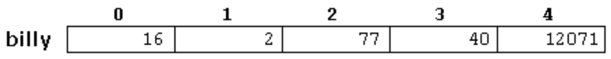
every element of billy will be set initialy to 0:



But additionally, when we declare an Array, we have the possibility to assign initial values to each one of its elements using curly brackets { }. For example:

```
int billy [5] = { 16, 2, 77, 40, 12071 };
```

this declaration would have created an array like the following one:



The number of elements in the array that we initialized within curly brackets { } must match the length in elements that we declared for the array enclosed within square brackets [ ]. For example, in the example of the billy array we have declared that it had 5 elements and in the list of initial values within curly brackets { } we have set 5 different values, one for each element.

Because this can be considered useless repetition, C++ includes the possibility of leaving the brackets empty [ ] and the size of the Array will be defined by the number of values included between curly brackets { }:

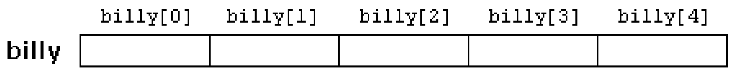
```
int billy [] = { 16, 2, 77, 40, 12071 };
```

### 2.1.6.3.2 Access to the values of an Array

In any point of the program in which the array is visible we can access individually anyone of its values for reading or modifying as if it was a normal variable. The format is the following:

name[index]

Following the previous examples in which billy had 5 elements and each of those elements was of type int, the name which we can use to refer to each element is the following:



For example, to store the value 75 in the third element of billy a suitable sentence would be:
billy[2] = 75;

and, for example, to pass the value of the third element of billy to the variable a, we could write:

a = billy[2];

Therefore, for all purposes, the expression billy[2] is like any other variable of type int. Notice that the third element of billy is specified billy[2], since first is billy[0], the second is billy[1], and therefore, third is billy[2]. By this same reason, its last element is billy[4]. Since if we wrote billy[5], we would be acceding to the sixth element of billy and therefore exceeding the size of the array.
In C++ it is perfectly valid to exceed the valid range of indices for an Array, which can create problems since they do not cause compilation errors but they can cause unexpected results or serious errors during execution. The reason why this is allowed will be seen farther ahead when we begin to use pointers.

At this point it is important to be able to clearly distinguish between the two uses that brackets[ ] have related to arrays. They perform two different tasks: one is to set the size of arrays when declaring them; and second is to specify indices for a concrete array element when referring to it. We must simply take care not to confuse these two possible uses of brackets [ ] with arrays:

```
        int billy[5];       // declaration of a new Array (begins with a type name)
billy[2] = 75;      // access to an element of the Array.
```

Other valid operations with arrays:

```
billy[0] = a;
billy[a] = 75;
b = billy
[a+2];
billy[billy[a]] = billy[2] + 5;
```

```
// arrays example
#include <iostream.h>

int billy [] = {16, 2, 77, 40, 12071}; int
n, result=0;

int main ()
```

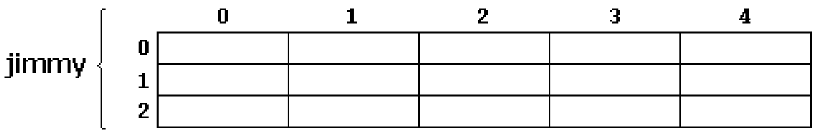
```
{
        for ( n=0 ; n<5 ; n++ )
        {
                result += billy[n];
        }
      cout      <<      result;
return 0;
}
```

Output

```
12206
```

#### 2.1.6.3.3 Multidimensional Arrays

Multidimensional arrays can be described as arrays of arrays. For example, a bi-dimensional array can be imagined as a bi-dimensional table of a uniform concrete data type.

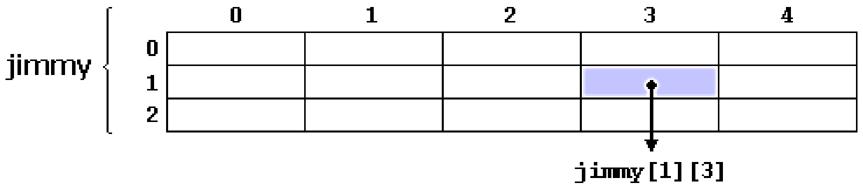


jimmy represents a bi-dimensional array of 3 per 5 values of type int. The way to declare this array would be:

```
int jimmy [3][5];
```

and, for example, the way to reference the second element vertically and fourth horizontally in an expression would be:

```
jimmy[1][3]
```



Remember that array indices always begin by 0. Multidimensional arrays are not limited to two indices (two dimensions). They can contain as many indices as needed, although it is rare to have to represent more than 3 dimensions. Just consider the amount of memory that an array with many indices may need. For example:

```
char century [100][365][24][60][60];
```

assigns a char for each second contained in a century, that is more than 3 billion chars! This would consume about 3000 megabytes of RAM memory if we could declare it.

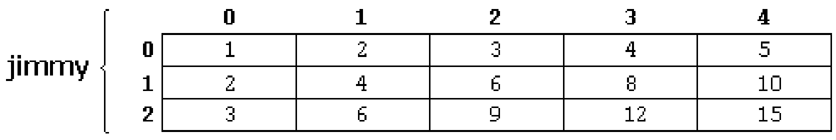
Multidimensional arrays are nothing more than an abstraction, since we can obtain the same results with a simple array just by putting a factor between its indices:

```
int jimmy [3][5];   is equivalent to int
jimmy [15];   (3 * 5 = 15)
```

with the only difference that the compiler remembers for us the depth of each imaginary dimension. Serve as example these two pieces of code, with exactly the same result, one using bi-dimensional arrays and the other using only simple arrays:

```
// multidimensional array
#include <iostream.h>

#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT][WIDTH];
int n,m;

int main ()
{
        for (n=0;n<HEIGHT;n++)
                for (m=0;m<WIDTH;m++)
                {
                        jimmy[n][m]=(n+1)*(m+1);
                }
                return 0;
}
// pseudo-multidimensional array
#include <iostream.h>

#define WIDTH 5
#define HEIGHT 3

int jimmy [HEIGHT * WIDTH];
int n,m;

int main ()
{
        for (n=0;n<HEIGHT;n++)      for
        (m=0;m<WIDTH;m++)
                {
                        jimmy[n * WIDTH + m]=(n+1)*(m+1);
                }
                return 0;
}
```

none of the programs above produce any output on the screen, but both assign values to the memory block called jimmy in the following way:



| jimmy | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 4 | 6 | 8 | 10 |
| 2 | 3 | 6 | 9 | 12 | 15 |

We have used defined constants (#define) to simplify possible future modifications of the program, for example, in case that we decided to enlarge the array to a height of 4 instead of 3 it could be done by changing the line:

```
#define HEIGHT 3
```

to

```
#define HEIGHT 4
```

with no need to make any other modifications to the program.

### 2.1.6.3.4 Arrays as Parameters

At some moment we may need to pass an array to a function as a parameter. In C++ is not possible to pass by value a complete block of memory as a parameter to a function, even if it is ordered as an array, but it is allowed to pass its address. This has almost the same practical effect and it is a much faster and more efficient operation.

In order to admit arrays as parameters the only thing that we must do when declaring the function is to specify in the argument the base type for the array, an identifier and a pair of void brackets []. For example, the following function:

```
void procedure (int arg[])
```

admits a parameter of type "Array of int" called arg. In order to pass to this function an array declared as:

```
int myarray [40];
```

it would be enough to write a call like this:

```
procedure (myarray);
```

Here you have a complete example:

```
// arrays as parameters
#include <iostream.h>

void printarray (int arg[], int length) {
        for (int n=0; n<length; n++)
```

```
                    cout << arg[n] << " ";  cout <<
"\n";
}

int main ()
{
          int firstarray[] = {5, 10, 15};    int
secondarray[] = {2, 4, 6, 8, 10};
printarray (firstarray,3);
          printarray              (secondarray,5);
return 0;
}
```

Output

```
5 10 15
2 4 6 8 10
```

As you can see, the first argument (int arg[]) admits any array of type int, wathever its length is. For that reason we have included a second parameter that tells the function the length of each array that we pass to it as the first parameter. This allows the for loop that prints out the array to know the range to check in the passed array.

In a function declaration is also possible to include multidimensional arrays. The format for a tridimensional array is:

```
base_type[][depth][depth]
```

for example, a function with a multidimensional array as argument could be:

```
void procedure (int myarray[][3][4])
```

notice that the first brackets [] are void and the following ones are not. This must always be thus because the compiler must be able to determine within the function which is the depth of each additional dimension.

Arrays, either simple or multidimensional, passed as function parameters are a quite common source of errors for less experienced programmers.

### 2.1.6.3.5 Pointers and arrays

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, like a pointer is equivalent to the address of the first element that it points to, so in fact they are the same thing. For example, supposing these two declarations:

```
int numbers [20];
```

```
                    int * p;
```
the following allocation would be valid:

```
                    p = numbers;
```

At this point p and numbers are equivalent and they have the same properties, the only difference is that we could assign another value to the pointer p whereas numbers will always point to the first of the 20 integer numbers of type int with which it was defined. So, unlike p, that is an ordinary variable pointer, numbers is a constant pointer (indeed an array name is a constant pointer). Therefore, although the previous expression was valid, the following allocation is not:

```
 numbers = p;
```

because numbers is an array (constant pointer), and no values can be assigned to constant identifiers. Due to the character of variables all the expressions that include pointers in the following example are perfectly valid:

```
// more pointers
#include <iostream.h>

int main ()
{
          int numbers[5];
int * p;            p =
numbers;        *p = 10;
p++;
           *p = 20;          p    =
&numbers[2];
             *p = 30;
             p = numbers + 3;
          *p = 40;
p = numbers;
*(p+4) = 50;
          for (int n=0; n<5; n++)
cout << numbers[n] << ", ";
             return 0;
}
```

Output

```
10, 20, 30, 40, 50,
```

In chapter "Arrays" we used bracket signs [] several times in order to specify the index of the element of the Array to which we wanted to refer. Well, the bracket signs operator [] are known as offset operators and they are equivalent to adding the number within brackets to the address of a pointer. For example, both following expressions:

```
a[5] = 0;           // a [offset of 5] = 0
*(a+5) = 0;         // pointed by (a+5) = 0
```

are equivalent and valid either if a is a pointer or if it is an array. More on pointers in later chapters.

### 2.1.6.4 Strings of Characters

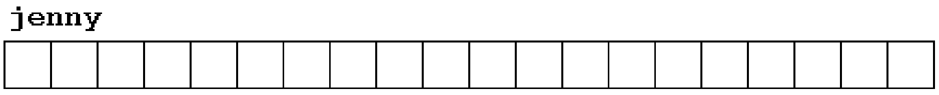
In all programs seen until now, we have used only numerical variables, used to express numbers exclusively. But in addition to numerical variables there also exist strings of characters, that allow us to represent successions of characters, like words, sentences, names, texts, etcetera. Until now we have only used them as constants, but we have never considered variables able to contain them.

In C++ there is no specific elemental variable type to store strings of characters. In order to fulfil this feature we can use arrays of type char, which are successions of char elements. Remember that this data type (char) is the one used to store a single character, for that reason arrays of them are generally used to make strings of single characters.

For example, the following array (or string of characters):

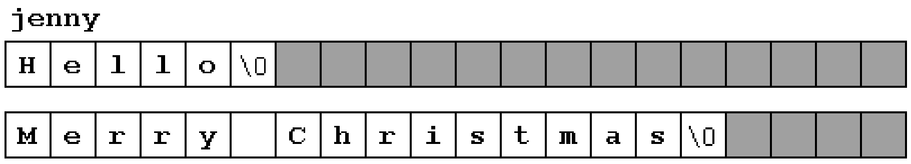
```
char jenny [20];
```

can store a string up to 20 characters long. You may imagine it thus:



This maximum size of 20 characters is not required to always be fully used. For example, jenny could store at some moment in a program either the string of characters "Hello" or the string "Merry Christmas". Therefore, since the array of characters can store shorter strings than its total length, a convention has been reached to end the valid content of a string with a null character, whose constant can be written 0 or '\0'.

We could represent jenny (an array of 20 elements of type char) storing the strings of characters "Hello" and "Merry Christmas" in the following way:



Notice how after the valid content a null character ('\0') it is included in order to indicate the end of the string. The panels in grey colour represent indeterminate values.

#### 2.1.6.4.1 Initialization of strings

Because strings of characters are ordinary arrays they fulfil all their same rules. For example, if we want to initialize a string of characters with predetermined values we can do it just like any other array:

```
char mystring[] = { 'H', 'e', 'l', 'l', 'o', '\0' };
```

In this case we would have declared a string of characters (array) of 6 elements of type char initialized with the characters that compose Hello plus a null character '\0'. Nevertheless, strings of characters have an additional way to initialize their values: using constant strings.

In the expressions we have used in examples in previous chapters constants that represented entire strings of characters have already appeared several times. These are specified enclosed between double quotes ("), for example:

```
"the result is: "
```

is a constant string that we have probably used on some occasion.

Unlike single quotes (') which specify single character constants, double quotes (") are constants that specify a succession of characters. Strings enclosed between double quotes always have a null character ('\0') automatically appended at the end.

Therefore we could initialize the string mystring with values by either of these two ways:

```
char mystring [] = { 'H', 'e', 'l', 'l', 'o', '\0' }; char
mystring [] = "Hello";
```

In both cases the array or string of characters mystring is declared with a size of 6 characters (elements of type char): the 5 characters that compose Hello plus a final null character ('\0') which specifies the end of the string and that, in the second case, when using double quotes (") it is automatically appended.

Before going further, notice that the assignation of multiple constants like double-quoted constants (") to arrays are only valid when initializing the array, that is, at the moment when declared. Expressions within the code like:

```
mystring = "Hello";
mystring[] = "Hello";
```

are not valid for arrays, like neither would be: mystring
= { 'H', 'e', 'l', 'l', 'o', '\0' };

So remember: We can "assign" a multiple constant to an Array only at the moment of initializing it. The reason will be more comprehensible when you know a bit more about pointers, since then it will be clarified that an array is simply a constant pointer pointing to an allocated block of memory. And because of this constantan, the array itself cannot be assigned any value, but we can assign values to each of the elements of the array.

The moment of initializing an Array it is a special case, since it is not an assignation, although the same equal sign (=) is used. Anyway, always have the rule previously underlined present.

### 2.1.6.4.2 Assigning values to strings

Since the lvalue of an assignation can only be an element of an array and not the entire array, it would be valid to assign a string of characters to an array of char using a method like this:

```
mystring[0] = 'H';
mystring[1] = 'e';
mystring[2] = 'l';
mystring[3] = 'l';
mystring[4] = 'o';
mystring[5] = '\0';
```

But as you may think, this does not seem to be a very practical method. Generally for assigning values to an array, and more specifically to a string of characters, a series of functions like strcpy are used. strcpy (string copy) is defined in the cstring (string.h) library and can be called the following way:
strcpy (string1, string2);

This does copy the content of string2 into string1. string2 can be either an array, a pointer, or a constant string, so the following line would be a valid way to assign the constant string "Hello" to mystring:

```
strcpy (mystring, "Hello");
```

For example:

```
// setting value to string
#include <iostream.h>
#include <string.h>

int main ()
{
        char szMyName [20];
strcpy (szMyName,"J. Soulie");
cout << szMyName;
          return 0;
}
```

Output

```
J. Soulie
```

Notice that we needed to include <string.h> header in order to be able to use function strcpy. Although we can always write a simple function like the following setstring with the same operation as cstring's strcpy:

```
// setting value to string
#include <iostream.h>

void setstring (char szOut [], char szIn [])
```

```
{
            int n=0;
            do
            {
                        szOut[n] = szIn[n];
            }
            while (szIn[n++] != '\0');
}

int main ()
{
            char szMyName [20];
          setstring      (szMyName,"J.      Soulie");
cout << szMyName;
            return 0;
}
```

Output

```
J. Soulie
```

Another frequently used method to assign values to an array is by directly using the input stream (cin). In this case the value of the string is assigned by the user during program execution. When cin is used with strings of characters it is usually used with its getline method, that can be called following this prototype:
cin.getline (char buffer[], int length, char delimiter = ' \n');

where buffer is the address of where to store the input (like an array, for example), length is the maximum length of the buffer (the size of the array) and delimiter is the character used to determine the end of the user input, which by default - if we do not include that parameter - will be the newline character ('\n').

The following example repeats whatever you type on your keyboard. It is quite simple but serves as an example of how you can use cin.getline with strings:

```
// cin with strings
#include <iostream.h>
int main ()
{
          char mybuffer [100];   cout << "What's your
name? ";         cin.getline (mybuffer,100);     cout
<< "Hello " << mybuffer << ".\n";       cout <<
"Which is your favourite team? ";      cin.getline
(mybuffer,100);
            cout << "I like " << mybuffer << " too.\n";
            return 0;
}
```

Output

```
What's your name? Juan Hello
Juan.
Which is your favourite team? Inter Milan I
like Inter Milan too.
```

Notice how in both calls to cin.getline we used the same string identifier (mybuffer). What the program does in the second call is simply step on the previous content of buffer with the new one that is introduced.

If you remember the section about communication through the console, you will remember that we used the extraction operator (>>) to receive data directly from the standard input. This method can also be used instead of cin.getline with strings of characters. For example, in our program, when we requested an input from the user we could have written:

```
cin >> mybuffer;
```

this would work, but this method has the following limitations that cin.getline has not:

- It can only receive single words (no complete sentences) since this method uses as a delimiter any occurrence of a blank character, including spaces, tabulators, newlines and carriage returns.
- It is not allowed to specify a size for the buffer. That makes your program unstable in case the user input is longer than the array that will host it.

For these reasons it is recommended that whenever you require strings of characters coming from cin you use cin.getline instead of cin >>.

#### 2.1.6.4.3 Converting strings to other types

Due to that a string may contain representations of other data types like numbers; it might be useful to translate that content to a variable of a numeric type. For example, a string may contain "1977", but this is a sequence of 5 chars not so easily convertible to a single integer data type. The cstdlib (stdlib.h) library provides three useful functions for this purpose:

- atoi: converts string to int type.
- atol: converts string to long type.
- atof: converts string to float type.

All of these functions admit one parameter and return a value of the requested type (int, long or float). These functions combined with getline method of cin are a more reliable way to get the user input when requesting a number than the classic cin>> method:

```
// cin and ato* functions
#include <iostream.h>
#include <stdlib.h>

int main ()
```

```
{
          char mybuffer [100];   float
price;   int quantity;    cout <<
"Enter price: ";          cin.getline
(mybuffer,100);           price = atof
(mybuffer);     cout << "Enter
quantity: ";     cin.getline
(mybuffer,100);           quantity =
atoi (mybuffer);          cout <<
"Total price: " << price*quantity;
            return 0;
}
```

Output

```
Enter price: 2.75
Enter quantity: 21
Total price: 57.75
```

### 2.1.6.4.4 Functions to manipulate strings

The cstring library (string.h) defines many functions to perform manipulation operations with C-like strings (like already explained strcpy). Here you have a brief look at the most usual:

strcat: char* strcat (char* dest, const char* src);
Appends src string at the end of dest string. Returns dest.

strcmp: int strcmp (const char* string1, const char* string2);
Compares strings string1 and string2. Returns 0 is both strings are equal.

strcpy: char* strcpy (char* dest, const char* src); Copies
the content of src to dest. Returns dest.

strlen: size_t strlen (const char* string); Returns
the length of string.

NOTE: char* is the same as char[]

## 2.1.7 Variables

It is a location in the computer memory which can store data and is given a symbolic name for easy reference. The variables can be used to hold different values at different times during the execution of a program.

To understand more clearly we should study the following statements:

```
Total = 20.00;
```

In this statement a value 20.00 has been stored in a memory location Total.

**2.1.7.1 Declaration of a variable**

Before a variable is used in a program, we must declare it. This activity enables the compiler to make available the appropriate type of location in the memory.

```
float Total;
```

You can declare more than one variable of same type in a single statement

```
int x,y;
```

**2.1.7.2 Initialization of variable**

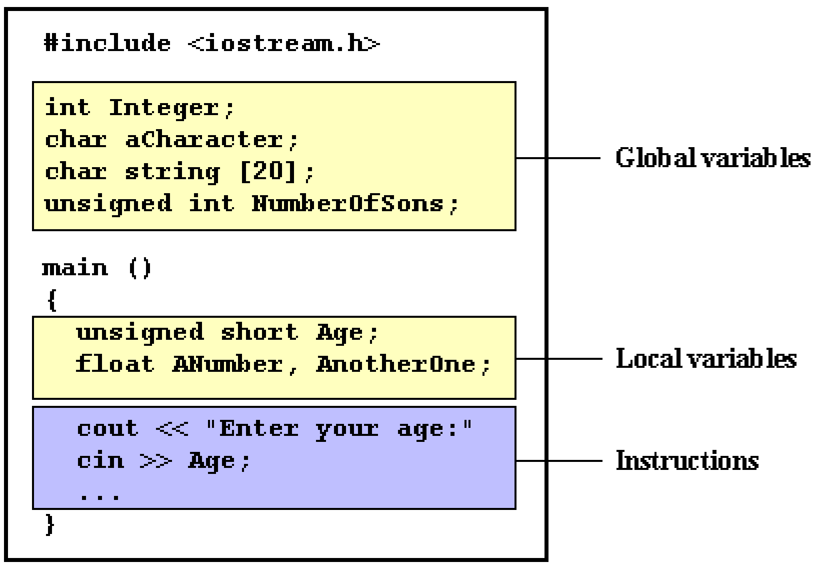
When we declare a variable its default value is undetermined. We can declare a variable with some initial value.

```
int a = 20;
```

**2.1.7.3 Scope of Variables**

All the variables that we are going to use must have been previously declared. An important difference between the C and C++ languages, is that in C++ we can declare variables anywhere in the source code, even between two executable sentences, and not only at the beginning of a block of instructions, like happens in C.

Anyway, it is recommended under some circumstances to follow the indications of the C language when declaring variables, since it can be useful when debugging a program to have all the declarations grouped together. Therefore, the traditional C-like way to declare variables is to include their declaration at the beginning of each function (for local variables) or directly in the body of the program outside any function (for global variables).



```
#include <iostream.h>

int Integer;
char aCharacter;
char string [20];
unsigned int NumberOfSons;

main ()
{
  unsigned short Age;
  float ANumber, AnotherOne;

  cout << "Enter your age:"
  cin >> Age;
  ...
}
```

**Global variables** can be referred to anywhere in the code, within any function, whenever it is after its declaration.
The scope of the **local variables** is limited to the code level in which they are declared. If they are declared at the beginning of a function (like in **main**) their scope is the whole **main** function. In the example above, this means that if another function existed in addition to main(), the local variables declared in **main** could not be used in the other function and vice versa.
In C++, the scope of a local variable is given by the block in which it is declared (a block is a group of instructions grouped together within curly brackets **{}** signs). If it is declared within a function it will be a variable with function scope, if it is declared in a loop its scope will be only the loop, etc...
In addition to **local** and **global** scopes there exists external scope that causes a variable to be visible not only in the same source file but in all other files that will be linked together.

### 2.1.8 Constants: Literals.

A constant is any expression that has a fixed value. They can be divided in Integer Numbers, Floating-Point Numbers, Characters and Strings.

#### 2.1.8.1 Integer Numbers

```
1776
707
-273
```

they are numerical constants that identify integer decimal numbers. Notice that to express a numerical constant we do not need to write quotes (**"**) nor any special character. There is no doubt that it is a constant: whenever we write **1776** in a program we will be referring to the value 1776.

In addition to decimal numbers (those that all of us already know) C++ allows the use as literal constants of octal numbers (base 8) and hexadecimal numbers (base 16). If we want to express an octal number we must precede it with a **0** character (zero character). And to express a hexadecimal number we have to precede it with the characters **0x** (zero, x). For example, the following literal constants are all equivalent to each other:

```
75      // decimal
0113     // octal
0x4b     // hexadecimal
```

All of them represent the same number: 75 (seventy five) expressed as a radix-10 number, octal and hexadecimal, respectively.
[Note: You can find more information on hexadecimal and octal representations in the document Numerical radixes]

### 2.1.8.2 Floating Point Numbers

They express numbers with decimals and/or exponents. They can include a decimal point, an **e** character (that expresses "by ten at the Xth height", where X is the following integer value) or both.

```
3.14159      // 3.14159
6.02e23      // 6.02 x 10^23
1.6e-19      // 1.6 x 10^-19
3.0          // 3.0
```

These are four valid numbers with decimals expressed in C++. The first number is PI, the second one is the number of Avogadro, the third is the electric charge of an electron (an extremely small number) -all of them approximated- and the last one is the number 3 expressed as a floating point numeric literal.

### 2.1.8.3 Characters and Strings

There also exist non-numerical constants, like:

```
'z'
'p'
"Hello world"
"How do you do?"
```

The first two expressions represent single characters, and the following two represent strings of several characters. Notice that to represent a single character we enclose it between single quotes (') and to express a string of more than one character we enclose them between double quotes (").

When writing both single characters and strings of characters in a constant way, it is necessary to put the quotation marks to distinguish them from possible variable identifiers or reserved words. Notice this:

```
x
'x'
```

**x** refers to variable **x**, whereas **'x'** refers to the character constant **'x'**.

Character constants and string constants have certain peculiarities, like the **escape codes**. These are special characters that cannot be expressed otherwise in the source code of a program, like newline (\n) or tab (\t). All of them are preceded by an inverted slash (\). Here you have a list of such escape codes:

| | |
|---|---|
| **\n** | Newline |
| **\r** | carriage return |

| | |
|---|---|
| **\t** | Tabulation |
| **\v** | vertical tabulation |
| **\b** | Backspace |
| **\f** | page feed |
| **\a** | alert (beep) |
| **\'** | single quotes (') |
| **\"** | double quotes (") |
| **\?** | question (?) |
| **\** | inverted slash (\) |

For example:

```
'\n'
'\t'
"Left \t Right"
"one\ntwo\nthree"
```

Additionally, you can express any character by its numerical ASCII code by writing an inverted slash bar character (\) followed by the ASCII code expressed as an octal (radix-8) or hexadecimal (radix-16) number. In the first case (octal) the number must immediately follow the inverted slash (for example **\23** or **\40**), in the second case (hexadecimal), you must put an **x** character before the number.

For example **\x20** or **\x4A**.

Constants of string of characters can be extended by more than a single code line if each code line ends with an inverted slash (\):

```
"string expressed in \ two
lines"
```

You can also concatenate several string constants separating them by one or several blank spaces, tabulators, newline or any other valid blank character:

```
"we form" "a single" "string" "of characters"
```

**2.1.8.4 Defined constants (#define)**

You can define your own names for constants that you use quite often without having to resort to variables, simply by using the **#define** preprocessor directive. This is its format:

```
#define identifier value
```

For example:

```
#define PI 3.14159265
#define NEWLINE '\n'
#define WIDTH 100
```

They define three new constants. Once they are declared, you are able to use them in the rest of the code as any if they were any other constant, for example:

```
circle = 2 * PI * r;
cout << NEWLINE;
```

In fact the only thing that the compiler does when it finds **#define** directives is to replace literally any occurrence of the them (in the previous example, **PI**, **NEWLINE** or **WIDTH**) by the code to which they have been defined (**3.14159265**, **'\n'** and **100**, respectively). For this reason, #define constants are considered macro constants.

The #define directive is not a code instruction, it is a directive for the preprocessor, therefore it assumes the whole line as the directive and does not require a semicolon (;) at the end of it. If you include a semicolon character (;) at the end, it will also be added when the preprocessor will substitute any occurence of the defined constant within the body of the program.

#### 2.1.8.5 declared constants (const)

With the **const** prefix you can declare constants with a specific type exactly as you would do with a variable: const int width = 100; const char tab = '\t';

```
const zip = 12440;
```

In case that the type was not specified (as in the last example) the compiler assumes that it is type **int**.

### 2.1.9 Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provides following type of operators:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators

This chapter will examine the arithmetic, relational, and logical, bitwise, assignment and other operators one by one.

#### 2.1.9.1 Arithmetic Operators:

There are following arithmetic operators supported by C++ language:

Assume variable A holds 10 and variable B holds 20 then:

| Operator | Description | Example |
|---|---|---|
| + | Adds two operands | A + B will give 30 |
| - | Subtracts second operand from the first | A - B will give -10 |
| * | Multiply both operands | A * B will give 200 |
| / | Divide numerator by de-numerator | B / A will give 2 |
| % | Modulus Operator and remainder of after an integer division | B % A will give 0 |
| ++ | Increment operator, increases integer value by one | A++ will give 11 |
| -- | Decrement operator, decreases integer value by one | A-- will give 9 |

**2.1.9.2 Relational Operators:**

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20 then:

| Operator | Description | Example |
|---|---|---|
| == | Checks if the value of two operands is equal or not, if yes then condition becomes true. | (A == B) is not true. |
| != | Checks if the value of two operands is equal or not, if values are not equal then condition becomes true. | (A != B) is true. |
| > | Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true. | (A > B) is not true. |
| < | Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true. | (A < B) is true. |

| | | |
|---|---|---|
| >= | Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true. | (A >= B) is not true. |
| <= | Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true. | (A <= B) is true. |

### 2.1.9.3 Logical Operators:

There are following logical operators supported by C++ language

Assume variable A holds 1 and variable B holds 0 then:

| Operator | Description | Example |
|---|---|---|
| && | Called Logical AND operator. If both the operands are non zero then condition becomes true. | (A && B) is false. |
| \|\| | Called Logical OR Operator. If any of the two operands is non zero then condition becomes true. | (A \|\| B) is true. |
| ! | Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false. | !(A && B) is true. |

### 2.1.9.4 Bitwise Operators:

Bitwise operator works on bits and performs bit by bit operation. The truth tables for &, \|, and ^ are as follows:

| p | q | p & q | p \| q | p ^ q |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 |

Assume if A = 60; and B = 13; Now in binary format they will be as follows:

A = 0011 1100
B = 0000 1101
A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13 then:

| Operator | Description | Example |
|---|---|---|
| & | Binary AND Operator copies a bit to the result if it exists in both operands. | (A & B) will give 12 which is 0000 1100 |
| \| | Binary OR Operator copies a bit if it exists in either operand. | (A \| B) will give 61 which is 0011 1101 |
| ^ | Binary XOR Operator copies the bit if it is set in one operand but not both. | (A ^ B) will give 49 which is 0011 0001 |
| ~ | Binary Ones Complement Operator is unary and has the effect of 'flipping' bits. | (~A ) will give -60 which is 1100 0011 |
| << | Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand. | A << 2 will give 240 which is 1111 0000 |
| >> | Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand. | A >> 2 will give 15 which is 0000 1111 |

**2.1.9.5 Assignment Operators:**

There are following assignment operators supported by C++ language:

| Operator | Description | Example |
|---|---|---|
| = | Simple assignment operator, Assigns values from right side operands to left side operand | C = A + B will assign value of A + B into C |
| += | Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand | C += A is equivalent to C = C + A |

| | | |
|---|---|---|
| -= | Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand | C -= A is equivalent to C = C - A |
| *= | Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand | C *= A is equivalent to C = C * A |
| /= | Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand | C /= A is equivalent to C = C / A |
| %= | Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand | C %= A is equivalent to C = C % A |
| <<= | Left shift AND assignment operator | C <<= 2 is same as C = C << 2 |
| >>= | Right shift AND assignment operator | C >>= 2 is same as C = C >> 2 |
| &= | Bitwise AND assignment operator | C &= 2 is same as C = C & 2 |
| ^= | bitwise exclusive OR and assignment operator | C ^= 2 is same as C = C ^ 2 |
| \|= | bitwise inclusive OR and assignment operator | C \|= 2 is same as C = C \| 2 |

### 2.1.9.6 Misc Operators

There are few other operators supported by C++ Language.

| **Operator** | **Description** |
|---|---|
| Sizeof | sizeof operator returns the size of a variable. For example sizeof(a), where a is integer, will return 4. |
| Condition ? X : Y | Conditional operator. If Condition is true ? then it returns value X : otherwise value Y |
| , | Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list. |
| . (dot) and -> (arrow) | Member operators are used to reference individual members of classes, structures, and unions. |

| | |
|---|---|
| Cast | Casting operators convert one data type to another. For example, int(2.2000) would return 2. |
| & | Pointer operator & returns the address of an variable. For example &a; will give actual address of the variable. |
| * | Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var. |

**2.1.9.7 Operators Precedence in C++:**

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:

For example x = 7 + 3 * 2; Here x is assigned 13, not 20 because operator * has higher precedence than + so it first get multiplied with 3*2 and then adds into 7.

Here operators with the highest precedence appear at the top of the table; those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

| **Category** | **Operator** | **Associativity** |
|---|---|---|
| Postfix | () [] -> . ++ - - | Left to right |
| Unary | + - ! ~ ++ - - (type)* & sizeof | Right to left |
| Multiplicative | * / % | Left to right |
| Additive | + - | Left to right |
| Shift | << >> | Left to right |
| Relational | < <= > >= | Left to right |
| Equality | == != | Left to right |
| Bitwise AND | & | Left to right |
| Bitwise XOR | ^ | Left to right |
| Bitwise OR | \| | Left to right |
| Logical AND | && | Left to right |

|  |  |  |
|---|---|---|
| Logical OR | \|\| | Left to right |
| Conditional | ?: | Right to left |
| Assignment | = += -= *= /= %=>>= <<= &= ^= \|= | Right to left |
| Comma | , | Left to right |

**2.1.10 Manipulators**

**What is a Manipulator?**

Manipulators are operators used in C++ for formatting output. The data is manipulated by the programmer's choice of display.

There are numerous manipulators available in C++. Some of the more commonly used manipulators are provided here below:

**endl Manipulator:**

This manipulator has the same functionality as the '\n' newline character.

**For example:**

Sample Code

```
cout << "Exforsys" << endl;
cout << "Training";
```

produces the output:

```
Exforsys
Training
```

**setw Manipulator:**

This manipulator sets the minimum field width on output. The syntax is:
setw(x)

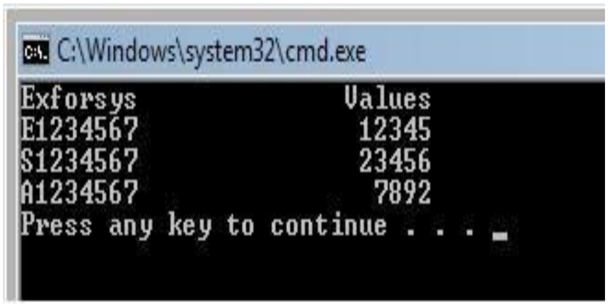
Here setw causes the number or string that follows it to be printed within a field of x characters wide and x is the argument set in setw manipulator. The header file that must be included while using setw manipulator is:

Sample Code

```
#include <iostream>
```

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

        void main( )
        {
        int x1=12345,x2= 23456, x3=7892;
        cout << setw(8) << "Exforsys" << setw(20) << "Values" << endl
        << setw(8) << "E1234567" << setw(20)<< x1 << endl
        << setw(8) << "S1234567" << setw(20)<< x2 << endl
        << setw(8) << "A1234567" << setw(20)<< x3 << endl;
        }
```

**The output of the above example is:**



C:\Windows\system32\cmd.exe

```
Exforsys                Values
E1234567                 12345
S1234567                 23456
A1234567                  7892
Press any key to continue . . .
```

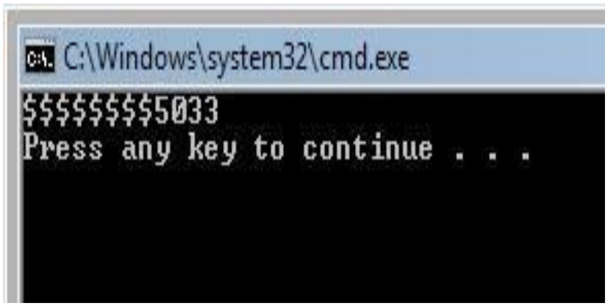
**setfill Manipulator:**

This is used after setw manipulator. If a value does not entirely fill a field, then the character specified in the setfill argument of the manipulator is used for filling the fields.

Sample Code

```
#include <iostream>
using namespace std;
#include <iomanip>
void main()
{
cout << setw(10) << setfill('$') << 50 << 33 << endl;
}
```

**The output of the above example is:**



C:\Windows\system32\cmd.exe

```
$$$$$$$$5033
Press any key to continue . . .
```

This is because the setw sets 10 for the width of the field and the number 50 has only 2 positions in it. So the remaining 8 positions are filled with $ symbol which is specified in the setfill argument.

**setprecision Manipulator:**

The setprecision Manipulator is used with floating point numbers. It is used to set the number of digits printed to the right of the decimal point. This may be used in two forms:

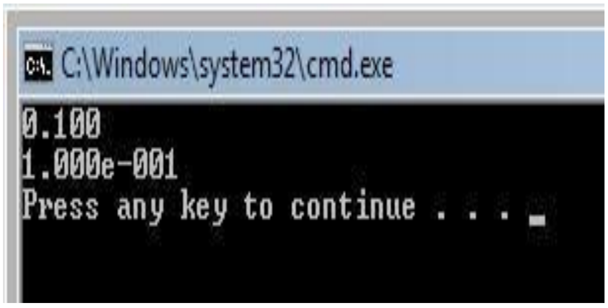
- fixed
- scientific

These two forms are used when the keywords fixed or scientific are appropriately used before the setprecision manipulator. The keyword fixed before the setprecision manipulator prints the floating point number in fixed notation. The keyword scientific, before the setprecision manipulator, prints the floating point number in scientific notation.

Sample Code

```
#include <iostream>
using namespace std;
#include <iomanip>
void main( )
{
float x = 0.1;
cout << fixed << setprecision(3) << x << endl;
cout << scientific << x << endl;
}
```

The output of the above example is:



```
C:\Windows\system32\cmd.exe
0.100
1.000e-001
Press any key to continue . . .
```

The first cout statement contains fixed notation and the setprecision contains argument 3. This means that three digits after the decimal point and in fixed notation will output the first cout statement as 0.100. The second cout produces the output in scientific notation. The default value is used since no setprecision value is provided.

**2.1.11 Expressions**

In this section, we now discuss arithmetic expressions in detail. Arithmetic expressions were introduced earlier in this chapter.

If all operands (that is, numbers) in an expression are integers, the expression is called an integral expression. If all operands in an expression are floating-point numbers, the expression is called a floatingpoint or decimal expression. An integral expression yields an integral result; a floating-point expression yields a floating-point result.

Looking at some examples will help clarify these definitions.

Consider the following C++ integral expressions:

```
2 + 3 * 5 3 +
x – y / 7
x + 2 * (y - z) + 18
```

In these expressions, x, y and z represent variables of the integer type; that is, they can hold integer values.

Consider the following C++ floating-point expressions:

```
12.8 * 17.5 - 34.50
x * 0.5 + y – 26.2
```

Here, x and y represent variables of the floating-point type; that is they can hold floating-point values.

Evaluating an integral or a floating point expression is straightforward. As before, when operators have the same precedence, the expression is evaluated from left to right. You can always use parentheses to group operands and operators to avoid confusion.

**Mixed Expressions**

An expression that has operands of different data types is called a mixed expression. A mixed expression contains both integers and floating-point numbers. The following expressions are examples of mixed expressions:

```
2 + 8.6
6 / 3 + 3.54
2.98 * 3 – 10.11 + 18 / 9
```

In the first expression, the operand + has one integer and one floating-point operand. In the second expression, both operands for the operator / are integers, the first operand of + is the result of 6 / 3, and the second operand of + is a floating-point number. The third example is an even more complicated mix of integers and floating-point numbers. The obvious question is: How does C++ evaluate mixed expressions?

Two rules apply when evaluating a mixed expression:

1. When evaluating an operator in a mixed expression:
    a. If the operator has the same type of operands (that is, either both integers or both floatingpoint numbers), the operator is evaluated according to the type of the operands. Integer operands thus yield an integer result; floating-point numbers yield a floating-point number.
    b. If the operator has both types of operands (that is, one id an integer and the other is a floatingpoint number), then during calculation the integer is changed to a floating-point

number with the decimal part of zero and the operator is evaluated. The result is a floating-point number.

2. The entire expression is evaluated according to the precedence rules; the multiplication, division, and modulus operators are evaluated before the addition and subtraction operators. Operators having the same level of precedence are evaluated from left to right. Grouping is allowed for clarity.

From these rules, it follows that when evaluating a mixed expression, you concentrate on one operator at a time, using the rules of precedence. If the operator to be evaluated has operands of the same type, evaluate the operator using rule 1(a). That is, an operator with integer operands will yield an integer result, and an operator with floating-point operands will yield a floating-point result. If the operator to be evaluated has one integer operand and one floating-point operand, before evaluating this operator convert the integer operand to a floating-point number with the decimal part of 0.

The following examples show how to evaluate mixed expressions.

| Mixed Expression | Evaluation | Rule Applied |
|---|---|---|
| 3 / 2 + 5.5 | = 1 + 5.5<br>= 6.5 | 3 / 2 = 1 (integer division; Rule 1(a))<br>(1 + 5.5<br>= 1.0 + 5.5 (Rule 1(b))<br>= 6.5) |
| 15.6 / 2 + 5 | = 7.8 + 5 | 15.6 / 2<br>= 15.6 / 2.0 (Rule 1(b))<br>= 7.8 |
| | = 12.8 | 7.8 + 5<br>= 7.8 + 5 (Rule 1(b))<br>= 12.8 |
| 4 + 5 / 2.0 | = 4 + 2.5 | 5 / 2.0 = 5.0 / 2.0 (Rule 1(b))<br>= 2.5 |
| | = 6.5 | 4 + 2.5 = 4.0 + 2.5 (Rule 1(b))<br>= 6.5 |
| 4 * 3 + 7 / 5 – 25.5 | = 12 + 7 / 5 – 25.5 | 4 * 3; (Rule 1(a)) |
| | = 12 + 1 – 25.5 | 7 / 5 = 1 (integer division; Rule 1(a)) |
| | = 13 – 25.5 | 12 + 1 = 13 (Rule 1(a)) |
| | = -12.5 | 13 – 25.5 = 13.0 – 25.5 (Rule 1(a))<br>= - 12.5 |

**2.1.12 Type Casting**

**Explicit type casting operators**

Type casting operators allows you to convert a datum of a given type to another. There are several ways to do this in C++, the most popular one, compatible with the C language is to precede the expression to be converted by the new type enclosed between parenthesis **()**:

```
int i; float f =
3.14;
i = (int) f;
```

The previous code converts the float number **3.14** to an integer value (**3**). Here, the type casting operator was **(int)**. Another way to do the same thing in C++ is using the constructor form: preceding the expression to be converted by the type and enclosing the expression between parentheses:

```
i = int (f);
```

Both ways of type casting are valid in C++. And additionally ANSI-C++ added new type casting operators more specific for object oriented programming.

**Implicit type casting conversion**

Data type can be mixed in the expression. For example

```
double a;  int
b = 5;  float c
= 8.5;  a = b
* c;
```

When two operands of different type are encountered in the same expression, the lower type variable is converted to the higher type variable. The following table shows the order of data types.

**Order of data types**

| **Data type** | **order** |
|---|---|
| long double<br>double<br>float long | (highest) |
| int<br>char | To<br><br>(lowest) |

The int value of b is converted to type float and stored in a temporary variable before being multiplied by the float variable c. The result is then converted to double so that it can be assigned to the double variable a.

### 2.1.13 Standard Input/ Output (I/O)

C++ supports input/output statements which can be used to feed new data into the computer or obtain output on an output device such as: VDU, printer etc. The following C++ stream objects can be used for the input/output purpose.

**cin** console input **cout**
console output

cout is used in conjunction with << operator, known as insertion or put to operator. cin is used in conjunction with >> operator, known as extraction or get from operator.

```
 cout << “My first computer";
```

Once the above statement is carried out by the computer, the message "My first computer" will appear on the screen.

cin can be used to input a value entered by the user from the keyboard. However, the get from operator>> is also required to get the typed value from cin and store it in the memory location.

Let us consider the following program segment:

```
  int marks;    cin >> marks;
```

In the above segment, the user has defined a variable marks of integer type in the first statement and in the second statement he is trying to read a value from the keyboard and store it in marks.

## 2.2 Control structures in C++

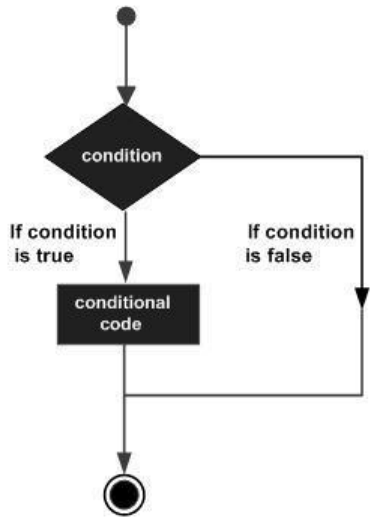
A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done to perform our program.
With the introduction of control sequences we are going to have to revisit the concept: the block of instructions. A block of instructions is a group of instructions separated by semicolons (;) but grouped in a block delimited by curly bracket signs: { and }.

Most of the control structures that we will see in this section allow a generic statement as a parameter; this refers to either a single instruction or a block of instructions, as we want. If we want the statement to be a single instruction we do not need to enclose it between curly-brackets ({}). If we want the statement to be more than a single instruction we must enclose them between curly brackets ({}) forming a block of instructions.

### 2.2.1 Conditional Structure (Selection)

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general from of a typical decision making structure found in most of the programming languages:



#### 2.2.1.1 if Statement

The if statement is a one-way decisions which means that it would either do some particular thing or do nothing at all. The decision is based on a logical expression which either evaluates to true or false, a value of 1 or a value of 0, respectively. Recall that a logical expression always evaluates to a value of true or false; in C++, a nonzero value is always true, and a value of zero is always false. If the logical expression is evaluated to true, then the corresponding statement is executed; if the logical expression evaluates to false, control goes to the next executable statement.

The form of a one-way decision statement is as follows:

```
if (logical expression)
{
        stmtT;
}
```

The stmtT can be a simple statement or compound statement. A simple statement involves a single statement. A compound statement involves one or more statements enclosed with curly braces { }. A compound statement is called a block statement.

Example 1: simple statement

```
int c = 3;

if ( c > 0 )
        cout << "c = " << c << endl;
```

Example 2: compound statement

```
int b = 5;
int c = 10;

if ( c < 5 )
{
        b = 2 * b + c;
        cout << c * c * b << endl;
}
```

**2.2.1.2 if…else Statement**

The if…else statement is a two-way decision making statement. Two-way decisions either do one particular thing or do another. Similar to one-way decisions, the decision is based on a logical expression.

If the expression is true, stmtT will be executed; if the expression is false, stmtF will be executed.

The form of a two-way decision statement is as follows:

```
if (logical expression)
{
        stmtT;
}
else
{
        stmtF;
}
```

stmtT and stmtF can be a simple statement or compound. Remember that compound statements are always enclosed with curly braces { }.

Example 1:

```
int c = 10;

if (c >= 0)
{
        cout << "c is a positive or neutral integer" << endl;
}
```

```
else
{
            cout << "c is a negative integer" << endl;
}
```

#### 2.2.1.3 if… else if Statement

The if…else if statement is a multi-way decision making statement. Multi-way decisions are used to evaluate a logical expression that could have several possible values. “if…else if” statements are often used to choose between ranges of values.

The form of a multi-way decision using an “if…else if” construct is as follows:

```
if ( logical expression )
{
            stmtT1;
}
else if ( logical expression )
{
            stmtT2;
}
else if ( logical expression )
{
            stmtT3;
}
else if ( logical expression )
{
            stmtTN;
}
else
{
            stmtF;
}
```

If the first logical expression is evaluated to true, then stmtT1 is executed. If the second logical expression is true, then stmtT2 is executed and so on down the line of logical expression tests. If none of the logical expressions are true, then the statement after else is executed which is stmtF.

#### 2.2.1.4 Nested if Statement

The nested if statement is also a multi-way decision making statement. The form of a multi-way decision using an “nested if” construct is as follows:

```
if ( conditionA )
{
            if ( conditionB )
```

```
		{
			stmtBT;
		}
		else
		{
			stmtBF;
		}
}
else
{
		stmtAF;
}
```

If conditionA is evaluated to true, then execution moves into the nested if and evaluates conditionB. If conditionA is evaluated to false, then stmtAF is executed.

Example 1:

```
int x = 500;

if ( x > 0 )
{
		cout << "x is positive" << endl;
}
else if ( x = 0 )
{
		cout << "x is zero" << endl;
}
else
{
		cout << "x is negative" << endl;
}
```

**2.2.1.5 switch case**

A switch statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

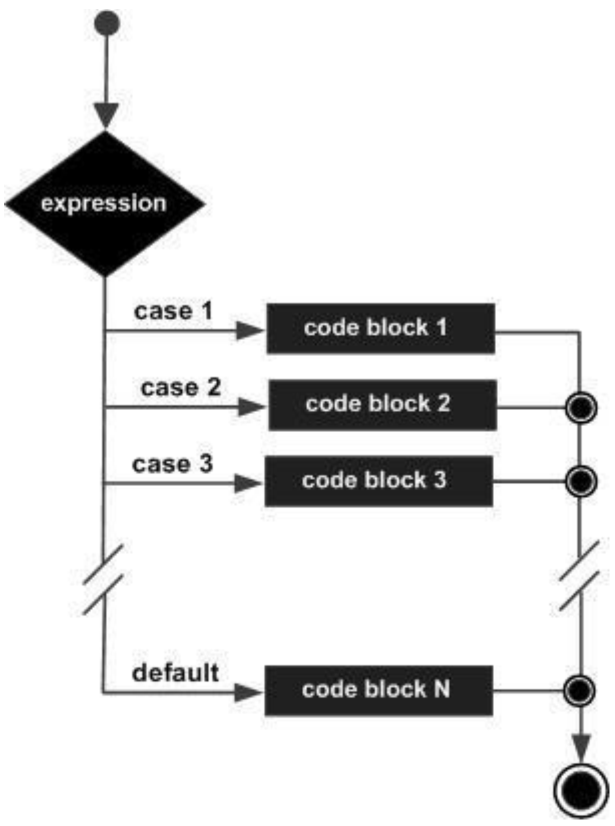
The syntax for a switch statement in C++ is as follows:

```
switch(expression)
{
   case constant-expression  :
      statement(s);          break;
//optional        case  constant-
expression  :
      statement(s);
      break; //optional

    // you can have any number of case statements.
    default : //Optional
      statement(s);
}
```

The following rules apply to a switch statement:

- The expression used in a switch statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The constant-expression for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a break statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will fall through to subsequent cases until a break is reached.
- A switch statement can have an optional default case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

**Flow Diagram:**



Example:

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

int main ()
{
                              // local variable declaration:   char
grade = 'D';

             switch(grade)
             {
                          case 'A' :
                                      cout << "Excellent!" << endl;
                              break;
          case 'B' :                      case
'C' :
                                      cout << "Well done" << endl;
                              break;
case 'D' :
                                      cout << "You passed" << endl;
                              break;
case 'F' :
                                      cout << "Better try again" << endl;
                              break;
default :
                                      cout << "Invalid grade" << endl;
```

```
        }
        cout << "Your grade is " << grade << endl;

        return 0;
}
```

This would produce following result:

```
You passed
Your grade is D
```

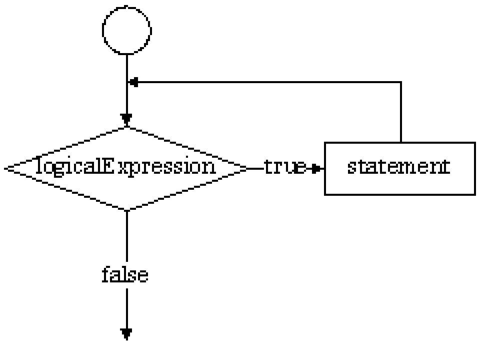
### 2.2.2 Loops (Repetition Structure)

There may be a situation when you need to execute a block of code several number of times. In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. Programming languages provide various control structures that allow for more complicated execution paths. A loop statement allows us to execute a statement or group of statements multiple times.

#### 2.2.2.1 The while Repetition Structure

The first looping structure is called a while loop.

```
while (logicalExpression)
{
        statement;
}
```



- In C++, while is a reserved word.
- The expression acts as a decision maker.  It is called the loop condition.
- The statement can be either a simple or compound statement.  It is called the body of the loop.

Explanations:
(Step 1)  If logicalExpression evaluates to **true**, the statement executes.
(Step 2) The logicalExpression is re-evaluated.  The body of the loop continues to execute until the logicalExpression is false.
Definition: Infinite loop - a loop that continues to execute endlessly is called an.
An infinite loop just never ends - loops and loops forever, because the stopping condition is never met.

Example: Consider the following loop (in pseudocode):

```
initialize counter to 0 repeat
until counter = 40
    statement1
    statement2
    add 3 to counter
end repeat
```

This is an unintentional infinite loop -- the counter will have values: 0, 3, 6, 9,..., 36, 39, 42, 45, etc. It will never be equal to 40, so the loop will continue executing forever.

To avoid an infinite loop, make sure that the loop's body contains statement(s) that assure that the expression will eventually be false.

There are four types of while loops:

Case 1: Counter-controlled while loop

- the exact number of pieces of data is known – for example it is N
- the counter is initialized to 0, then is evaluated to be <= N, then incremented in the body of the loop.

Example:

```
int i, sum;  sum
= 0;
i = 1;
while (i <= 100)
{
        sum = sum + i;
        i++;
}
cout << "The sum of the first 100 numbers is " << sum << endl;
```

Case 2: Sentinel-controlled while loop

- the exact number of pieces of data is not known, but the last entry is a special value, called a sentinel
- the first item is read before the while statement if it does not equal the sentinel, the body of the while statement executes  each additional item is read within the while statement.

Example:

```
int N, sum;
cout << "Enter the numbers to be added (for a sentinel enter 0):" << endl;
sum = 0;  cin >> N;
while (N != 0)
```

```
{
        sum = sum + N;
        cin >> N;
}
cout << "The sum of the entered numbers is " << sum << endl;
```

Case 3: Flag-controlled while loops

- uses a Boolean variable to control the loop
- the Boolean variable is set to either true or false
- while the expression evaluates to true, the loop statement continues  the Boolean variable must be set/reset within the while statement.

Case 4: EOF-controlled while loop

- the while statement executes until there are no more data items
- if the program has reached the end of the input data or enters into the fail state, the input stream variable returns false; in other cases, the input stream variable returns true
- the first item is read before the while statement and if the input stream variable is true, the while statement is entered; additional items are read within the while statement.

Example:

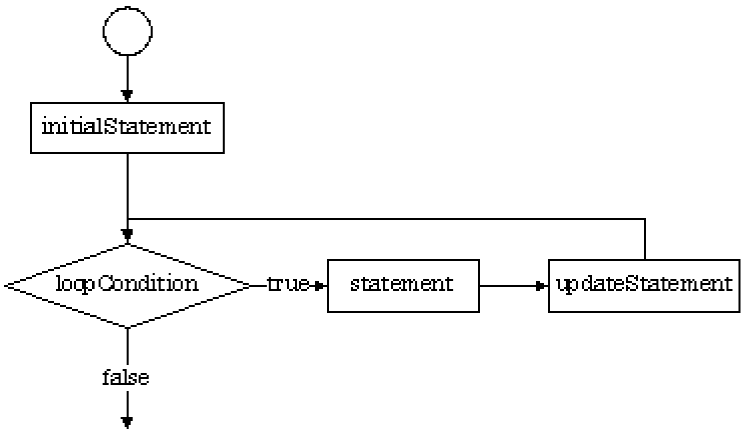
```
int N, sum;
cout << "Enter the numbers to be added (for end enter ctrl+z):" <<
endl;  sum = 0;  cin >> N;
while (cin)
{
        sum = sum + N;
        cin >> N;
}
cout << "The sum of the entered numbers is " << sum << endl;
```

**2.2.2.2 The for Repetition Structure**

C++ has a simplified count-control loop called a counted or indexed for loop.

```
for (initialStatement; loopCondition; updateStatement)
{
    statement;
}
```

Explanations:
(Step 1) The for loop begins with initialStatement
(Step 2) If the loopCondition is true, statement (the body of the loop) is executed
(Step 3) The updateStatement is executed
(Step 4) The cycle is repeated from (Step 2) until the loopCondition is false.



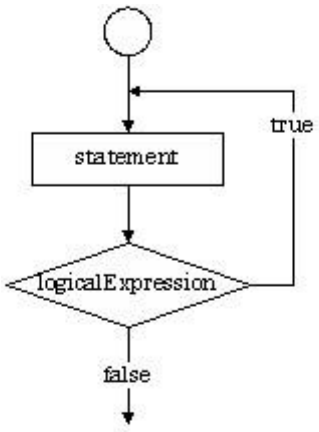
The for loop works when the update is either incremented or decremented. When decrementing, the initial value should be greater than the control value.

```
Example:   int
i, sum;  sum =
0;
for(i = 1; i <= N; i++)
        sum = sum + i;
```

**2.2.2.3 The do…while Repetition Structure**

```
do
{
    statement;
} while (logicalExpression);
```

The do…while loop is called a post-test loop as opposed to the while loop, which is a pre-test loop. A do…while loop will always be executed at least once since the condition test is at the end of the loop.



Example:

```
int i, sum;  sum
= 0;
i = 1;  do
{
        sum  =  sum  +  i;
        i++;
} while (i <= N);
```

Example: Using while, do-while, for statements to code same algorithm.

```
int i, sum;                          //i – counter, sum - accumulator
sum = 0;
i = 1;
while (i <= 100)
{
                sum = sum
+ i;            i++;
}

int i, sum;                          //i – counter, sum - accumulator
sum = 0;
i = 1;   do
{

    sum = sum + i;
    i++;
} while (i <= 100);

int i, sum;                          //i – counter, sum - accumulator        sum
= 0;
for (i = 1; I <= 100; i++)
        sum = sum + i;
```

### 2.2.2.4 break and continue Commands

The commands break; and continue; are useful for altering the flow of control. If a break command is executed in a while, do/while, for, or switch statement, then the execution immediately leaves that control structure. If a continue statement is executed in a while, do/while, or for structure, then the flow of control immediately skips the rest of the code in the loop. In the case of a while or do/while statement, the continue statement means that the program will execute the conditional test. In the case of a for loop, the increment statement is executed, after which the conditional test is checked.

### 2.2.2.5 The goto instruction.

It allows making an absolute jump to another point in the program. You should use this feature carefully since its execution ignores any type of nesting limitation.

The destination point is identified by a label, which is then used as an argument for the goto instruction. A label is made of a valid identifier followed by a colon (**:**).
This instruction does not have a concrete utility in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using **goto**:

```
// goto loop example #include
<iostream.h>
int main ()
{
    int     n=10;
loop:
      cout << n << ", ";
      n--;
      if (n>0) goto loop;
      cout << "FIRE!";
      return 0;
}
```

Output:

```
10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

## 2.3 Pointers

A pointer is a variable that is used to store a memory address. The address is the location of the variable in the memory. Pointers help in allocating memory dynamically. Pointers improve execution time and saves space. Pointer points to a particular data type. The general form of declaring pointer is:-

```
type *variable_name;
```

type is the base type of the pointer and variable_name is the name of the variable of the pointer. For example,

```
int *x;
```

x is the variable name and it is the pointer of type integer.

**Pointer Operators**

There are two important pointer operators such as ‘*’ and ‘&’. The ‘&’ is a unary operator. The unary operator returns the address of the memory where a variable is located. For example,

```
int x*;
int c;
x=&c;
```

variable x is the pointer of the type integer and it points to location of the variable c. When the statement

```
x=&c;
```

is executed, '&' operator returns the memory address of the variable c and as a result x will point to the memory location of variable c.

The '*' operator is called the indirection operator. It returns the contents of the memory location pointed to. The indirection operator is also called deference operator. For example,

```
 int x*;
int c=100;
        int p; x=&c;
        p=*x;
```

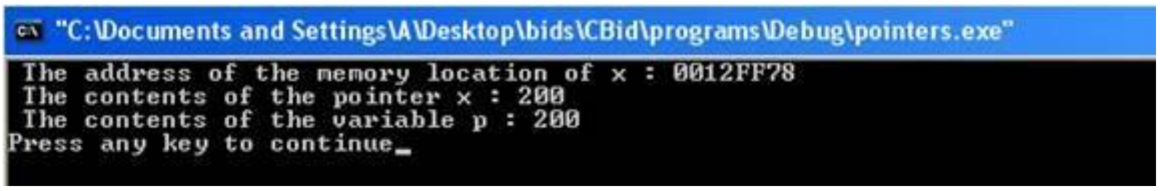
variable x is the pointer of integer type. It points to the address of the location of the variable c. The pointer x will contain the contents of the memory location of variable c. It will contain value 100. When statement

```
p=*x;
```

is executed, '*' operator returns the content of the pointer x and variable p will contain value 100 as the pointer x contain value 100 at its memory location. Here is a program which illustrates the working of pointers.

```
#include<iostream> using
namespace std;
int main ()
{          int *x;
int c=200;
int p;
x=&c;
p=*x;
          cout << " The address of the memory location of x : " << x << endl;
cout << " The contents of the pointer x : " << *x << endl;          cout << "
The contents of the variable p : " << p << endl;          return(0);
}
```

Output:-



```
"C:\Documents and Settings\A\Desktop\bids\CBid\programs\Debug\pointers.exe"
 The address of the memory location of x : 0012FF78
 The contents of the pointer x : 200
 The contents of the variable p : 200
Press any key to continue_
```

In the program variable x is the pointer of integer type. The statement

```
  x=&c;
```

points variable x to the memory location of variable c. The statement

```
            p=*x;
```

makes the contents of the variable p same as the contents of the variable c as x is pointing to the memory location of c. The statement

```
  cout << " The address of the memory location of x : " << x << endl;
```

prints the memory address of variable x which it is pointing to. It prints the hexadecimal address 0012FF78. This address will be different when the program is run on different computers. The statement

```
                    cout << " The contents of the pointer x : " << *x << endl;
```

prints the contents of memory location of the variable x which it is pointing to. The contents are same as the variable c which has value 200. The statement

```
  cout << " The contents of the variable p : " << p << endl;
```

has the same output 200 as the statement above. The contents of variable p is same as the contents of the pointer x.

## 2.4 <u>Functions</u>

Functions are building blocks of the programs. They make the programs more modular and easy to read and manage. All C++ programs must contain the function main( ). The execution of the program starts from the function main( ). A C++ program can contain any number of functions according to the needs. The general form of the function is: -

```
return_type  function_name(parameter list)
{
          body of the function

}
```

The function of consists of two parts function header and function body. The function header is:-

```
                    return_type   function_name(parameter list)
```

The return_type specifies the type of the data the function returns. The return_type can be void which means function does not return any data type. The function_name is the name of the function. The name of the function should begin with the alphabet or underscore. The parameter list consists of variables separated with comma along with their data types. The parameter list could be empty which means the function do not contain any parameters. The parameter list should contain both data type and name of the variable. For example,

```
                    int factorial(int n, float j)
```

is the function header of the function factorial. The return type is of integer which means function should return data of type integer. The parameter list contains two variables n and j of type integer and float respectively. The body of the function performs the computations.

### 2.4.1 Function Declaration

A function declaration is made by declaring the return type of the function, name of the function and the data types of the parameters of the function. A function declaration is same as the declaration of the variable. The function declaration is always terminated by the semicolon. A call to the function cannot be made unless it is declared. The general form of the declaration is:-

```
  return_type function_name(parameter list);
```

For example function declaration can be

```
                int factorial(int n1,float j1);
```

The variables name need not be same as the variables of parameter list of the function. Another method can be

```
                int factorial(int , float);
```

The variables in the function declaration can be optional but data types are necessary.

### 2.4.2 Function Arguments

The information is transferred to the function by the means of arguments when a call to a function is made. Arguments contain the actual value which is to be passed to the function when it is called. The sequence of the arguments in the call of the function should be same as the sequence of the parameters in the parameter list of the declaration of the function. The data types of the arguments should correspond with the data types of the parameters. When a function call is made arguments replace the parameters of the function.

### 2.4.3 Functions with Default Arguments

When declaring a function we can specify a default value for each parameter. This value will be used if that parameter is left blank when calling to the function. To do that we simply have to assign a value to the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is stepped on and the passed value is used. For example:

```
// default values in functions #include
<iostream.h>
int divide (int a, int b=2)
{
        int          r;
r=a/b;
          return (r);
}

int main ()
{
        cout << divide (12);
cout << endl;   cout << divide
(20,4);         return 0;
}
```

Output

```
6
5
```

As we can see in the body of the program there are two calls to the function divide. In the first one:

```
divide (12)
```

we have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we have specified to happen if this parameter is lacking (notice the function declaration, which finishes with int b=2). Therefore the result of this function call is 6 (12/2).

In the second call:

```
divide (20,4)
```

there are two parameters, so the default assignation (int b=2) is stepped on by the passed parameter, that is 4, making the result equal to 5 (20/4).

### 2.4.4 The Return Statement and Return values

A return statement is used to exit from the function where it is. It returns the execution of the program to the point where the function call was made. It returns a value to the calling code. The general form of the return statement is:-

```
return expression;
```

The expression evaluates to a value which has type same as the return type specified in the function declaration. For example the statement,

```
return(n);
```

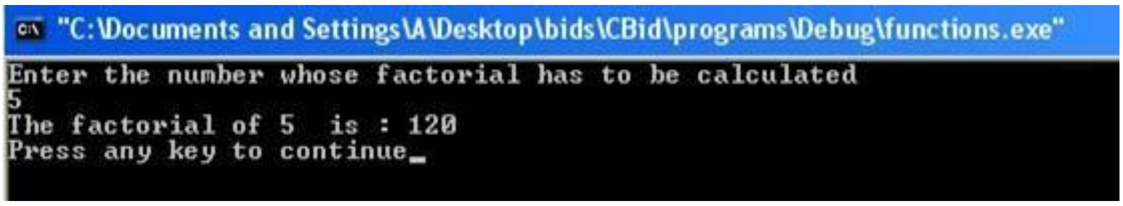
is the return statement of the factorial function. The type of variable n should be integer as specified in the declaration of the factorial function. If a function has return type as void then return statement does not contain any expression. It is written as:-

```
return;
```

The function with return type as void can ignore the return statement. The closing braces at the end indicate the exit of the function. Here is a program which illustrates the working of functions.

```
#include<iostream>
using namespace std;
int factorial(int n);

int main ()
{
        int n1,fact;
        cout <<"Enter the number whose factorial has to be calculated" << endl;
cin >> n1;          fact=factorial(n1);
        cout << "The factorial of " << n1 << "  is : " << fact << endl;
return(0);

}

int factorial(int n)
{
        int             i=0,fact=1;
if(n<=1)
        {
                return(1);
        }
else
        {
                for(i=1;i<=n;i++)
                {
                        fact=fact*i;
                }
                return(fact);
        }
}
```

Output:-



```
"C:\Documents and Settings\A\Desktop\bids\CBid\programs\Debug\functions.exe"
Enter the number whose factorial has to be calculated
5
The factorial of 5  is : 120
Press any key to continue_
```

The function factorial calculates the factorial of the number entered by the user. If the number is less than or equal to 1 then function returns 1 else it returns the factorial of the number. The statement

    int factorial(int n);

is a declaration of the function. The return type is of integer. The parameter list consists of one data type which is integer. The statement

```
        cout <<"Enter the number whose factorial has to be calculated" <<  endl;
    cin >> n1;
```

makes the user enter the number whose factorial is to be calculated. The variable n1 stores the number entered by the user. The user has entered number 5. The statement

```
        fact = factorial(n1);
```

makes a call to the function. The variable n1 is now argument to the function factorial. The argument is mapped to the parameters in the parameter list of the function. The function header is

```
        int factorial(int n)
```

The body of the function contains two return statements. If the value entered by the user is less than and equal to 1 then value 1 is returned else computed factorial is returned. The type of the expression returned is integer.

### 2.4.5 Function Prototype

The programmer may provide the remainder of a C++ source file, or module, the extended name (the name and functions) during the definition of the function. A function may be defined anywhere in the module. (A module is another name for a C++ source file.) However, something has to tell **main()** the full name of the function before it can be called. Consider the following code snippet:

```
int main(int argc, char* pArgs[])
{
    someFunc(1, 2);
}

int someFunc(double arg1, int arg2)
{
    // ...do something
}
```

The call to someFunc() from within main() doesn't know the full name of the function. It may surmise from the arguments that the name is someFunc(int, int) and that its return type is void; however, as you can see, this is incorrect. C++ could be less lazy and look ahead to determine the full name of someFunc()s on its own, but it doesn't. What is needed is some way to inform main() of the full name of someFunc() before it is used. What is needed is a before use function declaration. Some type of prototype is necessary. A prototype declaration appears the same as a function with no body. In use, a prototype declaration appears as follows:

```
int someFunc(double, int);              //function prototype  int
main(int argc, char* pArgs[])
{
        someFunc(1, 2);
}

int someFunc(double arg1, int arg2)
{
                                        // ...do something
}
```

The prototype declaration tells the world (at least that part of the world after the declaration), that the extended name for someFunc() is someFunction(double, int). The call in main() now knows to cast the 1 to a double before making the call. In addition, main() knows that the value returned by someFunc() is an int.

### 2.4.6 Functions with no types. The use of void.

If you remember the syntax of a function declaration:
 type name ( argument1, argument2 ...) statement

you will see that it is obligatory that this declaration begins with a type, that is the type of the data that will be returned by the function with the return instruction. But what if we want to return no value? Imagine that we want to make a function just to show a message on the screen. We do not need it to return any value, moreover, we do not need it to receive any parameters. For these cases, the void type was devised in the C language. Take a look at:

```
// void function example #include
<iostream.h>
void dummyfunction (void)
{
        cout << "I'm a function!";
}

int main ()
{
        dummyfunction ();
        return 0;
}
```

Output

```
I'm a function!
```

Although in C++ it is not necessary to specify void, its use is considered suitable to signify that it is a function without parameters or arguments and not something else. What you must always be aware of is that the format for calling a function includes specifing its name and enclosing the arguments between parenthesis. The non-existence of arguments does not exempt us from the obligation to use parenthesis. For that reason the call to dummyfunction is

```
                    dummyfunction ();
```

This clearly indicates that it is a call to a function and not the name of a variable or anything else.

### 2.4.7 Parameter passing mechanism

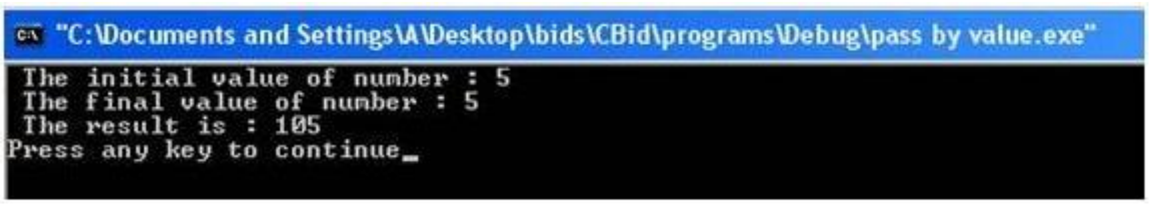
There are two parameter passing mechanisms for passing arguments to functions such as pass by value and pass by reference.

#### 2.4.7.1 Pass by value

In pass be value mechanism copies of the arguments are created and which are stored in the temporary locations of the memory. The parameters are mapped to the copies of the arguments created. The changes made to the parameter do not affect the arguments. Pass by value mechanism provides security to the calling program. Here is a program which illustrates the working of pass by value mechanism.

```
#include<iostream>
using namespace std; int
add(int n);

int main()
{
        int                 number,result;
number=5;
        cout   <<   "   The   initial   value   of   number   :   "   <<   number   <<   endl;
result=add(number);
        cout << " The final value of number : " << number << endl;
cout << " The result is : " << result << endl;          return(0);
}

int add(int number)
{
        number=number+100;
        return(number);
}
```

Output:-



```
"C:\Documents and Settings\A\Desktop\bids\CBid\programs\Debug\pass by value.exe"
 The initial value of number : 5
 The final value of number : 5
 The result is : 105
Press any key to continue_
```

The value of the variable number before calling the function is 5. The function call is made and function adds 100 to the parameter number. When the function is returned the result contains the added value.

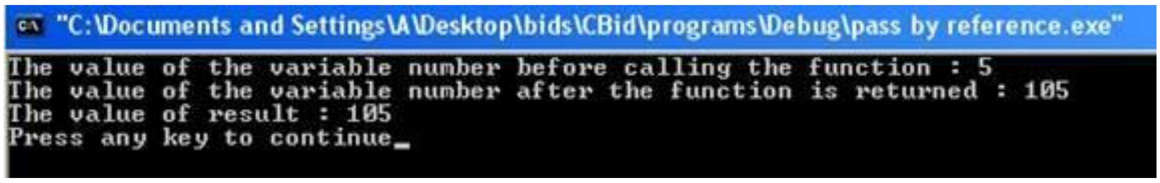
The final value of the number remains same as 5. This shows that operation on parameter does not produce effect on arguments.

### 2.4.7.2 Pass by reference

Pass by reference is the second way of passing parameters to the function. The address of the argument is copied into the parameter. The changes made to the parameter affect the arguments. The address of the argument is passed to the function and function modifies the values of the arguments in the calling function. Here is a program which illustrates the working of pass by reference mechanism.

```
 #include<iostream> using
namespace std;
int add(int &number);

int main ()
{
        int number;
int result;
number=5;
        cout << "The value of the variable number before calling the function : " << number << endl;
result=add(&number);
        cout << "The value of the variable number after the function is returned : " << number << endl;
cout << "The value of result : " << result << endl;        return(0);
}

int add(int &p)
{
        *p=*p+100;
return(*p);
}
```

Output:-



```
"C:\Documents and Settings\A\Desktop\bids\CBid\programs\Debug\pass by reference.exe"
The value of the variable number before calling the function : 5
The value of the variable number after the function is returned : 105
The value of result : 105
Press any key to continue_
```

The address of the variable is passed to the function. The variable p points to the memory address of the variable number. The value is incremented by 100. It changes the actual contents of the variable number. The value of variable number before calling the function is 100 and after the function is returned the value of variable number is changed to 105.

### 2.4.8 inline functions

The inline directive can be included before a function declaration to specify that the function must be compiled as code at the same point where it is called. This is equivalent to declaring a macro. Its advantage is only appreciated in very short functions, in which the resulting code from compiling the program may be faster if the overhead of calling a function (stacking of arguments) is avoided.

The format for its declaration is:

```
                    inline type name ( arguments ... ) {
            instructions ... }
```

and the call is just like the call to any other function. It is not necessary to include the inline keyword before each call, only in the declaration.

Following is an example which makes use of inline function to returns max of two numbers:

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

inline int Max(int x, int y)
{
        return (x > y)? x : y;
}

// Main function for the program
int main( )
{
      cout << "Max (20,10): " << Max(20,10) << endl;
cout << "Max (0,200): " << Max(0,200) << endl;
        cout << "Max (100,1010): " << Max(100,1010) << endl;
        return 0;
}
```

Output

```
Max (20,10): 20
Max (0,200): 200
Max (100,1010): 1010
```

### 2.4.9 Function Overloading

C++ permits the use of two or more functions with the same name. However such functions essentially have different argument list. The difference can be in terms of number or type of arguments or both.

This process of using two or more functions with the same name but differing in the signature is called function overloading. You overload a function name f by declaring more than one function with the name f in the same scope. The declarations of f must differ from each other by the types and/or the number of arguments in the argument list. When you call an overloaded function named f, the correct function is selected by comparing the argument list of the function call with the parameter list of each of the overloaded candidate functions with the name f. A candidate function is a function that can be called based on the context of the call of the overloaded function name. Overloading of functions with different return types is not allowed.

Consider a function print, which displays an int. As shown in the following example, you can overload the function print to display other types, for example, double and char*. You can have three functions with the same name, each performing a similar operation on a different data type:

```
#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");
}
```

The following is the output of the above example:

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

## 2.5 Dynamic Memory

Until now, in all our programs, we have only had as much memory available as we declared for our variables, having the size of all of them to be determined in the source code, before the execution of the program. But, what if we need a variable amount of memory that can only be determined during runtime? For example, in the case that we need some user input to determine the necessary amount of memory space. The answer is dynamic memory, for which C++ integrates the operators new and delete.

### 2.5.1 Operators new and new[]

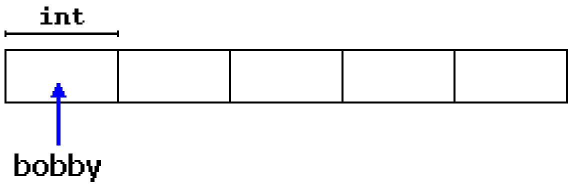
In order to request dynamic memory we use the operator new. new is followed by a data type specifier and -if a sequence of more than one element is required- the number of these within brackets []. It returns a pointer to the beginning of the new block of memory allocated. Its form is:

```
pointer = new type
pointer = new type [number_of_elements]
```

The first expression is used to allocate memory to contain one single element of type type. The second one is used to assign a block (an array) of elements of type type, where number_of_elements is an integer value representing the amount of these. For example:

```
int * bobby; bobby =
new int [5];
```

In this case, the system dynamically assigns space for five elements of type int and returns a pointer to the first element of the sequence, which is assigned to bobby. Therefore, now, bobby points to a valid block of memory with space for five elements of type int.



The first element pointed by bobby can be accessed either with the expression bobby[0] or the expression *bobby. Both are equivalent as has been explained in the section about pointers. The second element can be accessed either with bobby[1] or *(bobby+1) and so on...You could be wondering the difference between declaring a normal array and assigning dynamic memory to a pointer, as we have just done. The most important difference is that the size of an array has to be a constant value, which limits its size to what we decide at the moment of designing the program, before its execution, whereas the dynamic memory allocation allows us to assign memory during the execution of the program (runtime) using any variable or constant value as its size.
The dynamic memory requested by our program is allocated by the system from the memory heap. However, computer memory is a limited resource, and it can be exhausted. Therefore, it is important to have some mechanism to check if our request to allocate memory was successful or not. C++ provides two standard methods to check if the allocation was successful:
One is by handling exceptions. Using this method an exception of type bad_alloc is thrown when the allocation fails. Exceptions are a powerful C++ feature explained later in these tutorials. But for now you should know that if this exception is thrown and it is not handled by a specific handler, the program execution is terminated. This exception method is the default method used by new, and is the one used in a declaration like:

```
 bobby = new int [5];   // if it fails an exception is thrown
```

The other method is known as nothrow, and what happens when it is used is that when a memory allocation fails, instead of throwing a bad_alloc exception or terminating the program, the pointer returned by new is a null pointer, and the program continues its execution. This method can be specified by using a special object called nothrow, declared in header <new>, as argument for new:

```
 bobby = new (nothrow) int [5];
```

In this case, if the allocation of this block of memory failed, the failure could be detected by checking if bobby took a null pointer value:

```
int * bobby;
bobby = new (nothrow) int [5];
if (bobby == 0)
{
     // error assigning memory. Take measures.
};
```

This nothrow method requires more work than the exception method, since the value returned has to be checked after each and every memory allocation, but I will use it in our examples due to its simplicity. Anyway this method can become tedious for larger projects, where the exception method is generally preferred. The exception method will be explained in detail later in this tutorial.

### 2.5.2 Operators delete and delete[]

Since the necessity of dynamic memory is usually limited to specific moments within a program, once it is no longer needed it should be freed so that the memory becomes available again for other requests of dynamic memory. This is the purpose of the operator delete, whose format is:

```
delete pointer;
delete [] pointer;
```

The first expression should be used to delete memory allocated for a single element, and the second one for memory allocated for arrays of elements. The value passed as argument to delete must be either a pointer to a memory block previously allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

```
// rememb-o-matic
#include <iostream> #include
<new>
using namespace std;

int main ()
{
        int i,n;
        int * p;
      cout << "How many numbers would you like to type? ";
cin >> i;
        p = new (nothrow) int[i];
```

```
            if (p == 0)
                  cout << "Error: memory could not be allocated";      else
            {
                        for (n=0; n<i; n++)
                        {
                            cout     <<     "Enter     number:     ";
                  cin >> p[n];
                        }
                  cout << "You have entered: ";
          for (n=0; n<i; n++)
cout << p[n] << ", ";
                        delete[] p;
            }
            return 0;
}
```

Output

```
How many numbers would you like to type? 5
Enter number : 75
Enter number : 436
Enter number : 1067
Enter number : 8
Enter number : 32
You have entered: 75, 436, 1067, 8, 32,
```

Notice how the value within brackets in the new statement is a variable value entered by the user (i), not a constant value:

```
                  p= new (nothrow) int[i];
```

But the user could have entered a value for i so big that our system could not handle it. For example, when I tried to give a value of 1 billion to the "How many numbers" question, my system could not allocate that much memory for the program and I got the text message we prepared for this case (Error: memory could not be allocated). Remember that in the case that we tried to allocate the memory without specifying the nothrow parameter in the new expression, an exception would be thrown, which if it's not handled terminates the program. It is a good practice to always check if a dynamic memory block was successfully allocated. Therefore, if you use the nothrow method, you should always check the value of the pointer returned. Otherwise, use the exception method, even if you do not handle the exception. This way, the program will terminate at that point without causing the unexpected results of continuing executing a code that assumes a block of memory to have been allocated when in fact it has not.

# TOPIC 3

# 3. CLASSES AND OBJECTS

**LEARNING OUTCOMES**

***After Studying this topic you should be able to:***

- Create Classes
- Understand the software engineering concepts of encapsulation and data hiding
- Understand Constructors and Destructor
- Overloading Operators
- Understand the purpose of friend functions and friend classes
- Understand the use of the "this pointer"
- Understand how to convert objects from one class to another class a

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

## 3.1 C++ Class Definitions:

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword class followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example we defined the Box data type using the keyword class as follows:

```
class Box
{
        public:
```

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

An access specifier is one of the following three keywords: private, public or protected. These specifiers modify the access rights that the members following them acquire:

- private members of a class are accessible only from within other members of the same class or from their friends.
- protected members are accessible from members of their same class and from their friends, but also from members of their derived classes.
- Finally, public members are accessible from anywhere where the object is visible.

By default, all members of a class declared with the class keyword have private access for all its members.

## 3.2 Define C++ Objects:

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

```
                         Box Box1;        // Declare Box1 of type Box
                         Box Box2;        // Declare Box2 of type Box
```

Both of the objects Box1 and Box2 will have their own copy of data members.

## 3.3 Accessing the Data Members:

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try following example to make the things clear:

```
#include <iostream>

using namespace std;

class Box
{
          public:                    double length;                                   // Length
of a box                   double breadth;                        // Breadth of a box
double height;                                    // Height of a box
};

int main( )
{
          Box Box1;                                                 // Declare Box1 of type Box    Box Box2;
                              // Declare Box2 of type Box    double volume = 0.0;
          // Store the volume of a box here
```

```
                                                    // box 1 specification
      Box1.height = 5.0;
      Box1.length = 6.0;
      Box1.breadth = 7.0;
                                                    // box 2 specification
      Box2.height = 10.0;
      Box2.length = 12.0;
      Box2.breadth = 13.0;

                                                    // volume of box 1
     volume = Box1.height * Box1.length * Box1.breadth;
cout << "Volume of Box1 : " << volume <<endl;

                                                    // volume of box 2
      volume = Box2.height * Box2.length * Box2.breadth;
      cout << "Volume of Box2 : " << volume <<endl;
      return 0;
}
```

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

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

It is important to note that private and protected members cannot be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

**3.4 <u>Class Members</u>**

An optional member list declares sub-objects called class members. Class members can be data, functions, nested types, and enumerators.

Class member list syntax

```
   .--------------------------------------------------------------------------.
   V                                                                          |
>>---+-member_declaration--+-------------------------------+--;-+--+---><
     |                     +-=--0------------------------+     |
|                          '--=--constant_expression-'     |
     +-member_definition------------------------------------------+
     '-access_specifier--:--------------------------------------------'
```

The member list follows the class name and is placed between braces. The following applies to member lists, and members of member lists:

- A member_declaration or a member_definition may be a declaration or definition of a data member, member function, nested type, or enumeration. (The enumerators of an enumeration defined in a class member list are also members of the class.)
- A member list is the only place where you can declare class members.
- Friend declarations are not class members but must appear in member lists.
- The member list in a class definition declares all the members of a class; you cannot add members elsewhere.
- You cannot declare a member twice in a member list.
- You may declare a data member or member function as static but not auto, extern, or register.
- You may declare a nested class, a member class template, or a member function, and define it outside the class.
- You must define static data members outside the class.
- Non-static members that are class objects must be objects of previously defined classes; a class A cannot contain an object of class A, but it can contain a pointer or reference to an object of class A.
- You must specify all dimensions of a non-static array member.

A constant initializer (= constant_expression) may only appear in a class member of integral or enumeration type that has been declared static.
A pure specifier (= 0) indicates that a function has no definition. It is only used with member functions declared as virtual and replaces the function definition of a member function in the member list.

An access specifier is one of public, private, or protected.
A member declaration declares a class member for the class containing the declaration.

The order of allocation of non-static class members separated by an access_specifier is implementationdependent. The compiler allocates class members in the order in which they are declared.

Suppose A is a name of a class. The following class members of A must have a name different from A:

- All data members  All type members
- All enumerators of enumerated type members
- All members of all anonymous union members

### 3.4.1 Data members

Data members include members that are declared with any of the fundamental types, as well as other types, including pointer, reference, array types, bit fields, and user-defined types. You can declare a data member the same way as a variable, except that explicit initializers are not allowed inside the class definition. However, a const static data member of integral or enumeration type may have an explicit initializer.

If an array is declared as a non-static class member, you must specify all of the dimensions of the array.

A class can have members that are of a class type or are pointers or references to a class type. Members that are of a class type must be of a class type that has been previously declared. An incomplete class type can be used in a member declaration as long as the size of the class is not needed. For example, a member can be declared that is a pointer to an incomplete class type.

A class X cannot have a member that is of type X, but it can contain pointers to X, references to X, and static objects of X. Member functions of X can take arguments of type X and have a return type of X. For example:

```
class X
{
        X();
      X     *xptr;
X &xref;
        static X xcount;
        X xfunc(X);
};
```

### 3.4.2 Member functions

Member functions are operators and functions that are declared as members of a class. Member functions do not include operators and functions declared with the friend specifier. These are called friends of a class. You can declare a member function as static; this is called a static member function. A member function that is not declared as static is called a non-static member function.

The definition of a member function is within the scope of its enclosing class. The body of a member function is analyzed after the class declaration so that members of that class can be used in the member function body, even if the member function definition appears before the declaration of that member in the class member list. When the function add() is called in the following example, the data variables a, b, and c can be used in the body of add().

```
class x
{
        public:
                int add()         // inline member function add
            {return             a+b+c;};
private:
                int a,b,c;
};
```

### 3.4.3 Inline member functions

You may either define a member function inside its class definition, or you may define it outside if you have already declared (but not defined) the member function in the class definition.

A member function that is defined inside its class member list is called an inline member function. Member functions containing a few lines of code are usually declared inline. In the above example, add() is an inline member function. If you define a member function outside of its class definition, it must appear in a namespace scope enclosing the class definition. You must also qualify the member function name using the scope resolution (::) operator.

An equivalent way to declare an inline member function is to either declare it in the class with the inline keyword (and define the function outside of its class) or to define it outside of the class declaration using the inline keyword.

In the following example, member function Y::f() is an inline member function:

```
class Y
{
        private:
            char*           a;
public:
                char* f() { return a; }
};
```

The following example is equivalent to the previous example; Y::f() is an inline member function:

```
class Y
{
        private:
            char*           a;
public:
                char* f();
};
inline char* Y::f()
{
        return a;
}
```

The inline specifier does not affect the linkage of a member or nonmember function: linkage is external by default.

Member functions of a local class must be defined within their class definition. As a result, member functions of a local class are implicitly inline functions. These inline member functions have no linkage.

### 3.4.4 Pointers to members

Pointers to members allow you to refer to non-static members of class objects. You cannot use a pointer to member to point to a static class member because the address of a static member is not associated with any particular object. To point to a static class member, you must use a normal pointer.
You can use pointers to member functions in the same manner as pointers to functions. You can compare pointers to member functions, assign values to them, and use them to call member functions. Note that

a member function does not have the same type as a nonmember function that has the same number and type of arguments and the same return type.

There are two pointer to member operators: .* and ->*.

The .* operator is used to dereference pointers to class members. The first operand must be of class type. If the type of the first operand is class type T, or is a class that has been derived from class type T, the second operand must be a pointer to a member of a class type T.

The ->* operator is also used to dereference pointers to class members. The first operand must be a pointer to a class type. If the type of the first operand is a pointer to class type T, or is a pointer to a class derived from class type T, the second operand must be a pointer to a member of class type T.

The .* and ->* operators bind the second operand to the first, resulting in an object or function of the type specified by the second operand.
If the result of .* or ->* is a function, you can only use the result as the operand for the ( ) (function call) operator. If the second operand is an lvalue, the result of .* or ->* is an lvalue.

Pointers to members can be declared and used as shown in the following example:

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

class X
{
        public:
                int a;
                void f(int b)
                {
                        cout << "The value of b is "<< b << endl;
                 }
};

int main()
{
                                // declare pointer to data member    int
X::*ptiptr = &X::a;

                                // declare a pointer to member function      void
(X::* ptfptr) (int) = &X::f;

                                        // create an object of class type X
        X xobject;

                                        // initialize data member
        xobject.*ptiptr = 10;
        cout << "The value of a is " << xobject.*ptiptr << endl;
```

```
                                        // call member function
        (xobject.*ptfptr) (20);
}
```

The output for this example is:

```
The value of a is 10
The value of b is 20
```

To reduce complex syntax, you can declare a typedef to be a pointer to a member. A pointer to a member can be declared and used as shown in the following code fragment:

```
typedef int X::*my_pointer_to_member; typedef
void (X::*my_pointer_to_function) (int);

int main()
{
        my_pointer_to_member ptiptr = &X::a;
        my_pointer_to_function ptfptr = &X::f;
        X xobject;
        xobject.*ptiptr = 10;
        cout << "The value of a is " << xobject.*ptiptr << endl;
        (xobject.*ptfptr) (20);
}
```

The pointer to member operators .* and ->* are used to bind a pointer to a member of a specific class object. Because the precedence of () (function call operator) is higher than .* and ->*, you must use parentheses to call the function pointed to by ptf. Pointer-to-member conversion can occur when pointers to members are initialized, assigned, or compared. Note that pointer to a member is not the same as a pointer to an object or a pointer to a function.

### 3.4.5 The this pointer

The keyword this identifies a special type of pointer. Suppose that you create an object named x of class A, and class A has a non-static member function f(). If you call the function x.f(), the keyword this in the body of f() stores the address of x. You cannot declare the this pointer or make assignments to it. A static member function does not have a this pointer.

The type of the this pointer for a member function of a class type X, is X* const. If the member function is declared with the const qualifier, the type of the this pointer for that member function for class X, is const X* const.

A const this pointer can be used only with const member functions. Data members of the class will be constant within that function. The function is still able to change the value, but requires a const_cast to do so:

```
void foo::p() const
{
        member = 1;                          // illegal
        const_cast <int&> (member) = 1;      // a bad practice but legal
}
```

A better technique would be to declare member mutable.

If the member function is declared with the volatile qualifier, the type of the this pointer for that member function for class X is volatile X* const. For example, the compiler will not allow the following:

```
class A
{
        int a;
        int f() const { return a++; }
};
```

The compiler will not allow the statement a++ in the body of function f(). In the function f(), the this pointer is of type A* const. The function f() is trying to modify part of the object to which this points.

The this pointer is passed as a hidden argument to all non-static member function calls and is available as a local variable within the body of all non-static functions.

For example, you can refer to the particular class object that a member function is called for by using the this pointer in the body of the member function. The following code example produces the output a = 5:

```
#include <iostream>
using namespace std; class
X
{
        private:
             int a;    public:
      void Set_a(int a)
                {
                                        // The 'this' pointer is used to retrieve 'xobj.a'
                                // hidden by the automatic variable 'a'
this->a = a;
                }

                void Print_a()
                {
                        cout << "a = " << a << endl;
                }
};

int main()
{
        X xobj;
int a = 5;
xobj.Set_a(a);
            xobj.Print_a();
}
```

In the member function Set_a(), the statement this->a = a uses the this pointer to retrieve xobj.a hidden by the automatic variable a.

Unless a class member name is hidden, using the class member name is equivalent to using the class member name with the this pointer and the class member access operator (->).

The example in the first column of the following table shows code that uses class members without the this pointer. The code in the second column uses the variable THIS to simulate the first column's hidden use of the this pointer:

### 3.4.6 Static members

Class members can be declared using the storage class specifier static in the class member list. Only one copy of the static member is shared by all objects of a class in a program. When you declare an object of a class having a static member, the static member is not part of the class object.
A typical use of static members is for recording data common to all objects of a class. For example, you can use a static data member as a counter to store the number of objects of a particular class type that are created. Each time a new object is created; this static data member can be incremented to keep track of the total number of objects.

You access a static member by qualifying the class name using the :: (scope resolution) operator. In the following example, you can refer to the static member f() of class type X as X::f() even if no object of type X is ever declared:

```
class X
{
            static int f();
};

int main()
{
            X::f();
}
```

## 3.5 Constructors

When a class is instantiated, its constructor is called. When a constructor is called, special code is run that initializes the object's state. There are several different constructors in C++ that each serves a specific purpose.

Remember the following points in constructor

- There is no virtual constructor.
- Constructor cannot return any value.
- If you have to return any value from constructor, then use the exceptions.
- Constructors can be called implicitly or explicitly.
- They can be declared private, protected or public to restrict the access level.
- There can be any number of constructors but only one destructor
- There can be constructors with default parameters

### 3.5.1 Default Constructor

This constructor is called whenever a new object is instantiated and no arguments are passed in. This constructor is useful for creating a general-case where object values are initialized to a useful default value. The following is a default constructor to a class named "ExampleClass."

```
ExampleClass();
```

### 3.5.2 Constructor with Arguments

You can make a constructor accept arguments. These are typically used to initialize certain values to something other than their default. When an object is instantiated, arguments are passed into the constructor and these are then used to initialize the variables. The following is a constructor that accepts arguments.

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

### 3.5.3 Copy Constructor

A copy constructor is called whenever the assignment operator -- the equals sign -- is invoked on an object. This copies the data from one object into another. This is useful when a deep copy of data is necessary. A deep copy is one where the actual data is copied, not just references to the original data. If you do not include a copy constructor, one is generated by the compiler, and almost certainly produces shallow copies of data. This means that both objects contain references to the same data, and changing one piece of data will affect both objects. The following is what a copy constructor looks like. ExampleClass(const ExampleClass& copy);

### 3.5.4 Destructor

A destructor is the opposite of a constructor. It is called whenever an object is destroyed. This is a useful place to put clean-up code that returns any allocated memory to the system. For declaring a destructor we have to use the ~ tilde symbol in front of the destructor. The following is what a destructor looks like. ~ExampleClass();

```
// overloading class constructors
#include <iostream>
```

```
using namespace std;

class Line
{
        public:
                int getLength( void );
                Line();                                 // default constructor
                Line( int len );                        // constructor with parameter
                Line( const Line &obj);                 // copy constructor
                ~Line();                                // destructor

        private:
                int *ptr;
};

// Member functions definitions including constructor

Line::Line()
{
}

Line::Line(int len)
{
        cout << "Normal constructor allocating ptr" << endl;
                                        // allocate memory for the pointer;        ptr   =
new int;
        *ptr = len;
}
Line::Line(const Line &obj)
{
      cout  <<  "Copy  constructor  allocating  ptr."  <<  endl;
ptr = new int;
        *ptr = *obj.ptr;                                // copy the value
}

Line::~Line(void)
{
      cout    <<    "Freeing    memory!"    <<    endl;
delete ptr;
}

int Line::getLength( void )
{
        return *ptr;
}
```

```
void display(Line obj)
{
        cout << "Length of line : " << obj.getLength() <<endl;
}

// Main function for the program
int main( )
{
        Line line1(10);            // This calls the parameterised constructor
      Line line2 = line1;     // This calls the copy constructor
display(line1);        display(line2);
        return 0;
}
```

Output

```
Normal constructor allocating ptr Copy
constructor allocating ptr.
Copy constructor allocating ptr.
Length of line : 10 Freeing
memory!
Copy constructor allocating ptr.
Length of line : 10 Freeing
memory!
Freeing memory!
Freeing memory!
```

## 3.6 Overloading operators

C++ incorporates the option to use standard operators to perform operations with classes in addition to with fundamental types. For example:

```
Int a,b,c; a
= b + c;
```

This is obviously valid code in C++, since the different variables of the addition are all fundamental types. Nevertheless, it is not so obvious that we could perform an operation similar to the following one:

```
class
{
        string product;
        float price;
} a, b, c;
```

```
a = b + c;
```

In fact, this will cause a compilation error, since we have not defined the behaviour our class should have with addition operations. However, thanks to the C++ feature to overload operators, we can design classes able to perform operations using standard operators. Here is a list of all the operators that can be overloaded:

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

To overload an operator in order to use it with classes we declare operator functions, which are regular functions whose names are the operator keyword followed by the operator sign that we want to overload.

The format is:

```
type operator sign (parameters) { /*...*/ }
```

Here you have an example that overloads the addition operator (+). We are going to create a class to store bi-dimensional vectors and then we are going to add two of them: a(3,1) and b(1,2). The addition of two bi-dimensional vectors is an operation as simple as adding the two x coordinates to obtain the resulting x coordinate and adding the two y coordinates to obtain the resulting y. In this case the result will be (3+1, 1+2) = (4,3).

```
// vectors: overloading operators example
#include <iostream>
using namespace std;

class CVector
{
        public:
int x,y;
CVector () {};
          CVector (int,int);
          CVector operator + (CVector);
};
CVector::CVector (int a, int b)
{
x = a;
y = b;
}

CVector CVector::operator+ (CVector param)
{
        CVector temp;
temp.x = x + param.x;
temp.y = y + param.y;
          return (temp);
}

int main ()
{
          CVector a (3,1);
        CVector b (1,2);
CVector c;      c = a + b;
        cout   <<   c.x   <<   ","   <<   c.y;
return 0;
}
```

Output

```
4,3
```

It may be a little confusing to see so many times the CVector identifier. But, consider that some of them refer to the class name (type) CVector and some others are functions with that name (constructors must have the same name as the class). Do not confuse them:

```
CVector (int, int);                  //  function  name  CVector  (constructor)  CVector
operator+ (CVector);            // function returns a CVector
```

The function operator+ of class CVector is the one that is in charge of overloading the addition operator (+). This function can be called either implicitly using the operator, or explicitly using the function name:

```
c = a + b;
c = a.operator+ (b);
```

Both expressions are equivalent.

Notice also that we have included the empty constructor (without parameters) and we have defined it with an empty block:

```
CVector () { };
```

This is necessary, since we have explicitly declared another constructor:

```
CVector (int, int);
```

And when we explicitly declare any constructor, with any number of parameters, the default constructor with no parameters that the compiler can declare automatically is not declared, so we need to declare it ourselves in order to be able to construct objects of this type without parameters. Otherwise, the declaration:

```
CVector c;
```

included in main() would not have been valid. Anyway, I have to warn you that an empty block is a bad implementation for a constructor, since it does not fulfil the minimum functionality that is generally expected from a constructor, which is the initialization of all the member variables in its class. In our case this constructor leaves the variables x and y undefined. Therefore, a more advisable definition would have been something similar to this:

```
CVector () { x=0; y=0; };
```

which in order to simplify and show only the point of the code I have not included in the example. As well as a class includes a default constructor and a copy constructor even if they are not declared, it also includes a default definition for the assignment operator (=) with the class itself as parameter. The behaviour which is defined by default is to copy the whole content of the data members of the object passed as argument (the one at the right side of the sign) to the one at the left side:

```
CVector d (2,3); CVector
e;
e = d;                    // copy assignment operator
```

The copy assignment operator function is the only operator member function implemented by default. Of course, you can redefine it to any other functionality that you want, like for example, copy only certain class members or perform additional initialization procedures.

The overload of operators does not force its operation to bear a relation to the mathematical or usual meaning of the operator, although it is recommended. For example, the code may not be very intuitive if you use operator + to subtract two classes or operator== to fill with zeros a class, although it is perfectly possible to do so.

Although the prototype of a function operator+ can seem obvious since it takes what is at the right side of the operator as the parameter for the operator member function of the object at its left side, other operators may not be so obvious. Here you have a table with a summary on how the different operator functions have to be declared (replace @ by the operator in each case):

| **Expression** | **Operator** | **Member function** | **Global function** |
|---|---|---|---|
| @a | + - * & ! ~ ++ -- | A::operator@() | operator@(A) |
| a@ | ++ -- | A::operator@(int) | operator@(A, int) |
| a@b | + - * / % ^ & \| < > == != <= >= << >> && \|\| , | A::operator@ (B) | operator@(A,B) |
| a@b | = += -= *= /= %= ^= &= \|= <<= >>= [] | A::operator@ (B) | - |
| a(b, c...) | () | A::operator() (B, C...) | - |

| | | | |
|---|---|---|---|
| a->x | -> | A::operator->() | - |

Where a is an object of class A, b is an object of class B and c is an object of class C. You can see in this panel that there are two ways to overload some class operators: as a member function and as a global function. Its use is indistinct, nevertheless I remind you that functions that are not members of a class cannot access the private or protected members of that class unless the global function is its friend (friendship is explained later).

## 3.7 User-defined conversions

User-defined conversions allow you to specify object conversions with constructors or with conversion functions. User-defined conversions are implicitly used in addition to standard conversions for conversion of initializers, functions arguments, function return values, expression operands, expressions controlling iteration, selection statements, and explicit type conversions.

There are two types of user-defined conversions:
- Conversion by constructor
- Conversion functions

The compiler can use only one user-defined conversion (either a conversion constructor or a conversion function) when implicitly converting a single value. The following example demonstrates this:

```
class A
{
        int x;    public:
                operator int() { return x; };
};

class B
{
            A y;
              publi
            c:
                operator A() { return y; };
};

int main ()
{
            B b_obj;
                // int i = b_obj;
          int j = A(b_obj);
}
```

The compiler would not allow the statement int i = b_obj. The compiler would have to implicitly convert b_obj into an object of type A (with B::operator A()), then implicitly convert that object to an integer (with

A::operator int()). The statement int j = A(b_obj) explicitly converts b_obj into an object of type A, then implicitly converts that object to an integer.

User-defined conversions must be unambiguous, or they are not called. A conversion function in a derived class does not hide another conversion function in a base class unless both conversion functions convert to the same type. Function overload resolution selects the most appropriate conversion function. The following example demonstrates this:

```
class A
{
        int a_int;
char* a_carp;  public:
                    operator int() { return a_int; }
                    operator char*() { return a_carp; }
};

class B : public A
{
        float b_float;
char* b_carp;
public:
                    operator float() { return b_float; }
                    operator char*() { return b_carp; }
};

int main ()
{
          B b_obj;
                    //  long a = b_obj;
          char* c_p = b_obj;
}
```

The compiler would not allow the statement long a = b_obj. The compiler could either use A::operator int() or B::operator float() to convert b_obj into a long. The statement char* c_p = b_obj uses B::operator char*() to convert b_obj into a char* because B::operator char*() hides A::operator char*().
When you call a constructor with an argument and you have not defined a constructor accepting that argument type, only standard conversions are used to convert the argument to another argument type acceptable to a constructor for that class. No other constructors or conversions functions are called to convert the argument to a type acceptable to a constructor defined for that class. The following example demonstrates this:

```
class A
{
        public:
A() { }
                    A(int) { }
};

int main()
{
        A a1 = 1.234;
                // A moocow = "text string";
}
```

The compiler allows the statement A a1 = 1.234. The compiler uses the standard conversion of converting 1.234 into an int, then implicitly calls the converting constructor A(int). The compiler would not allow the statement A moocow = "text string"; converting a text string to an integer is not a standard conversion.

## 3.8 Friendship

### 3.8.1 Friend functions

In principle, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, this rule does not affect friends. Friends are functions or classes declared with the friend keyword.

If we want to declare an external function as friend of a class, thus allowing this function to have access to the private and protected members of this class, we do it by declaring a prototype of this external function within the class, and preceding it with the keyword friend:

```
// friend functions #include
<iostream>
using namespace std;

class CRectangle
{
        int width, height;

        public:
            void set_values (int, int);          int area
() {return (width * height);}              friend
CRectangle duplicate (CRectangle);
};

void CRectangle::set_values (int a, int b)
{
      width = a;
height = b; }

CRectangle duplicate (CRectangle rectparam)
{
        CRectangle rectres;     rectres.width =
rectparam.width*2;    rectres.height =
rectparam.height*2;             return (rectres);
}

int main ()
{
        CRectangle rect, rectb;
rect.set_values (2,3);
rectb = duplicate (rect);
        cout      <<      rectb.area();
return 0;
}
```

Output

```
24
```

The duplicate function is a friend of CRectangle. From within that function we have been able to access the members width and height of different objects of type CRectangle, which are private members. Notice that neither in the declaration of duplicate() nor in its later use in main() have we considered duplicate a member of class CRectangle. It isn't! It simply has access to its private and protected members without being a member.

The friend functions can serve, for example, to conduct operations between two different classes. Generally, the use of friend functions is out of an object-oriented programming methodology, so whenever possible it is better to use members of the same class to perform operations with them. Such as in the previous example, it would have been shorter to integrate duplicate() within the class CRectangle.

### 3.8.2 Friend classes

Just as we have the possibility to define a friend function, we can also define a class as friend of another one, granting that first class access to the protected and private members of the second one.

```
// friend class
#include <iostream> using
namespace std;

class CSquare;

class CRectangle
{
          int width, height;

          public:
```

```
                    int area ()
{
                                        return (width * height);
                            }
                            void convert (CSquare a);
};

class CSquare
{
        private:                int side;
public:                 void set_side
(int a)
                            {
                                        side=a;
                            }
            friend class CRectangle;
};

void CRectangle::convert (CSquare a)
{
            width = a.side;
            height = a.side;
}
int main ()
{
        CSquare sqr;
CRectangle rect;
sqr.set_side(4);
rect.convert(sqr);
            cout << rect.area();
            return 0;
}
```

Output

```
16
```

In this example, we have declared CRectangle as a friend of CSquare so that CRectangle member functions could have access to the protected and private members of CSquare, more concretely to CSquare::side, which describes the side width of the square.

You may also see something new at the beginning of the program: an empty declaration of class CSquare. This is necessary because within the declaration of CRectangle we refer to CSquare (as a parameter in convert()). The definition of CSquare is included later, so if we did not include a previous empty declaration for CSquare this class would not be visible from within the definition of CRectangle. Consider that friendships are not corresponded if we do not explicitly specify so. In our example, CRectangle is considered as a friend class by CSquare, but CRectangle does not consider CSquare to be a friend, so CRectangle can access the protected and private members of CSquare but not the reverse way. Of course, we could have declared also CSquare as friend of CRectangle if we wanted to.

Another property of friendships is that they are not transitive: The friend of a friend is not considered to be a friend unless explicitly specified.

# TOPIC 4

# 4. INHERITANCE

**LEARNING OUTCOMES**

***After Studying this topic you should be able to:***

- Create new classes by inheriting from existing classes
- understand how inheritance promotes software reusability and use multiple inheritance to derive a class from several base classes
- Understand the notion of polymorphism, how to declare and use virtual functions to affect polymorphism
- understand how C++ implements virtual functions and dynamic binding “under the hood”

## 4.1 Inheritance

Inheritance is a mechanism of reusing and extending existing classes without modifying them, thus producing hierarchical relationships between them. Inheritance is almost like embedding an object into a class. Suppose that you declare an object x of class A in the class definition of B. As a result, class B will have access to all the public data members and member functions of class A. However, in class B, you have to access the data members and member functions of class A through object x.

**Base Class:** It is the class whose properties are inherited by another class. It is also called Super Class.
**Derived Class:** It is the class that inherit properties from base class(es).It is also called Sub Class.

In order to derive a class from another, we use a colon (:) in the declaration of the derived class using the following format:

```
class derived_class_name: public base_class_name
{ /*...*/ };
```

Where derived_class_name is the name of the derived class and base_class_name is the name of the class on which it is based. The public access specifier may be replaced by any one of the other access specifiers protected and private. This access specifier limits the most accessible level for the members inherited from the base class: The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

**What is inherited from the base class?**

In principle, a derived class inherits every member of a base class except:

- its constructor and its destructor
- its operator=() members
- its friends

## 4.2 Forms of Inheritance

**Single Inheritance:** It is the inheritance hierarchy wherein one derived class inherits from one base class.

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

class A
{
        int             data;
public:
                void f(int arg)
                {
                        data = arg;
                }
                int g()
                {
                        return data;
                }
};

class B : public A { };

int main()
{
        B               obj;
obj.f(20);
          cout << obj.g() << endl;
}
```

Output

```
20
```

**Multiple Inheritance:** It is the inheritance hierarchy wherein one derived class inherits from multiple base class(es).

```
#include <iostream.h> using
namespace std;
class Square
```

```
{
            protected:
                  int l;              public:
          void set_values (int x)
                        {
                                  l=x;
                        }
};

class CShow
{
            public:
                        void show(int i);
};

void CShow::show (int i)
{
            cout << "The area of the square is::" << i << endl;
}

class Area: public Square, public CShow
{
            public:
                  int        area()
          {
                                  return (l *l);
                        }
};

int main ()
{
            Area r;
            r.set_values (5);
            r.show(r.area());
            return 0;
}
```

Output

```
The area of the square is:: 25
```

In the above example the derived class "Area" is derived from two base classes "Square" and "CShow". This is the multiple inheritance OOP's concept in C++.

**Hierarchical Inheritance:** It is the inheritance hierarchy wherein multiple subclasses inherit from one base class.

```
#include <iostream.h>
class Side
{
        protected:                    int
l;      public:          void set_values
(int x)                 {
                            l=x;
                    }
};
class Square: public Side
{
          public:
               int       sq()
        {
                            return (l *l);
                    }
 };

class Cube: public Side
{
          public:
               int       cub()
        {
                            return (l *l*l);
                    }
 };

int main ()
{
          Square s;
        s.set_values (10);      cout << "The square value
is::" << s.sq() << endl;
          Cube c;
        c.set_values (20);      cout << "The cube value
is::" << c.cub() << endl;       return 0;
}
```

Output

```
The square value is:: 100
The cube value is::8000
```

In the above example the two derived classes "Square", "Cube" uses a single base class "Side". Thus two classes are inherited from a single class. This is the hierarchical inheritance OOP's concept in C++.

**Multilevel Inheritance:** It is the inheritance hierarchy wherein subclass acts as a base class for other classes.

```
#include <iostream.h> class
mm
{
        protected:
                int rollno;
        public:
                void get_num(int a)
            {
      rollno = a;
                }

                void put_num()
                {
                        cout << "Roll Number Is:\n"<< rollno << "\n";
                }
};

class marks : public mm
{
        protected:
                int sub1;
                int sub2;
         public
  :
                void get_marks(int x,int y)
                {
                        sub1 = x;
                        sub2 = y;
                }

                void put_marks(void)
                {
                         cout << "Subject 1:" << sub1 << "\n";
                         cout << "Subject 2:" << sub2 << "\n";
                }
};

class res : public marks
{
      protected:
            float tot;
      public:
};

                              void disp(void)           {
                      tot = sub1+sub2;
           put_num();                          put_marks();
                      cout << "Total:"<< tot;
                           }
};

int main()
{
           res std1;
           std1.get_num(5);
           std1.get_marks(10,20);
             std1.disp();
             return 0;
}
```

Output

```
Roll Number Is:
5
Subject 1: 10
Subject 2: 20
Total: 30
```

In the above example, the derived function "res" uses the function "put_num()" from another derived class "marks", which just a level above. This is the multilevel inheritance OOP's concept in C++.

**Hybrid Inheritance:** The inheritance hierarchy that reflects any legal combination of other four types of inheritance.

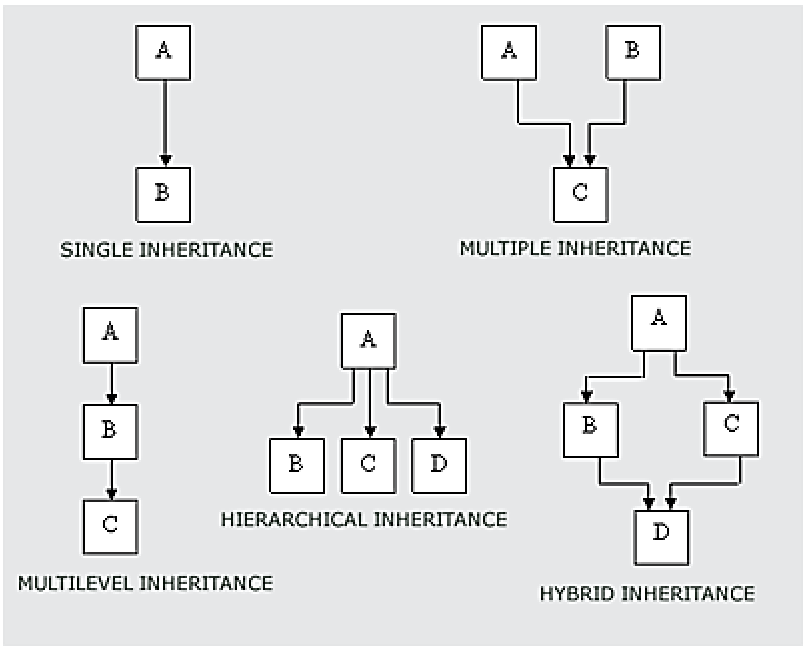
```
#include <iostream.h> class
mm
 {
             protected:
                    int rollno;        public:
            void get_num(int a)
                           {
                                      rollno = a;
                           }
                    void        put_num()
           {
                                      cout << "Roll Number Is:"<< rollno << "\n";
                           }
 };

class marks : public mm
{
             protected:
                    int sub1;                    int
sub2;               public:                      void
get_marks(int x,int y)
                    {
sub1 = x;
sub2 = y;
                         }
                         void put_marks(void)
                         {
                              cout << "Subject 1:" << sub1 << "\n";
          cout << "Subject 2:" << sub2 << "\n";
                         }
 };

class extra
{
          protected:
float e;            public:
void get_extra(float s)
                         {
                                     e=s;
                         }
                         void put_extra(void)
                         {
                                     cout << "Extra Score::" << e << "\n";
                         }
 };

class res : public marks, public extra
{
          protected:                     float tot;
public:                       void disp(void)
{                             tot = sub1+sub2+e;
                    put_num();
put_marks();                             put_extra();
                    cout << "Total:"<< tot;
                         }
};

int main()
{
          res std1;
std1.get_num(10);
std1.get_marks(10,20);
std1.get_extra(33.12);
std1.disp();
```

```
        return 0;
}
```

Output

```
Roll Number Is: 10
Subject 1: 10
Subject 2: 20
Extra score:33.12
Total: 63.12
```

In the above example the derived class "res" uses the function "put_num()". Here the "put_num()" function is derived first to class "marks". Then it is derived and used in class "res". This is an example of "multilevel inheritance-OOP's concept". But the class "extra" is inherited a single time in the class "res", an example for "Single Inheritance". Since this code uses both "multilevel" and "single" inheritance it is an example of "Hybrid Inheritance".



## 4.3 Visibility Mode

It is the keyword that controls the visibility and availability of inherited base class members in the derived class. It can be either private or protected or public.

**Private Inheritance:** It is the inheritance facilitated by private visibility mode. In private inheritance, the protected and public members of base class become private members of the derived class.

**Public Inheritance:** It is the inheritance facilitated by public visibility mode. In public inheritance ,the protected members of base class become protected members of the derived class and public members of the base class become public members of derived class.;

**Protected Inheritance:** It is the inheritance facilitated by protected visibility mode. In protected inheritance, the protected and public members of base class become protected members of the derived class.

| **Base Class Visibility** | **Derived class visibility** | | |
|---|---|---|---|
| | **Public derivation** | **Private derivation** | **Protected derivation** |
| **Private** | Not inherited | Not inherited | Not inherited |
| **Protected** | Protected | Private | Protected |
| **Public** | Public | Private | Protected |

**Containership:** When a class contains objects of other class types as its members, it is called containership. It is also called containment, composition, aggregation.

**Execution of base class constructor**

| **Method of inheritance** | **Order of execution** |
|---|---|
| class B : public A { }; | A(); base constructor<br>B(); derived constructor |
| class A : public B, public C | B();base (first)<br>C();base (second)<br>A();derived constructor |

When both derived and base class contains constructors, the base constructor is executed first and then the constructor in the derived class is executed. In case of multiple inheritances, the base classes are constructed in the order in which they appear in the declaration of the derived class.

## 4.4 Features or Advantages of Inheritance:

**Reusability:**

Inheritance helps the code to be reused in many situations. The base class is defined and once it is compiled, it need not be reworked. Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.

**Saves Time and Effort:**

- The above concept of reusability achieved by inheritance saves the programmer time and effort since the main code written can be reused in various situations as needed.
- Increases Program Structure which results in greater reliability.

## 4.5 Overriding of method (function) in inheritance

We may face a problem in multiple inheritance, when a function with the same name appears in more than one base class. Compiler shows ambiguous error when derived class inherited by these classes uses this function.

We can solve this problem, by defining a named instance within the derived class, using the class resolution operator with the function as below:

```
class P : public M, public N       //multiple inheritance
{
        public :
                void display()  //overrides display() of M and N
                {
                        M::display()
                }
};
```

we can now use the derived class as follows :

```
void main()
{
        P obj;
        obj.display();
}
```

## 4.6 Polymorphism

Polymorphic functions are functions that can be applied to objects of more than one type. In C++, polymorphic functions are implemented in two ways:

- Overloaded functions are statically bound at compile time.
- C++ provides virtual functions. A virtual function is a function that can be called for a number of different user-defined types that are related through derivation. Virtual functions are bound dynamically at run time.

### 4.6.1 Pointers to base class

One of the key features of derived classes is that a pointer to a derived class is type-compatible with a pointer to its base class. Polymorphism is the art of taking advantage of this simple but powerful and versatile feature that brings Object Oriented Methodologies to its full potential.

Example:

```
// pointers to base class #include
<iostream>
using namespace std;

class CPolygon
{
        protected:
             int       width,      height;
public:
                  void set_values (int a, int b)
                  {
                           width=a; height=b;
                  }
 };

class CRectangle: public CPolygon
{
      public:
int area ()            {
                           return (width * height);
                  }
 };

class CTriangle: public CPolygon
{
      public:
int area ()            {
                           return (width * height / 2);
                  }
 };

int main ()
{
        CRectangle rect;
        CTriangle trgl;
      CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;   ppoly1-
>set_values (4,5);     ppoly2-
>set_values (4,5);      cout <<
rect.area() << endl;
        cout    <<    trgl.area()    <<    endl;
return 0;
}
```

Output

```
20
10
```

In function main, we create two pointers that point to objects of class CPolygon (ppoly1 and ppoly2). Then we assign references to rect and trgl to these pointers, and because both are objects of classes derived from CPolygon, both are valid assignment operations. The only limitation in using *ppoly1 and *ppoly2 instead of rect and trgl is that both *ppoly1 and *ppoly2 are of type CPolygon* and therefore we can only use these pointers to refer to the members that CRectangle and CTriangle inherit from CPolygon. For that reason when we call the area() members at the end of the program we have had to use directly the objects rect and trgl instead of the pointers *ppoly1 and *ppoly2.
In order to use area() with the pointers to class CPolygon, this member should also have been declared in the class CPolygon, and not only in its derived classes, but the problem is that CRectangle and CTriangle implement different versions of area, therefore we cannot implement it in the base class. This is when virtual members become handy:

### 4.6.2 Virtual Functions

C++ virtual function is a member function of a class, whose functionality can be over-ridden in its derived classes. The whole function body can be replaced with a new set of implementation in the derived class. The concept of C++ virtual functions is different from C++ Function overloading.

**Virtual Function - Properties:**

C++ virtual function is,

- A member function of a class
- Declared with virtual keyword
- Usually has a different functionality in the derived class
- A function call is resolved at run-time

The difference between a non-virtual C++ member function and a virtual member function is the nonvirtual member functions are resolved at compile time. This mechanism is called static binding. Whereas the C++ virtual member functions are resolved during run-time. This mechanism is known as dynamic binding. The most prominent reason why a C++ virtual function will be used is to have a different functionality in the derived class.

For example a Create function in a class Window may have to create a window with white background. But a class called CommandButton derived or inherited from Window, may have to use a grey background and write a caption on the center. The Create function for CommandButton now should have a functionality different from the one at the class called Window.

Example:

This article assumes a base class named Window with a virtual member function named Create. The derived class name will be CommandButton, with our over ridden function Create.

```
class Window                        // Base class for C++ virtual function example
{
        public:
                virtual void Create()  // virtual function for C++ virtual function example
                {
                        cout <<"Base class Window"<<ENDL;
                }
};

class CommandButton : public Window
{
        public:
                void Create()
                {
          cout<<"Derived class Command Button - Overridden C++ virtual function"<<endl;         }
};

void main()
{
        Window  *x, *y;

x = new Window();                x-
  >Create();

y = new CommandButton();
        y->Create();
}
```

Output

```
Base class Window
Derived class Command Button
```

If the function had not been declared virtual, then the base class function would have been called all the times. Because, the function address would have been statically bound during compile time. But now, as the function is declared virtual it is a candidate for run-time linking and the derived class function is being invoked.

**Virtual function - Call Mechanism:**

Whenever a program has a C++ virtual function declared, a v-table is constructed for the class. The v-table consists of addresses to the virtual functions for classes and pointers to the functions from each of the objects of the derived class. Whenever there is a function call made to the C++ virtual function, the vtable is used to resolve to the function address. This is how the Dynamic binding happens during a virtual function call.

### 4.6.3 Virtual Base Class

Multipath inheritance may lead to duplication of inherited members from a grandparent base class. This may be avoided by making the common base class a virtual base class. When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited.

```
class A
{
 ....
 ....
};
class B1 : virtual public A
{
 ....
 ....
};
class B2 : virtual public A
{
 ....
 ....
};
class C : public B1, public B2
{
 ....  // only one copy of A
 ....  // will be inherited
};
```

### 4.6.4 Abstract base classes

Abstract base classes are something very similar to our CPolygon class of our previous example. The only difference is that in our previous example we have defined a valid area() function with a minimal functionality for objects that were of class CPolygon (like the object poly), whereas in an abstract base classes we could leave that area() member function without implementation at all. This is done by appending =0 (equal to zero) to the function declaration.

An abstract base CPolygon class could look like this:

```
// abstract class CPolygon class
CPolygon
{
        protected:
             int      width,     height;
public:
                void set_values (int a, int b)
                {
                        width = a;
                        height = b;
                }
                virtual int area () =0;
};
```

Notice how we appended =0 to virtual int area () instead of specifying an implementation for the function. This type of function is called a pure virtual function, and all classes that contain at least one pure virtual function are abstract base classes. The main difference between an abstract base class and a regular polymorphic class is that because in abstract base classes at least one of its members lacks implementation we cannot create instances (objects) of it. But a class that cannot instantiate objects is not totally useless. We can create pointers to it and take advantage of all its polymorphic abilities. Therefore a declaration like:

```
             CPolygon poly;
```

would not be valid for the abstract base class we have just declared, because tries to instantiate an object. Nevertheless, the following pointers:

```
             CPolygon * ppoly1;
             CPolygon * ppoly2;
```

would be perfectly valid.

This is so for as long as CPolygon includes a pure virtual function and therefore it's an abstract base class. However, pointers to this abstract base class can be used to point to objects of derived classes. Here you have the complete example:

```
// abstract base class
#include <iostream> using
namespace std;
class CPolygon
{
        protected:
             int      width,     height;
public:
                void set_values (int a, int b)
```

```
                    {
                        width = a;
height = b;
                    }
                    virtual int area (void) =0;
 };

class CRectangle: public CPolygon
{
        public:                 int
area (void)                     {
                              return (width * height);
                    }
 };
class CTriangle: public CPolygon
{
        public:                 int
area (void)                     {
                              return (width * height / 2);
                    }
 };

int main ()
{
          CRectangle rect;
          CTriangle trgl;
        CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;    ppoly1-
>set_values (4,5);       ppoly2-
>set_values (4,5);       cout << ppoly1-
>area() << endl;         cout << ppoly2-
>area() << endl;         return 0;
}
```

Output

```
20
10
```

If you review the program you will notice that we refer to objects of different but related classes using a unique type of pointer (CPolygon*). This can be tremendously useful. For example, now we can create a function member of the abstract base class CPolygon that is able to print on screen the result of the area() function even though CPolygon itself has no implementation for this function:

```
// pure virtual members can be called
// from the abstract base class
#include <iostream>
using namespace std;
```

```
class CPolygon
{
        protected:
             int      width,      height;
public:
                 void set_values (int a, int b)
                 {
                         width = a;
                         height = b;
                 }
                 virtual int area (void) =0;
                 void printarea (void)
                 {
                         cout << this->area() << endl;
                 }
 };

class CRectangle: public CPolygon
{
      public:                    int
area (void)                      {
                                return (width * height);
                 }
 };

class CTriangle: public CPolygon
{
      public:                    int
area (void)                      {
                                return (width * height / 2);
                 }
 };

int main ()
{
        CRectangle rect;
        CTriangle trgl;
      CPolygon * ppoly1 = &rect;
CPolygon * ppoly2 = &trgl;
ppoly1->set_values (4,5);
ppoly2->set_values (4,5);
ppoly1->printarea();
ppoly2->printarea();        return
0;
}
```

Output

```
20
10
```

Virtual members and abstract classes grant C++ the polymorphic characteristics that make object-oriented programming such a useful instrument in big projects. Of course, we have seen very simple uses of these features, but these features can be applied to arrays of objects or dynamically allocated objects.

Let's end with the same example again, but this time with objects that are dynamically allocated:

```
// dynamic allocation and polymorphism
#include <iostream>
using namespace std;

class CPolygon
{
        protected:
            int     width,     height;
public:
                void set_values (int a, int b)
                {
                        width = a;
                        height = b;
                }
                virtual int area (void) =0;
                void printarea (void)
                {
                        cout << this->area() << endl;
                }
 };

class CRectangle: public CPolygon
{
      public:              int
area (void)
                {
                        return (width * height);
                }
 };

class CTriangle: public CPolygon
{
      public:              int
area (void)                {
```

```
                                        return (width * height / 2);
                              }
  };

int main ()
{
        CPolygon * ppoly1 = new CRectangle;
CPolygon * ppoly2 = new CTriangle;  ppoly1-
>set_values (4,5);      ppoly2->set_values
(4,5);   ppoly1->printarea();   ppoly2-
>printarea();   delete ppoly1;          delete
ppoly2;
          return 0;
}
```

Output

```
20
10
```

Notice that the ppoly pointers:

```
          CPolygon * ppoly1 = new CRectangle;
          CPolygon * ppoly2 = new CTriangle;
```

are declared being of type pointer to CPolygon but the objects dynamically allocated have been declared having the derived class type directly.

# TOPIC 5

# 5. ADVANCED CONCEPTS

**LEARNING OUTCOMES**

***After Studying this topic you should be able to:***

- Understand the relationships among templates, friends, inheritance and static members
- Use try, throw and catch to watch for, indicate and handle exceptions respectively, use auto-ptr to prevent memory leaks and understand the standard exception hierarchy
- Understand the concept of a container class
- Understand the notion of iterator classes that walk through the elements of container classes

## 5.1 Templates

### 5.1.1 Function templates

Templates allow for the ability to create generic functions that admit any data type as parameters and return a value without having to overload the function with all the possible data types. Until certain point they fulfil the functionality of a macro. Its prototype is any of the two following ones:

```
template <class identifier> function_declaration; template
<typename identifier> function_declaration;
```

the only difference between both prototypes is the use of keyword class or typename, its use is indistinct since both expressions have exactly the same meaning and behave exactly the same way. For example, to create a template function that returns the greater one of two objects we could use:

```
template <class GenericType>
GenericType GetMax (GenericType a, GenericType b)
{
        return (a>b?a:b);
}
```

As the first line specifies, we have created a template for a generic data type that we have called GenericType. Therefore in the function that follows, GenericType becomes a valid data type and it is used as the type for its two parameters a and b and as the return type for the function GetMax. GenericType still does not represent any concrete data type; when the function GetMax will be called we will be able to call it with any valid data type. This data type will serve as a pattern and will replace GenericType in the function. The way to call a template class with a type pattern is the following:

function **<**pattern**>** **(**parameters**);**

Thus, for example, to call GetMax and to compare two integer values of type int we can write:

```
int x,y;
GetMax <int> (x,y);
```

so GetMax will be called as if each appearance of GenericType was replaced by an int expression. Here is the complete example:

```
// function template
```

```
#include <iostream.h>

template <class T> T
GetMax (T a, T b)
{
	T result;
	result = (a>b)? a : b;
	return (result);
}

int main ()
{
	int i=5, j=6, k;
long l=10, m=5, n;
k=GetMax<int>(i,j);
n=GetMax<long>(l,m);
cout << k << endl;	cout << n
<< endl;
	return 0;
}
```

Output

```
6
10
```

(In this case we have called the generic type T instead of GenericType because it is shorter and in addition is one of the most usual identifiers used for templates, although it is valid to use any valid identifier).

In the example above we used the same function GetMax() with arguments of type int and long having written a single implementation of the function. That is to say, we have written a function template and called it with two different patterns.

As you can see, within our GetMax() template function the type T can be used to declare new objects:

```
T result;
```

result is an object of type T, like a and b, that is to say, of the type that we enclose between angle-brackets <> when calling our template function.

In this concrete case where the generic T type is used as a parameter for function GetMax the compiler can find out automatically which data type is passed to it without having to specify it with patterns <int> or <long>. So we could have written:

```
int                                                              i,j;
GetMax (i,j);
```

since both i and j are of type int the compiler would assume automatically that the wished function is for type int. This implicit method is more usual and would produce the same result:

```
// function template II
#include <iostream.h>

template <class T> T
GetMax (T a, T b)
{
        return (a>b?a:b);
}

int main ()
{
      int i=5, j=6, k;
long l=10, m=5, n;
k=GetMax(i,j);
n=GetMax(l,m);          cout
<< k << endl;           cout
<< n << endl;
        return 0;
}
```

Output

```
6
10
```

Notice how in this case, within function main() we called our template function GetMax() without explicitly specifying the type between angle-brackets <>. The compiler automatically determines what type is needed on each call.

Because our template function includes only one data type (class T) and both arguments it admits are both of that same type, we cannot call our template function with two objects of different types as parameters:

```
            int i; long
            l;
            k = GetMax (i,l);
```

This would be incorrect, since our function waits for two arguments of the same type (or class). We can also make template-functions that admit more than one generic class or data type. For example:

```
template <class T, class U>
T GetMin (T a, U b)
{
```

```
            return (a<b?a:b);
}
```

In this case, our template function GetMin() admits two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call the function by writing:

```
            int i,j;   long
           l;
            i = GetMin<int,long> (j,l);
```

or simply

```
            i = GetMin (j,l);
```

even though j and l are of different types.

### 5.1.2 Class templates

We also have the possibility to write class templates, so that a class can have members based on generic types that do not need to be defined at the moment of creating the class or whose members use these generic types. For example:

```
template <class T>
class pair
{
        T values [2];
        public:
                pair (T first, T second)
                {
                        values[0]=first; values[1]=second;
                }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
 pair<int> myobject (115, 36);
```

this same class would also serve to create an object to store any other type:

```
                pair<float> myfloats (3.0, 2.18);
```

The only member function has been defined inline within the class declaration. If we define a function member outside the declaration we must always precede the definition with the prefix template <... >.

```
// class templates
```

```
#include <iostream.h>
template <class T>
class pair
{
        T       value1,       value2;
public:
                    pair (T first, T second)
                {
value1 = first;
                              value2 = second;
                    }
                    T getmax ();
};

template <class T> T
pair<T>::getmax ()
{
          T retval;
          retval = value1>value2? value1 : value2;
          return retval;
}

int main ()
{
        pair   <int>   myobject   (100,   75);
cout << myobject.getmax();
          return 0;
}
```

Output

```
100
```

notice how the definition of member function getmax begins:

```
                template <class T> T
                pair<T>::getmax ()
```

All Ts that appear are necessary because whenever you declare member functions you have to follow a format similar to this (the second T makes reference to the type returned by the function, so this may vary).

### 5.1.3 Template specialization

A template specialization allows a template to make specific implementations when the pattern is of a determined type. For example, suppose that our class template pair included a function to return the result of the module operation between the objects contained in it, but we only want it to work when the contained type is int. For the rest of the types we want this function to return 0. This can be done the following way:

```
// Template specialization
#include <iostream.h>

template <class T>
class pair
{
        T       value1,       value2;
public:
                    pair (T first, T second)
                {                       value1
= first;                        value2 =
second;
                    }
                    T module ()
                    {
                            return 0;
                    }
};

template <> class
pair <int>
{
        int       value1,       value2;
public:
                    pair (int first, int second)
                {                       value1
= first;                        value2 =
second;
                    }
                    int module ();
};

template      <>      int
pair<int>::module()
{
          return value1%value2;
}

int main ()
{
        pair <int> myints (100,75);     pair
<float> myfloats (100.0,75.0);          cout
<< myints.module() << '\n';
```

```
        cout << myfloats.module() << '\n';
        return 0;
}
```

```
Output
25
0
```

As you can see in the code the specialization is defined this way:

```
    template <> class class_name <type>
```

The specialization is part of a template, for that reason we must begin the declaration with template <>. And indeed because it is a specialization for a concrete type, the generic type cannot be used in it and the first angle-brackets <> must appear empty. After the class name we must include the type that is being specialized enclosed between angle-brackets <>.

When we specialize a type of a template we must also define all the members equating them to the specialization (if one pays attention, in the example above we have had to include its own constructor, although it is identical to the one in the generic template). The reason is that no member is "inherited" from the generic template to the specialized one.

### 5.1.4 Parameter values for templates

Besides the template arguments preceded by the class or typename keywords that represent a type, function templates and class templates can include other parameters that are not types whenever they are also constant values, like for example values of fundamental types. As an example look at this class template that serves to store arrays:

```
// array template
#include <iostream.h>

template <class T, int N>
class array
{
    T     memblock     [N];
public:
            void setmember (int x, T value);
            T getmember (int x);
};

template <class T, int N>
array<T,N>::setmember (int x, T value)
{
        memblock[x]=value;
}
```

```
template  <class  T,  int  N>  T
array<T,N>::getmember (int x)
{
            return memblock[x];
}

int main ()
{
          array <int,5> myints;              array
<float,5> myfloats;     myints.setmember
(0,100);          myfloats.setmember
(3,3.1416);      cout << myints.getmember(0)
<< '\n';
            cout << myfloats.getmember(3) << '\n';
            return 0;
}
```

Output

```
100
3.1416
```

It is also possible to set default values for any template parameter just as it is done with function parameters. Some possible template examples seen above:

```
template <class T>                      // The most usual: one class parameter.
template <class T, class U>             // Two class parameters.
template <class T, int N>               // A class and an integer.
template <class T = char>               // With a default value.
template <int Tfunc (int)>              // A function as parameter.
```

### 5.1.5 Templates and multiple-file projects

From the point of view of the compiler, templates are not normal functions or classes. They are compiled on demand, meaning that the code of a template function is not compiled until an instantiation is required. At that moment, when an instantiation is required, the compiler generates a function specifically for that type from the template.

When projects grow it is usual to split the code of a program in different source files. In these cases, generally the interface and implementation are separated. Taking a library of functions as example, the interface generally consists of the prototypes of all the functions that can be called. These are generally declared in a "header file" with .h extension, and the implementation (the definition of these functions) is in an independent file of C++ code.

The macro-like functionality of templates, forces a restriction for multi-file projects: the implementation (definition) of a template class or function must be in the same file as the declaration. That means we cannot separate the interface in a separate header file and we must include both interface and implementation in any file that uses the templates.

Going back to the library of functions, if we wanted to make a library of function templates, instead of creating a header file (.h) we should create a "template file" with both the interface and implementation of the function templates (there is no convention on the extension for this type of file other than there be no extension at all or to keep the .h). The inclusion more than once of the same template file with both declarations and definitions in a project doesn't generate linkage errors, since they are compiled on demand and compilers that allow templates should be prepared to not generate duplicate code in these cases.

### 5.1.6 Templates and Inheritance

Templates and Inheritance relate in several ways:

- A class-template can be derived from a class-template specialisation.
- A class-template can be derived from a non-template class.
- A class-template specialisation can be derived from a class-template specialisation.
- A non-template class can be derived from a class-template specialisation.

### 5.1.7 Templates and static Members

Remember that, with a non-template class, one copy of a static data member is shared among all objects of the class and the static data member must be initialised at file scope.

Each class-template specialisation instantiated from a class template has its own copy of each static data member of the class-template; all objects of that specialisation share that one static data member. In addition, as with static data members of non-template classes, static data members of class-template specialisation must be initialised at file scope. Each class-template specialisation gets its own copy of the class template’s static data member.

### 5.1.8 Templates and Friends

There are four kinds of relationships between classes and their friends when templates are involved:

- One-to-many: A non-template function may be a friend to all template class instantiations.
- Many-to-one: All instantiations of a template function may be friends to a regular non-template class.
- One-to-one: A template function instantiated with one set of template arguments may be a friend to one template class instantiated with the same set of template arguments. This is also the relationship between a regular non-template class and a regular non-template friend function.
- Many-to-many: All instantiations of a template function may be a friend to all instantiations of the template class.

The following example demonstrates these relationships:

```
class B
{
        template<class V> friend int j();
}

template<class S> g();

template<class T> class A
{
      friend int e();   friend
    int f(T);           friend
    int g<T>();
      template<class U>
    friend int h();
};
```

- Function e() has a one-to-many relationship with class A. Function e() is a friend to all instantiations of class A.
- Function f() has a one-to-one relationship with class A. The compiler will give you a warning for this kind of declaration similar to the following:

The friend function declaration "f" will cause an error when the enclosing template class is instantiated with arguments that declare a friend function that does not match an existing definition. The function declares only one function because it is not a template but the function type depends on one or more template parameters.

- Function g() has a one-to-one relationship with class A. Function g() is a function template. It must be declared before here or else the compiler will not recognize g<T> as a template name. For each instantiation of A there is one matching instantiation of g(). For example, g<int> is a friend of A<int>.
- Function h() has a many-to-many relationship with class A. Function h() is a function template. For all instantiations of A all instantiations of h() are friends.
- Function j() has a many-to-one relationship with class B.

These relationships also apply to friend classes.

## 5.2 Exception handling

During the development of a program, there may be some cases where we do not have the certainty that a piece of the code is going to work right, either because it accesses resources that do not exist or because it gets out of an expected range, etc...

These types of anomalous situations are included in what we consider exceptions and C++ has recently incorporated three new operators to help us handle these situations: try, throw and catch. Their form of use is the following:

```
 try
{
  // code to be tried
  throw exception;
}
catch (type  exception)
{
  // code to be executed in case of exception
}
```

And its operation:

- The code within the try block is executed normally. In case that an exception takes place, this code must use the throw keyword and a parameter to throw an exception. The type of the parameter details the exception and can be of any valid type.
- If an exception has taken place, that is to say, if it has executed a throw instruction within the try block, the catch block is executed receiving as parameter the exception passed by throw.

For example:

```
 // exceptions
#include <iostream.h>

int main ()
{
	char myarray[10];
	try
	{
		for (int n=0; n<=10; n++)
		{
			if (n>9) throw "Out of range";
				myarray[n]='z';
		}
	}
	catch (char * str)
	{
		cout << "Exception: " << str << endl;
	}
	return 0;
}
```

Output

```
Exception: Out of range
```

In this example, if within the n loop, n gets to be more than 9 an exception is thrown, since myarray[n] would in that case point to a non-trustworthy memory address. When throw is executed, the try block finalizes right away and every object created within the try block is destroyed. After that, the control is passed to the corresponding catch block (that is only executed in these cases). Finally the program continues right after the catch block, in this case: return 0;.

The syntax used by throw is similar to that of return: Only the parameter does not need to be enclosed between parentheses.

The catch block must go right after the try block without including any code line between them. The parameter that catch accepts can be of any valid type. Even more, catch can be overloaded so that it can accept different types as parameters. In that case the catch block executed is the one that matches the type of the exception sent (the parameter of throw):

```
// exceptions: multiple catch blocks
#include <iostream.h>

int main ()
{
        try
         {
              char * mystring;
mystring = new char [10];
              if (mystring == NULL) throw "Allocation failure";
       for (int n=0; n<=100; n++)
                          {
                                 if (n>9) throw n;
                                        mystring[n]='z';
                          }
        }
        catch (int i)
        {
               cout << "Exception: ";
               cout << "index " << i << " is out of range" << endl;
        }
        catch (char * str)
        {
               cout << "Exception: " << str << endl;
        }
        return 0;
}
```

Output

```
Exception: index 10 is out of range
```

In this case there is a possibility that at least two different exceptions could happen:

1. That the required block of 10 characters cannot be assigned (something rare, but possible): in this case an exception is thrown that will be caught by catch (char * str).
2. That the maximum index for mystring is exceeded: in this case the exception thrown will be caught by catch (int i), since the parameter is an integer number.

We can also define a catch block that captures all the exceptions independently of the type used in the call to throw. For that we have to write three points instead of the parameter type and name accepted by catch:

```
try
{
        // code here
}
catch (...)
{
     cout << "Exception occurred";  }
```

It is also possible to nest try-catch blocks within more external try blocks. In these cases, we have the possibility that an internal catch block forwards the exception received to the external level, for that the expression throw; with no arguments is used. For example:

```
     try
     {
             try
             {
                     // code here
             }
             catch (int n)
         {
throw;
             }
     }
     catch (...)
     {
             cout << "Exception occurred";
     }
```

### 5.2.1 Exceptions not caught

If an exception is not caught by any catch statement because there is no catch statement with a matching type, the special function terminate will be called. This function is generally defined so that it terminates the current process immediately showing an "Abnormal termination" error message. Its format is: void terminate();

### 5.2.2 Standard exceptions

Some functions of the standard C++ language library send exceptions that can be captured if we include them within a try block. These exceptions are sent with a class derived from std::exception as type. This class (std::exception) is defined in the C++ standard header file <exception> and serves as a pattern for the standard hierarchy of exceptions:

exception
bad_alloc (thrown by new)
(thrown by dynamic_cast when fails with a
bad_cast referenced type)
(thrown when an exception doesn't match any
bad_exception
catch)
bad_typeid (thrown by typeid)
logic_error domain_error
invalid_argument length_error
out_of_range runtime_error
overflow_error range_error
underflow_error
ios_base::failure (thrown by ios::clear)

Because this is a class hierarchy, if you include a catch block to capture any of the exceptions of this hierarchy using the argument by reference (i.e. adding an ampersand & after the type) you will also capture all the derived ones (rules of inheritance in C++).

The following example catches an exception of type bad_typeid (derived from exception) that is generated when requesting information about the type pointed by a null pointer:

```
// standard exceptions

#include <iostream.h>
#include <exception>
#include <typeinfo>

class A
{
        virtual f() {};
};

int main ()
{
        try
        {
                A * a = NULL;
                typeid (*a);
        }
        catch (std::exception& e)
        {
                cout << "Exception: " << e.what();
        }
        return 0;
}
```

Output

```
Exception: Attempted typeid of NULL pointer
```

You can use the classes of standard hierarchy of exceptions to throw your exceptions or derive new classes from them.

### 5.2.3 Define New Exceptions

You can define your own exceptions by inheriting and overriding exception class functionality. Following is the example which shows how you can use std::exception class to implement your own exception in standard way:

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

class MyException : public exception
{
        const char * what () const throw ()
        {
                return "C++ Exception";
        }
};
 int main()
{
        try
        {
                throw MyException();
        }
        catch(MyException& e)
        {
   std::cout << "MyException caught" << std::endl;      std::cout << e.what()
<< std::endl;
        }
        catch(std::exception& e)
        {
                //Other errors
        }
}
```

Output

```
MyException caught
C++ Exception
```

Here what() is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.

### 5.2.4 auto_ptr

auto_ptr type is provided by the C++ standard library as a sort of smart pointer that helps to avoid resource leaks when exceptions are thrown. Here is a typical example which has potential of memory leak.

```
void memory_leak()
{
        ClassA * ptr = new ClassA;
        ...
        delete ptr;
}
```

The reason why this function is a source of trouble is that the deletion of the object might be forgotten especially if we have return inside of it. Also an exception would exit the function before the delete statement at the end of the function causing a resource leak. Usually, we do try to capture all exceptions as in the example below.

```
void memory_leak()
{
        ClassA * ptr = new ClassA;
        try
        {
        ...
        }
        catch(...)
        {
            delete ptr;
throw;
        }
        delete ptr;
}
```

As we see in the example, trying to handle the deletion of this object properly in the event of an exception makes the code more complicated and redundant. So, we need a pointer which can free the data to which it points whenever the pointer itself gets destroyed. Because the pointer is a local variable, it will be destroyed automatically when the function is exited regardless of whether the exit is normal or caused by an exception.

In other words, if an exception occurs after successful memory allocation but before the delete statement executes, a memory leak could occur. The C++ standard provides class template auto_ptr in header file <memory> to deal with this situation.

Our auto_ptr is a pointer that serves as owner of the object to which it refers. So, an object gets destroyed automatically when its auto_ptr gets destroyed. The function in the 1st example can be rewritten using auto_ptr:

```
#include <memory>
void memory_leak()
{
        std::auto_ptr ptr(new ClassA);
        ...
}
```

The delete statement and catch clause are no longer needed.

An auto_ptr has the same interface as an ordinary pointer. Operator * dereferences the object and operator -> provides access to a member if the object is a class or a structure. But the pointer arithmetic such as ++ is not defined.

One more thing we should be careful about the usage of the pointer is that auto_ptr does not allow us to initialize an object with an ordinary pointer by using the assignment syntax. So, we must initialize the auto_ptr directly by using its value.

```
            std::auto_ptr ptr1(new ClassA);            // RIGHT
std::auto_ptr ptr1 = new ClassA;            // WRONG
```

Here is the example of auto_ptr in action:

```
#include <iostream>
#include <memory>

using namespace std;

class Double
{
        public:
                Double(double d = 0) : dValue(d) { cout << "constructor: " << dValue << endl; }
                ~Double() { cout << "destructor: " << dValue << endl; }
            void setDouble(double d) { dValue = d; }        private:
                    double dValue;
};

int main()
{
        auto_ptr<Double> ptr(new Double(3.14));
      (*ptr).setDouble(6.28);
return 0;
}
```

The example creates auto_ptr object ptr and initializes it with a pointer to a dynamically allocated Double object. Because ptr is a local automatic variable in main(), ptr is destroyed when main terminates. The auto_ptr destructor forces a delete of the Double object pointed to by ptr, which in turn calls the Double class destructor. The memory that Double occupies is released. The Double object will be deleted automatically when the auto_ptr object's destructor gets called.

Only one auto_ptr at a time can own a dynamically allocated object. Thus, the object cannot be an array. By using its overloaded assignment operator or copy constructor, an auto_ptr can transfer ownership of the dynamic memory it manages. The last auto_ptr object that maintains the pointer to the dynamic memory will delete the memory. This makes auto_ptr an ideal mechanism for returning dynamically allocated memory to client code. When the auto_ptr goes out of scope in the client code, the auto_ptr's destructor deletes the dynamic memory.

Though std::auto_ptr is responsible for managing dynamically allocated memory and automatically calls delete to free the dynamic memory when the auto_ptr is destroyed or goes out of scope, auto_ptr have some limitations.

- An auto_ptr can't point to an array. When deleting a pointer to an array we must use delete[] to ensure that destructors are called for all objects in the array, but auto_ptr uses delete.
- It can't be used with the STL containers-elements in an STL container. When an auto_ptr is copied, ownership of the memory is transferred to the new auto_ptr and the original is set to NULL. In other words, auto_ptrs don't work in STL containers because the containers, or algorithms manipulating them, might copy the stored elements. Copies of auto_ptrs aren't equal because the original is set to NULL after being copied. An STL container may make copies of its elements, so you can't guarantee that a valid copy of the auto_ptr will remain after the algorithm processing the container's elements finishes.

An auto_ptr is simply an object that holds a pointer for us within a function. Holding a pointer to guarantee deletion at the end of a scope is what auto_ptr is for, and for other uses requires very specialized skills from a programmer.

## 5.3 <u>Container Classes</u>

A container is an STL (Standard template library) template class that manages a sequence of elements. Such elements can be of any object type that supplies a copy constructor, a destructor, and an assignment operator (all with sensible behavior, of course). The destructor may not throw an exception. This document describes the properties required of all such containers, in terms of a generic template class Cont. An actual container template class may have additional template parameters. It will certainly have additional member functions.

The STL template container classes are:

- deque
- list
- map
- multimap
- multiset
- set
- vector

### 5.3.1 Iterators

Iterator: a pointer-like object that can be incremented with ++, dereferenced with *, and compared against another iterator with !=. Iterators are generated by STL container member functions, such as begin() and end(). Some containers return iterators that support only the above operations, while others return iterators that can move forward and backward, be compared with <, and so on.

The generic algorithms use iterators just as you use pointers in C to get elements from and store elements to various containers. Passing and returning iterators makes the algorithms

- more generic, because the algorithms will work for any containers, including ones you invent, as long as you define iterators for them
- more efficient

Some algorithms can work with the minimal iterators, others may require the extra features. So a certain algorithm may require certain containers because only those containers can return the necessary kind of iterators.

Let us take the following program demonstrates the vector container (a C++ Standard Template) which is similar to an array with an exception that it automatically handles its own storage requirements in case it grows:

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

int main()
{
                    //    create    a    vector    to    store    int
vector<int> vec;
        int i;
```

```
                                // display the original size of vec
          cout << "vector size = " << vec.size() << endl;

                          // push 5 values into the vector        for(i
= 0; i < 5; i++)
          {
                    vec.push_back(i);
          }

                                // display extended size of vec
          cout << "extended vector size = " << vec.size() << endl;

                          // access 5 values from the vector     for(i    =
0; i < 5; i++)
          {
                    cout << "value of vec [" << i << "] = " << vec[i] << endl;
          }

                                // use iterator to access the values
        vector<int>::iterator     v     =     vec.begin();
while( v != vec.end())
          {
                    cout << "value of v = " << *v << endl;
                    v++;
          }

          return 0;
}
```

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

```
vector size = 0 extended
vector size = 5 value of
vec [0] = 0 value of vec
[1] = 1
value of vec [2] = 2
value of vec [3] = 3
value of vec [4] = 4
value of v = 0 value
of v = 1 value of v
= 2 value of v = 3
value of v = 4
```

Here are following points to be noted related to various functions we used in the above example:

- The push_back( ) member function inserts value at the end of the vector, expanding its size as needed.
- The size( ) function displays the size of the vector.
- The function begin( ) returns an iterator to the start of the vector.
- The function end( ) returns an iterator to the end of the vector.

# TOPIC 6

# 6. WORKING WITH FILES

**LEARNING OUTCOMES**

***After Studying this topic you should be able to:***

- Understand working with files
- Be Informed understanding of file manipulations including error handling techniques for file operations
- Be comfortable with command line arguments

## 6.1 File Streams

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 chapter 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:

| Data Type | Description |
|---|---|
| Ofstream | This data type represents the output file stream and is used to create files and to write information to files. |
| Ifstream | This data type represents the input file stream and is used to read information from files. |
| 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.

## 6.2 Opening a File

A file must be opened before you can read from it or write to it. Either the 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.

| Mode Flag | Description |
|---|---|
| ios::app | Append mode. All output to that file to be appended to the end. |
| ios::ate | Open a file for output and move the read/write control to the end of the file. |
| ios::in | Open a file for reading. |

| | |
|---|---|
| ios::out | Open a file for writing. |
| ios::trunk | If the file already exists, its contents will be truncated before opening the file. |

You can combine two or more of these values by ORing them together. For example if you want to open a file in write mode and want to truncate it in case it 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 );
```

## 6.3 Closing a File

When a C++ program terminates it automatically closes 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();
```

## 6.4 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.

## 6.5 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.

## 6.6 Read & Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information inputted 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 to 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 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 makes 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.

## 6.7 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.

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 );
```

## 6.8 End-of-file Checking For Systems Which Implement "eof()"

Special care has to be taken with input when the end of a file is reached. Most versions of C++ (including GNU g++ and Microsoft Visual C++) incorporate an end-of-file (EOF) flag, and a member function called "eof()" for ifstreams to test if this flag is set to True or False. It's worth discussing such systems briefly, since many text books assume this useful facility.

In such systems, when an ifstream is initially connected to a file, the EOF flag is set to False (even if the file is empty). However, if the ifstream "in_stream" is positioned at the end of a file, and the EOF flag is False, the statement

```
in_stream.get(ch);
```

will leave the variable "ch" in an unpredictable state, and set the EOF flag to True. Once the EOF flag is set to True, no attempt should be made to read from the file, since the results will be unpredictable. To illustrate with a diagrammatic example, if we start from

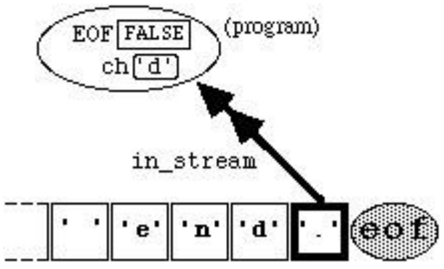


**Figure 6.8.1**

and then execute the statement

in_stream.get(ch);.

this results in the state

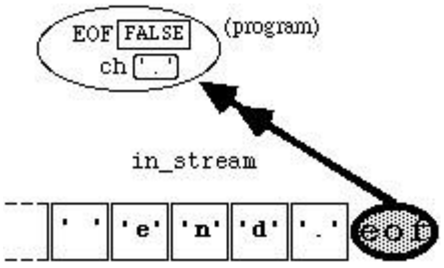


**Figure 6.8.2**

If we again execute the statement

```
in_stream.get(ch);
```

this now results in the state

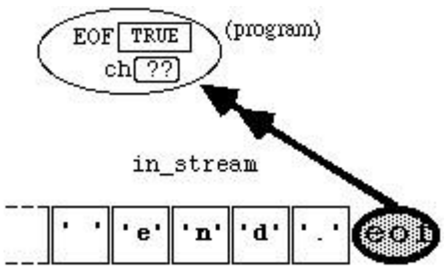


**Figure 6.8.3**

The boolean expression

in_stream.eof()

will now evaluate to True.

Below is a simple program which uses these techniques to copy the file "Eg" simultaneously to the screen and to the file "Copy_of_Eg". Note the use of a while loop in this program. "While" loops are simplified versions of "for" loops, without the initialisation and update statements in the "()" parentheses.

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
 char      character;          ifstream
in_stream;
             ofstream out_stream;

             in_stream.open("Lecture_4");
             out_stream.open("Copy_of_4");

             in_stream.get(character);
             while (!in_stream.eof())
             {
                    cout << character;
out_stream.put(character);
                          in_stream.get(character);
             }

             out_stream.close();
             in_stream.close();
```

```
        return 0;
}
```

## 6.9 Checking for Failure with File Commands

File operations, such as opening and closing files, are a notorious source of errors. Robust commercial programs should always include some check to make sure that file operations have completed successfully, and error handling routines in case they haven't. A simple checking mechanism is provided by the member function "fail()". The function call:

```
                in_stream.fail();
```

returns True if the previous stream operation on "in_stream" was not successful (perhaps we tried to open a file which didn't exist). If a failure has occurred, "in_stream" may be in a corrupted state, and it is best not to attempt any more operations with it. The following example program fragment plays very safe by quitting the program entirely, using the "exit(1)" command from the library "cstdlib":

```
#include <iostream>
#include <fstream>
#include <cstdlib>

using namespace std;

int main()
{
        ifstream in_stream;

        in_stream.open("Lecture_4");
        if (in_stream.fail())
        {
                cout << "Sorry, the file couldn't be opened!\n";
                exit(1);
        }
        ...
}
```

**Error Handling Functions**

| **Function** | **Return Value And Meaning** |
|---|---|
| eof() | returns true (non zero) if end of file is encountered while reading; otherwise return false(zero) |
| fail() | return true when an input or output operation has failed |

|  |  |
|---|---|
| bad() | returns true if an invalid operation is attempted or any unrecoverable error has occurred. |
| good() | returns true if no error has occurred. |

## 6.10 Command Line Arguments

In C++ it is possible to accept command line arguments. Command-line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system. To use command line arguments in your program, you must first understand the full declaration of the main function, which previously has accepted no arguments. In fact, main can actually accept two arguments:

```
int main(int argc, char* argv[])
{
        ...
}
```

The first argument (argc) is the number of elements in the array, which is the second argument (argv). The second argument is always an array of char*, because the arguments are passed from the command line as character arrays (an array can be passed only as a pointer). Each whitespace-delimited cluster of characters on the command line is turned into a separate array argument. The following program prints out all its command-line arguments by stepping through the array:

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

int main(int argc, char* argv[])
{
        cout << "argc = " << argc << endl;
        for(int i = 0; i < argc; i++)
                cout << "argv[" << i << "] = " << argv[i] << endl;
return 0;
}
```

You'll notice that argv[0] is the path and name of the program itself. This allows the program to discover information about itself. It also adds one more to the array of program arguments, so a common error when fetching command-line arguments is to grab argv[0] when you want argv[1].

You are not forced to use argc and argv as identifiers in main( ), those identifiers are only conventions (but it will confuse people if you don't use them). Also, there is an alternate way to declare argv:

```
int main(int argc, char** argv)
```

```
{
	...
}
```

Both forms are equivalent.

All you get from the command-line is character arrays; if you want to treat an argument as some other type, you are responsible for converting it inside your program. To facilitate the conversion to numbers, there are some helper functions in the Standard C library, declared in <cstdlib>. The simplest ones to use are atoi( ), atol( ), and atof( ) to convert an ASCII character array to an int, long, and double, respectively. Here's an example using atoi( ) (the other two functions are called the same way):

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main(int argc, char* argv[])
{
	for(int i = 1; i < argc; i++)
cout << atoi(argv[i]) << endl;
return 0;
}
```

In this program, you can put any number of arguments on the command line. You'll notice that the for loop starts at the value 1 to skip over the program name at argv[0]. Also, if you put a floating-point number containing a decimal point on the command line, atoi( ) takes only the digits up to the decimal point. If you put non-numbers on the command line, these come back from atoi( ) as zero.