Onlinevarsity
your e-way to learning

Object-Oriented Programming with C++

Object-Oriented Programming with C++

Learner's Guide

APTECH LIMITED
Contact E-mail: ov-support@onlinevarsity.com

Edition 1 - 2014

Dear Learner,

We congratulate you on your decision to pursue an Aptech course.

Aptech Ltd. designs its courses using a sound instructional design model – from conceptualization to execution, incorporating the following key aspects:

- Scanning the user system and needs assessment

Needs assessment is carried out to find the educational and training needs of the learner

Technology trends are regularly scanned and tracked by core teams at Aptech Ltd. TAG* analyzes these on a monthly basis to understand the emerging technology training needs for the Industry.

An annual Industry Recruitment Profile Survey is conducted during August - October to understand the technologies that Industries would be adapting in the next 2 to 3 years. An analysis of these trends & recruitment needs is then carried out to understand the skill requirements for different roles & career opportunities.

The skill requirements are then mapped with the learner profile (user system) to derive the Learning objectives for the different roles.

- Needs analysis and design of curriculum

The Learning objectives are then analyzed and translated into learning tasks. Each learning task or activity is analyzed in terms of knowledge, skills and attitudes that are required to perform that task. Teachers and domain experts do this jointly. These are then grouped in clusters to form the subjects to be covered by the curriculum.

In addition, the society, the teachers, and the industry expect certain knowledge and skills that are related to abilities such as *learning-to-learn, thinking, adaptability, problem solving, positive attitude etc*. These competencies would cover both cognitive and affective domains.

A precedence diagram for the subjects is drawn where the prerequisites for each subject are graphically illustrated. The number of levels in this diagram is determined by the duration of the course in terms of number of semesters etc. Using the precedence diagram and the time duration for each subject, the curriculum is organized.

- Design & development of instructional materials

The content outlines are developed by including additional topics that are required for the completion of the domain and for the logical development of the competencies identified. Evaluation strategy and scheme is developed for the subject. The topics are arranged/organized in a meaningful sequence.

The detailed instructional material – Training aids, Learner material, reference material, project guidelines, etc.- are then developed. Rigorous quality checks are conducted at every stage.

- ➢ Strategies for delivery of instruction

Careful consideration is given for the integral development of abilities like thinking, problem solving, learning-to-learn etc. by selecting appropriate instructional strategies (training methodology), instructional activities and instructional materials.

The area of IT is fast changing and nebulous. Hence considerable flexibility is provided in the instructional process by specially including creative activities with group interaction between the students and the trainer. The positive aspects of Web based learning –acquiring information, organizing information and acting on the basis of insufficient information are some of the aspects, which are incorporated, in the instructional process.

- ➢ Assessment of learning

The learning is assessed through different modes – tests, assignments & projects. The assessment system is designed to evaluate the level of knowledge & skills as defined by the learning objectives.

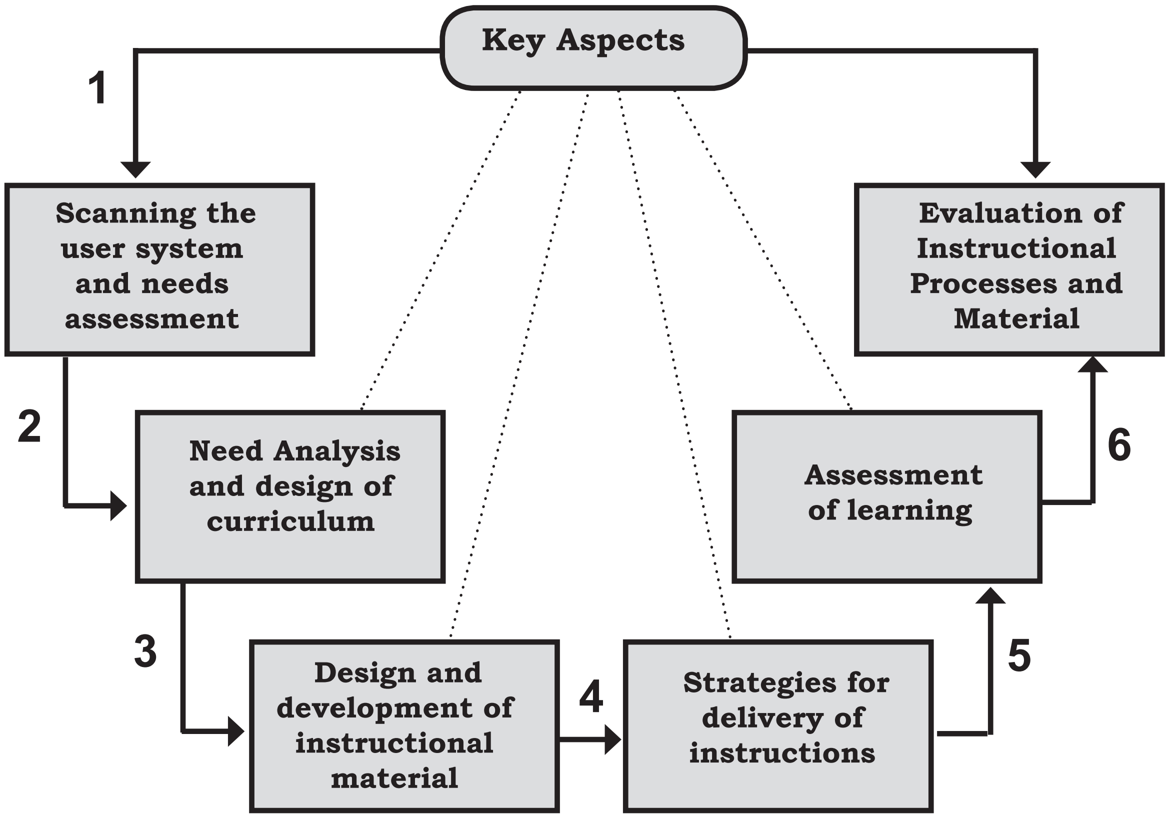
- ➢ Evaluation of instructional process and instructional materials

The instructional process is backed by an elaborate monitoring system to evaluate - on-time delivery, understanding of a subject module, ability of the instructor to impart learning. As an integral part of this process, we request you to kindly send us your feedback in the reply pre-paid form appended at the end of each module.

*TAG – Technology & Academics Group comprises of members from Aptech Ltd., professors from reputed Academic Institutions, Senior Managers from Industry, Technical gurus from Software Majors & representatives from regulatory organizations/forums.

Technology heads of Aptech Ltd. meet on a monthly basis to share and evaluate the technology trends. The group interfaces with the representatives of the TAG thrice a year to review and validate the technology and academic directions and endeavors of Aptech Ltd.

Aptech New Products Design Model


Key Aspects
1
Scanning the user system and needs assessment
2
Need Analysis and design of curriculum
3
Design and development of instructional material
4
Strategies for delivery of instructions
5
Assessment of learning
6
Evaluation of Instructional Processes and Material

Preface

One of the most popular object-oriented programming languages is C++. C++ was developed by Bjarne Stroustrup using C as a base but implementing the principles of object-orientation most effectively to give you a powerful language. Although C was used as the base for developing C++, you will be surprised to see that there is very little overlap between the two languages' features. While teaching the concepts of C++, we have endeavoured to relate the details of the language to object-oriented concepts. This will help you to understand why certain features of the language exist.

The objective of this book is to introduce Object oriented programming. Sessions aim at teaching students in-depth programming in C++. The different object-oriented concepts are briefly explained, and these are followed by their practical implementation using C++. Starting with programming basics, we gradually move toward the key features of object-oriented programming, namely, data hiding, and polymorphism. Inheritance, another important feature of object-oriented programming has been covered in depth. Interesting features of C++ such as function overloading, operator overloading, and file handing have been discussed to a greater extent.

The knowledge and information in this book is the result of a concerted effort of the Design Team, which is continuously striving to bring you the best and the latest in Information Technology. The process of Design has been a part of the ISO 9001 certification for Aptech - IT Division, Education Support Services. As a part of Aptech's quality drive, this team does extensive research and curriculum enrichment to keep it in line with the industry trends.

We will be glad to receive your suggestions. Please send us your feedback, addressed to the Design Head at Aptech's Corporate Office.

We will be glad to receive your suggestions.

Design Team

Sessions

1. Object Oriented Concepts and C++
2. Basics of C++
3. Flow control statements
4. Functions, Pointers and Arrays
5. Function Overloading
6. Inheritance
7. Multiple Inheritance and Polymorphism
8. Data Structures using C++ and Exception Handling

ASK to LEARN

Onlinevarsity
your e-way to learning

Questions *in your* mind?

are here to HELP

Post your queries in ASK to LEARN @

www.onlinevarsity.com

Session -1

Object Oriented Concepts in C++

Welcome to the Session, **Object Oriented Concepts in C++**.

This session introduces Object oriented Programming concepts. The session describes concept of Class and Objects. Further, the session describes the drawbacks of traditional programming and advantages of object-Oriented Programming. Finally, the session discusses on member functions and member functions accessibilty.

At the end of this session, you will be able to:

➔ Discuss the following
- The Object-Oriented approach
- Drawbacks of traditional programming
- Object-Oriented Programming

➔ Discuss basic Object-Oriented concepts such as:
- Objects
- Classes
  - Properties
  - Methods
- Abstraction
- Inheritance
- Encapsulation
- Polymorphism

➔ Define the structure of a C++ program

➔ Identify the standard input and output functions

➔ Use comments and width(), endl() functions

➔ Describe Private and Public sections of Classes with member functions

➔ Use the Objects and Member functions of a Class

1.1 Introduction

Bjarne Stroustrup developed C++ at the Bell Laboratories in 1983. C++, at a very abstract level, can be viewed as an enhancement to the C language. The C language was originally developed by Dennis Ritchie of the Bell Laboratories in 1972, primarily to facilitate managing, programming and maintaining large software projects.

There are several C++ compilers available - Turbo C++, Borland C++, Zortech C++, AT & T C++, and Sun C++ Preprocessor. Turbo C++ and Borland C++ are the most widely used amongst them. The difference between the two is that by using Borland C++, programs can be written to work under the Windows environment as well.

We live in the world of objects. The thoughts we think are in terms of objects that may be tangible. For example, when we say, “we are reading books …”, we refer to the tangible object - book. We try to define any tangible object that exists with certain attributes. The attributes basically define a boundary for the object we perceive.

This module focuses on several features of object-oriented technology. We will start with the concept of objects and classes and will gradually move towards the more advanced concepts.

This session aims to teach students the basics of the object-oriented approach. The different object-oriented concepts are briefly explained. Every concept has been followed by its practical implementation using C++.

1.2 The Object-Oriented Approach

We can classify living beings as objects just as we classify things we use as objects of different types. Let us think of departments as objects in an organization. Figure 1.1 depicts the concept of an organization. Typically, an organization has departments that deal with administration, sales, accounts, marketing, and so on. Each department has its own personnel who perform clearly assigned duties. Each department also has its own data, such as personnel records, inventory, sales figures, or other data related to the functioning of that department.

When the whole organization is split up into departments, it is easier to manage activities and personnel. The people in each department control and operate on that department’s data. The accounts department is in charge of salaries for the organization. If you are from the marketing division and need to find out details regarding the salaries of your division, all you need to do is enquire in the accounts department for the relevant information.

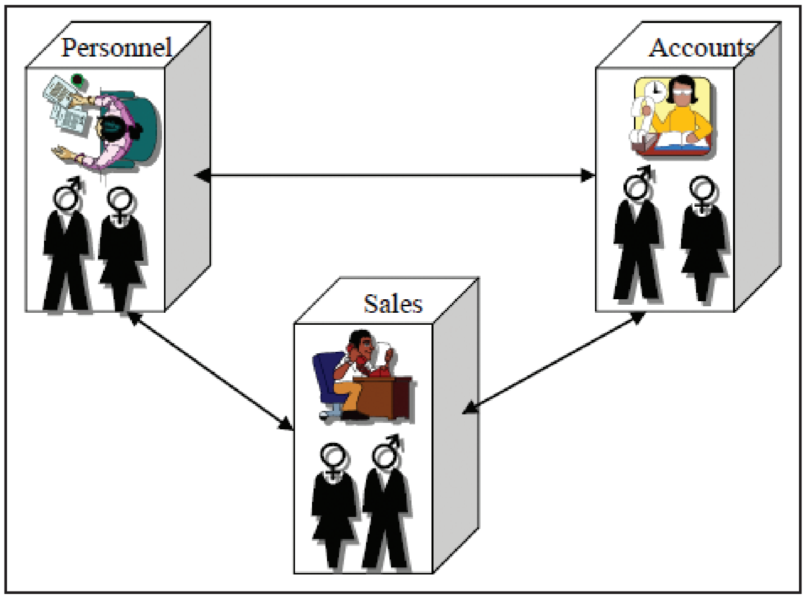


Figure 1.1: A Typical Organization

Let us first see how programming is done traditionally and then move on to the changes that object-orientation can bring about.

1.2.1 Drawbacks of Traditional Programming

It is often said that necessity is the mother of invention. The traditional approach was not able to handle the increasing demands of software application in several aspects of programming.

We have already learnt the C language that uses the traditional programming approach for application development. In the traditional approach, we divide the application into several tasks and write code for all the tasks. Therefore, the focus in this approach is on the tasks.

While this traditional programming approach has its own advantages; it has several limitations as well. Some of these limitations are described.

- **Unmanageable programs**

  Traditional programs consist of a list of instructions written in a language that tells the computer to perform some actions. Functions or procedures or subroutines were adopted to make programs more comprehensible to programmers. A program is divided into functions with clearly defined purposes. Each function can interact with another function or act upon given data. As programs grow ever larger and more complex, even the use of functions can create problems.

- **Problems in modification of data**

  Data plays a crucial role in traditional programming techniques. As we have seen in the case of an organization the data available in each department is important.

Some or all functions of one or more departments may need to access certain common data. If you add new data items, you will need to modify all the tasks or functions that access the data so that they can also access these new items. For example, personnel records are usually the kind of data that several departments will need to access at sometime or the other. As a result, when personnel records are revised, there are a number of functions that will be affected.

Similarly, in a programming application, when data is made available to all functions, it becomes difficult to locate all such functions, and even harder to modify them correctly as the size of the program increases. This is especially difficult when a new programmer tries to understand the intricacies of a large program. What is needed is a way to restrict access to the data so that only a few critical functions act upon it.

The requirements of a system are always being revised. When there are revisions in a program they should not affect the whole system. For example, if the sales department were to make changes in their sales strategy, the change should in no way affect the way salaries are paid to the personnel. In traditional programming techniques, it is next to impossible to separate parts of a program from revisions made in another part.

➔ **Difficulty in implementation**

Traditionally, code and data have been kept apart. For example, in the C language, units of code are called functions, while units of data are called structures. Functions and structures are not formally connected in C.

The focus of the traditional programming approach tends to be on the implementation details. However, functions and data structures do not model the real world adequately. Humans are not used to thinking in terms of functions and data. The focus of a person's thinking is usually in terms of things or entities, their properties and actions. When we think of an application involving, say, the accounts department of an organization, we think in terms of the following:

- The cashier pays the salaries
- An employee submits vouchers
- The accounts officer sanctions payments
- The cashier tallies accounts

It is difficult to decide how to implement this in terms of data structures, variables and functions. If there were a method by which functions and data at the programming level could correspond to real-world entities and actions, it would simplify the problem. This is where object-oriented programming techniques help.

1.2.2 Introduction to Object-Oriented Programming

The Object-oriented programming is a reaction to programming problems that were first seen in large programs being developed in the seventies. It offers a powerful model for writing computer software. Object-oriented programming allows for the analysis and design of an application in terms of entities or objects so that the process replicates the human thought process as closely as possible.

This means that the application has to implement the entities as they are seen in real life and associate actions and attributes with each. In the object-oriented programming, code and data are merged into a single indivisible thing - an object. Figure 1.2 represents this.

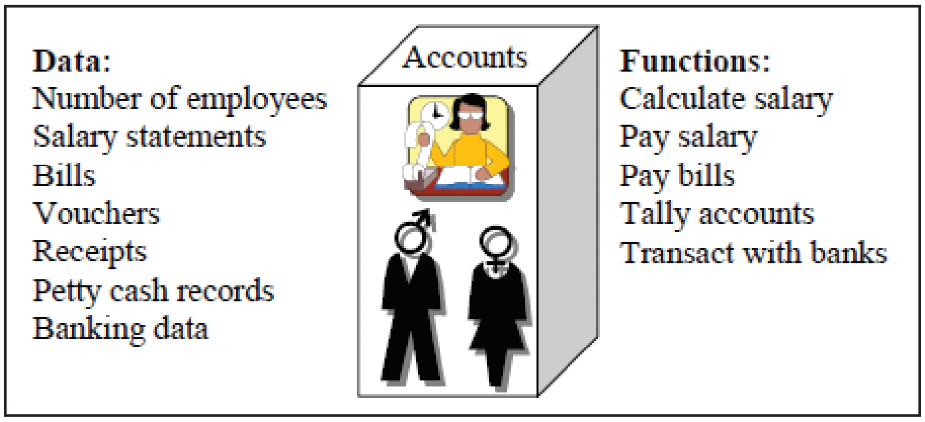


Figure 1.2: Data and Functions of an Object

As we look at various concepts related to object-orientation you will see that there is a close match between objects in the programming sense and objects in the real world.

1.3 Basic Object-Oriented Concepts

There are several concepts underlying object-oriented technology.

1.3.1 Objects

An object represents an entity in the real world. An object is simply something that makes sense in a particular application.

Definition - An object incorporates the combination of the set of data values as data members, and the set of operations as member function.

Objects help to understand the real world.

To see how real-life entities or things can become objects in object-oriented programs let us look at some typical examples listed.

➔ **Physical objects**

- Vehicles in a traffic-monitoring application
- Electrical components in a circuit design problem
- Countries in a global weather model

➔ **Elements of the computer-user environment**

- Windows

- Menus
- Graphics objects
- The mouse and the keyboard

➔ **User-defined data types**

- Complex numbers

Each object has its own properties or characteristics that describe what it is or does. For example the properties of a Person object would be:

➔ Name

➔ Age

➔ Weight

Some properties of a Car object would be:

➔ Color

➔ Weight

➔ Model-year

➔ Number of wheels

➔ Engine power

An object also executes some actions. Figure 1.3 represents some examples of objects.

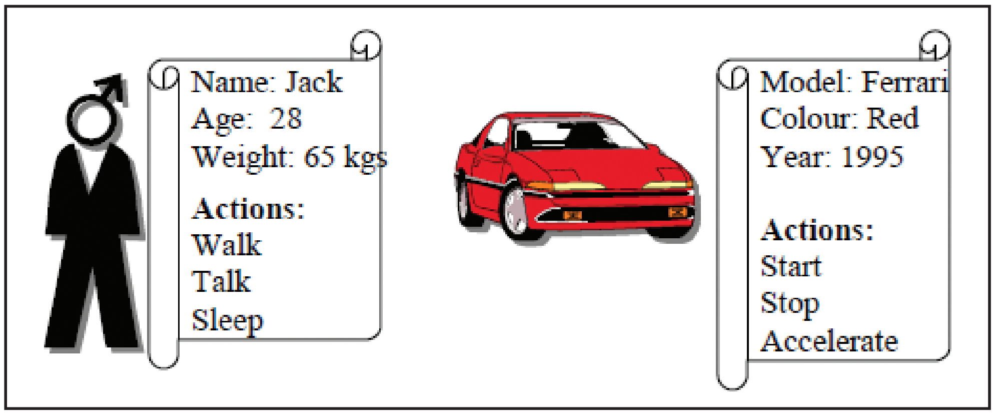


Figure 1.3: A Person Object and a Car Object

The actions a Car is capable of are as follows:

- Start
- Stop
- Accelerate
- Reverse

The match between programming objects and real-world objects is the result of combining the properties and actions of an object.

1.3.2 Classes

Objects with similar properties and actions need to be grouped together into a unit that can be used in a program. Similar objects or objects with common properties, are grouped into a class. Each class describes a set of individual objects. The term class is an abbreviation of "class of objects". A class of persons, class of animals, class of processes, class of polygons and class of window objects are all examples of classes.

Definition - A class is a group of objects that have the same properties, common behavior and common relationships.

The class defines the characteristics that the object is to possess. However, values can be assigned only after an object is created. An actual instance of the entity comes into existence only when the object is created.

Each object is said to be an **instance** of its class. For example, in a class of Polygons, a triangle, a square, a parallelogram, a rectangle and so on, are objects. Figure 1.4 shows some examples.

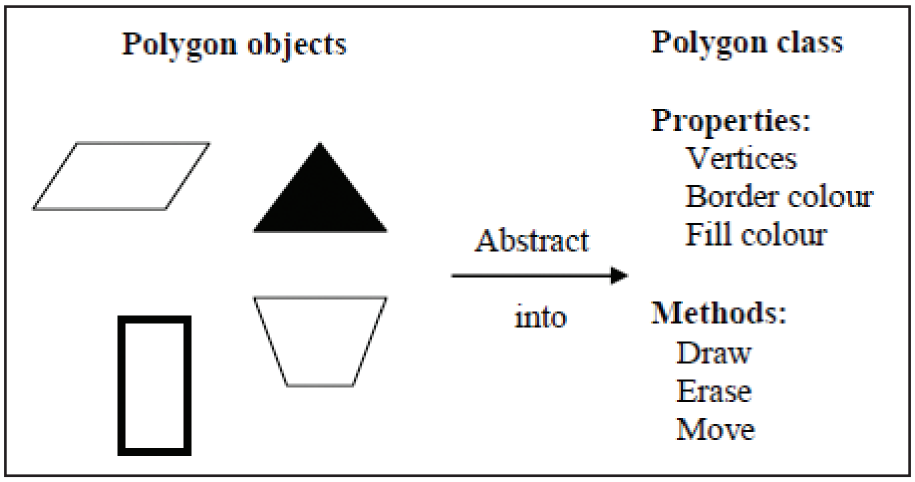


Figure 1.4: Objects and Classes

➔ **Property/Attribute**

The characteristics of the object are represented as the variables in a class and referred to as the **properties** or **attributes** of the class. For example, each polygon in a class has a name, vertices and edges, border color and Fill color. All polygons in the class share these common properties.

Definition - A characteristic required of an object or entity when represented in a class, is called a **property**.

A class is a prototype and not an actual specimen of the entity. Each instance of the class has its own value for each of its properties, but it shares the property names or operations with other instances of the class.

➔ **Method**

The actions that are required of an object have also to be programmatically represented in the corresponding class. All objects in a class perform certain common actions or operations. Each action required, of an object becomes, a function in the class that defines it, and is referred to as a **method**. In the polygon class, "draw", "erase" and "move" are examples of the methods that are part of the class.

Definition - An action required of an object or entity represented in a class, is called a *`method`*.

In the example of an organization, each department performs some actions like:

- Managing the operations of the department
- Filing data
- Sending out memos

These actions or methods are common to each department, apart from specific duties assigned to them.

In the example of an organization, as shown in figure 1.5, each department is considered an object. Information passed to and retrieved from each department in the form of inter-departmental memos or verbal instructions are the messages between objects. These messages can be translated to function calls in a program.

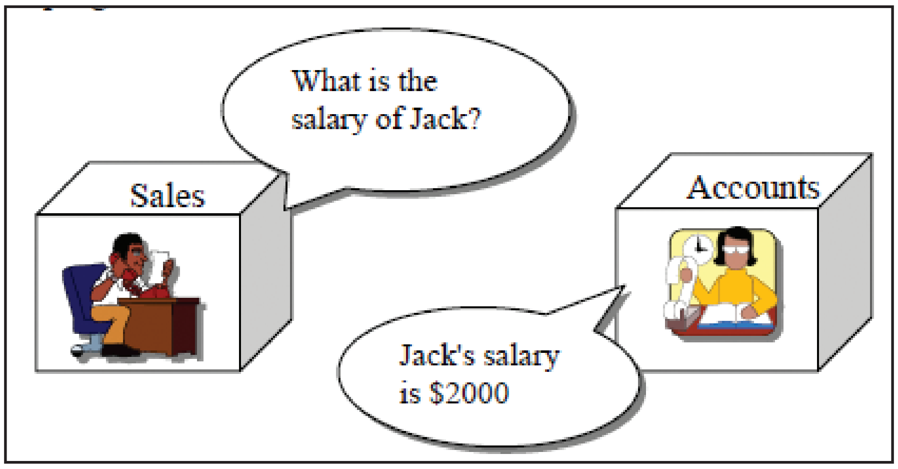


Figure 1.5: Objects Sending Messages to Each Other

It is a fact that the class is such a generalized concept that there are libraries of prewritten classes available in the marketplace. You can purchase classes that perform some generalized operations such as managing stacks, queues, or lists, sorting data, managing windows, etc.

1.3.3 Abstraction

Abstraction is the process of examining certain aspects of a problem. In system development, this means focussing on what an object is and does, before deciding how it should be implemented. An abstract specification tells us what an object does independent of how it works. All object-oriented languages implement data abstraction in a clean way using classes.

Data Abstraction

Data Abstraction is the process of examining all the available information about an entity to identify information that is relevant to the application. Data abstraction is used to identify properties and methods of each object as relevant to the application at hand.

Definition - Data abstraction is the process of identifying properties and methods related to a particular entity as relevant to the application.

By grouping objects into classes, we are in fact performing data abstraction of a problem. Common definitions are stored once per class rather than once per instance of the class. Methods can be written once for a class, so that all the objects in a class benefit from code reuse.

For example, all ellipses share the same methods to:

- Draw them
- Compute their area
- Test for intersection with a line

Polygons would have a separate set of methods. Even special cases, such as circles and squares, can use the general methods. In object-oriented programming data abstraction is defined as a collection of data and methods, which is what an object represents.

1.3.4 Inheritance

The idea of classes leads to the idea of **inheritance**. In our daily lives, we use the concept of classes being divided into subclasses. We know that a class of animals can be divided into mammals, amphibians, insects, reptiles, etc. A class of vehicles can be divided into cars, trucks, buses, and motorcycles. A class of shapes can be divided into lines, ellipses, boxes, and so on.

Definition - Inheritance is the property that allows the reuse of an existing class to build a new class.

The principle in this sort of division is that each subclass shares common properties with the class from which it is derived. For example, all vehicles in a class may share similar properties of having wheels and a motor. In addition, the subclass may have its own particular characteristics. For example, a bus may have seats for people, while trucks have space for carrying goods. The new class **inherits** all the behavior of the original class. The original class is called the **base class**, or **super class**, of the new class. Some more terms - A subclass is said to be a **specialization** of its super class, and conversely, a super class is a **generalization** of its subclasses. Figure 1.6 shows an example.

Definition - The superclass is the class from which another class inherits its behavior. The class that inherits the properties and methods of another class is called the ***subclass***.

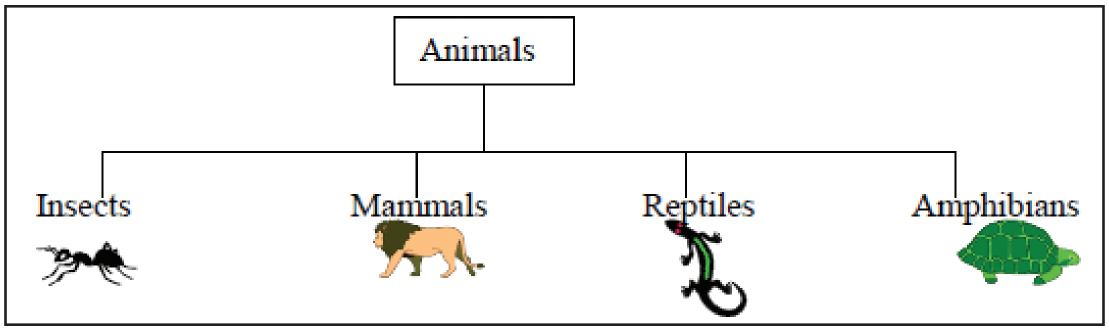


Figure 1.6: Class Animals and its Subclasses

1.3.5 Encapsulation

Using the black box as the example of an object, a primary rule of object-oriented programming is this: as the user of an object, you should never need to look inside the box. A class has many properties and methods. It is not necessary for a user to have access to all of them.

All communication to an object is done via messages. The object, which a message is sent to, is called the receiver of the message. Messages define the **interface** to the object. Everything an object can do is represented by its message interface. So you do not have to know anything about what is in the black box in order to use it. Providing access to an object only through its messages, while keeping the details private is called **information hiding**.

Definition - Encapsulation is the process that allows selective hiding of properties and methods in a class.

Figure 1.7 shows an example of an object interface.

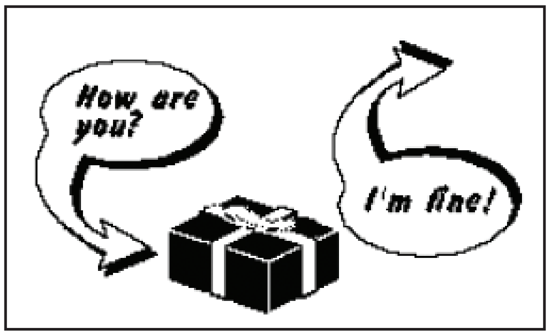


Figure 1.7: An Object's Interface via Messages

The advantage of encapsulation is that a class can have many properties and methods but only some of these need to be exposed to the user. Thus encapsulation may also be defined as:

In the example of an organization there are many methods and properties of the organization that are classified as private. When another organization, say, Company B deals with Company A, it does not interfere with the way in which Company A functions. Any information that Company B needs has to be strictly routed through proper channels such as, the public relations officer or administrative manager of Company A.

Looking at it from the programming angle, by not looking inside an object's black box you are not tempted to directly modify that object. If you did, you would be tampering with the details of how the object works. When we properly encapsulate some code we achieve two objectives:

1. We build an impenetrable wall to protect the code from accidental corruption due to the minor errors that we are all prone to make.
2. We also isolate errors to small sections of code thereby making them easier to find and fix.

Some of the costliest mistakes in computer history have come from software that breaks when someone tries to change it.

1.3.6 Polymorphism

Object oriented languages try to make existing code easily modifiable without actually changing the code. This is a unique and powerful concept, because it does not, at first, seem possible to revise something without changing it. Using **inheritance** and **polymorphism**, it is possible to do just that.

Polymorphism means the ability to assume different forms at different times. The existing object stays the same, and any changes made are only additions to it. Using this approach a programmer is able to maintain and revise code with a smaller number of errors since the original object is not changed.

Definition - Polymorphism means the ability to assume different forms at different times. In simple terms, polymorphism is the attribute that allows one interface to be used with a general class of actions.

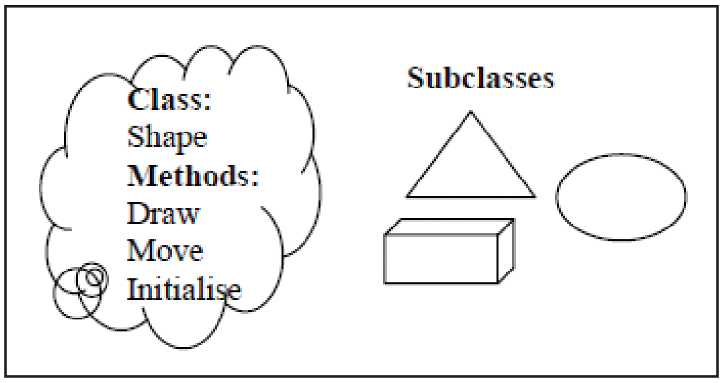


Figure 1.8: Class Shape and its Subclasses

In figure 1.8, you can see that Draw is a method or function, which all the subclasses of the class Shape share. However, the Draw method that is implemented for a box and that for an ellipse will be different.

Polymorphism promotes encapsulation. As far as the user is concerned there needs to be just one method for the class, such as Draw. How the Draw method is implemented for different cases need not be visible to the user.

Object oriented programming requires a major shift in thinking by programmers. This approach speeds up the development of new programs, and, if properly used, improves the maintenance, reusability, and revision of software.

1.4 Structure of a C++ Program

The term structure is synonymous with organization. For instance, while working for a school/college project or a presentation, we need to follow a definite pattern of the organization for our material. This organizational pattern taken up should follow a standard set of guidelines, which are applicable throughout the project. These guidelines could include defining a standard of presentation for all our transparencies and the naming of various documents using a pre-determined prefix or suffix to help classification. Similarly, the task of programming also requires us to follow a pre-defined pattern. This concept of following a pattern to fulfill our objectives is known as structured programming.

C++ is a structured programming language. The basic components or standard guidelines used while writing a C++ program have been discussed later. However, before we get more technical, let us see how a simple C++ program is written (shown in Example 1).

Example 1:

```
#include <iostream.h>

void main()

{

 cout << "Hello there !!!";

}
```

The program in Example 1 prints the following message on your screen:

```
Hello there !!!
```

The program illustrated in example 1 consists of steps given:

- #include... statement
- Declaration of the main function
- Processing statements

1.4.1 # include...Statement

The # **include**... statements are the first statements in any C++ program. They are the most fundamental set of instructions, which is required for writing any C++ program. For instance, before a painter starts working on a new assignment, he needs to verify whether he has the required set of colors and painting brushes to complete his job successfully. Similarly, a programmer should ensure the availability of certain tools before writing a successfully executable program. All programming languages have a set of built-in standard processing tools. These tools are termed as classes and functions in C++. Their exact terminology will be discussed in the later sessions.

The hash notation (#) at the beginning of the **include**... statement indicates that the following instruction is a special instruction to the C++ language. The **#include**... statement establishes a reference to the **header file** (as indicated by the extension **.h**). It is specified in the brackets '< >' following the **#include**... statement and is processed prior to any other processing taking place. It is referred to as a **preprocessor directive**. In Example 1, the header file being referred to is **iostream.h**.

On detecting this directive, the contents of the named header file are inserted into the program at the point of the directive. In other words, the specified file is included at the head of the program to define certain values and symbols (Header files are discussed in detail later in this session.)

1.4.2 The main() Function

A program consists of a single function, **main ()**. A function is a self-contained block of code that is referenced by a name, main in this case. There may be a lot of other code in a program, but every C++ program consists of at least the function **main ()**, and there can be only one function called main within a program. Execution of a C++ program always starts with the first statement in **main ()**. The word **main** is followed by parentheses '**( )**'.

1.4.3 Processing Statements

In Example 1, function main() contains one executable statement:

```
cout << "Hello there !!!" ;
```

This is executed in sequence, starting at the beginning. In general, the statements in a function are always executed sequentially, unless there is a statement that specifically alters the sequence of execution.

1.5 Input and Output

Input is the process of receiving data from an input device; the default being the keyboard and output is the process of transmitting data to an output device, the default being the monitor.

The I/O operations in C++ involve moving bytes from devices to memory and vice versa in a consistent and reliable manner. An example of input and output in C++ has been illustrated in Example 2.

Example 2:

```
#include <iostream.h>


void main (void)
{
 int i_number;
 cout << "Enter a number: "; // Using std output stream
 cin >> i_number; // Using std input stream
 cout << "\n You entered: " << i_number;
}
```

The output of the program in Example 2 is as follows:

```
Enter a number: 12
 You entered: 12
```

This program accepts a number from the user, which is stored in **i_number** and displays the same on the screen performing an input and output operation.

1.5.1 Standard Input and Output Streams

In C++, input and output are performed using streams. To output something, put it into an output stream, and to input something, get it from an input stream. The standard output and input streams in C++ are **cout** and **cin**, and they use computer's screen and keyboard respectively.

C++ associates a buffer to every stream of input and output. A buffer is defined as a storage area in the memory. Input is taken from this buffer and output written to this buffer and hence, I/O, as defined by C++, is buffered. The following objects, in combination with their corresponding insertion (>>) and extraction (<<) operators, perform input and output in C++.

➔ **cin:** This object corresponds to the standard input stream, which by default is the keyboard. The insertion operator (>>) is used with the cin statement for the input to be redirected to the memory.

cin >> i_number ;

➔ **cout:** This object corresponds to the standard output stream, which is the screen by default. The extraction operator (<<) is used with the cout statement for the output to be redirected to the monitor.

cout << i_number ;

1.5.2 Cascading I/O Operators

The input and output streams can be used in combination and this method of using I/O streams is referred to as **Cascading of I/O operators**. The use of cascaded output operators has been illustrated in Example 3.

Example 3:

```
#include <iostream.h> //Inclusion of header files
void main (void) // Declaration of main function
{
int age = 0 ;
float salary = 0.00 ;

// Cascaded output using stream insertion and
// extraction operators.
cout << "\nEnter your age: " ;
cin >> age ;
cout << "\nEnter your salary: " ;
cin >> salary ;
    cout << "\nMy age is " << age ;
    cout << "\nMy salary is " << salary ;
}
```

The first **cout** statement in the program prompts the user to enter a value for **age**. This entry is then stored in the memory and is identified by the name **age**. The next statement accepts input pertaining to the user's salary and stores the same in the memory. The name **salary** identifies this stored data. The program then displays the **age** and **salary** entered by fetching the same from the memory.

Output:

```
Enter your age: 20
Enter your salary: 5000.00
```

```
My age is 20
My salary is 5000
```

Note the following steps of this program:

1. Inclusion of header files
2. Declaration of main function
3. Cascaded output using the stream extraction operators.

1.5.3 Simple Formatting Functions for Input and Output

Output in C++ can be formatted using special characters associated with the **cin** and **cout** statements. These special characters are used in example 4.

Example 4:

```
#include <iostream.h>

void main (void)
{
 cout << "This line uses the end line operator" << endl ;
 cout << "Default stream width = " << cout.width () << endl ;
 cout << "This displays the default stream width - " ;
 cout << " [" << "A" << "] " << endl ;

 cout << "This displays the modified field width - " ;
 cout << " [" ;
 cout.width (10) ;
 cout << "A" ;
 cout << "]" ;
}
```

The output of the program would be:

```
This line uses the end line operator
Default stream width = 0
This displays the default stream width - [A]
This displays the modified field width - [ A]
```

Note the uses of two of the important formatting functions in C++.

`endl` This function inserts a new line i.e. prints the next output on the next line. It clears the current output stream of any existing data.

`width` The width function used by the output stream is used to indicate the current output stream width and is also used to modify the output stream width.

Some examples of these formatting functions are as follows:

```
cout << endl ;
```

This would simply insert a new line in the output without displaying anything on the screen.

```
cout.width(20);

cout << "Hello world";
```

The statements would set the width of the standard output stream to 20 and the message being displayed – **Hello world** - would contain spaces in front of it.

1.6 Essentials

Some of the important essentials for coding the C++ program are being discussed:

1.6.1 Function Definition

C++ programs are divided into code units called functions. Functions are defined to break up large operations into smaller ones. The general format for function definition in C++ is the function name followed by two simple parentheses '( )'. The **main()** statement, discussed earlier, is an example of a function.

1.6.2 Delimiters

After the function definition are the braces, which signal the beginning and the end of a function. The opening bracket or the left brace ({) indicates that a code of block is about to begin and the closing bracket or the right brace (}) indicates that the block of code is terminated. Curly braces are also used to delimit blocks of code in other situations like loops and decision-making statements.

The use of delimiters has been illustrated in Example 5.

Example 5:

```
#include <iostream.h>


main()
{ // Opening bracket
 cout << "Explaining delimiters";
} // Closing bracket
```

The output of the program in Example 5 is:

```
Explaining delimiters
```

1.6.3 Statement Terminator

Every code instruction in C++ must be terminated with a semicolon (;). There can be more than one statement on the same line as long as each one of them is terminated with a semi-colon. This form of programming however is not recommended as it makes code maintenance, tedious. A code instruction, which does not end with a semicolon, is treated as an erroneous.

To output a simple statement in C++, we need to use the **cout**. The code is as follows:

```
      cout << "Output a simple text statement" ;
```

1.6.4 Comments

Comments are an integral part of any program. Comments help in coding and maintaining a program as they increase the readability of a program. They make no difference to the actual logic of the program since sections specified as comments are simply ignored at run-time. Comments therefore, do not slow down the execution speed nor do they increase the size of the executable program. Comments should be used wherever necessary.

In C++, a comment starts with two forward slash symbols (//) and ends with the end of that line. A comment can start at the beginning of the line or on a line following a program statement. These comments are called end-line comments. If a comment continues on more than one line, the two forward slash symbols should be given on every line. Alternatively the multi-line comment symbol should be used. The multi-line comment initiation symbol consists of a forward slash followed by an asterisk (/*), which is placed at the beginning of the comment, and the multi-line comment terminator symbol consists of an asterisk followed by a forward slash (*/), which is placed after the end of the comment. All the information placed between these two notations is considered as comment text.

Example 6 illustrates the use of comments in a program.

Example 6:

```
#include <iostream.h>
void main(void)
{ //Opening bracket or delimiter

 /*   This is the style of giving multi-line comments in

 C++. It can be continued on the next line and has to
 be explicitly ended with a multi-line comment
 terminator. */

 // This comment on
 // multiple lines is also
 // correct
 // This is a single-line comment

cout<<"Hello world" ; // This is an end-line comment
} // Closing bracket or delimiter
```

The output of the program would be:

```
Hello world
```

Note that the compiler ignores the sections, which are commented, at the time of execution and only the statement beginning with cout is processed.

1.7 Private, Protected, and Public Sections of a Class

Class members can be declared in the public or protected or the private sections of a class. As one of the features of object-oriented programming is to prevent data from unrestricted access, the data members of a class are normally described in the private section. Information hiding is the insulation of data members from direct access in a program.

The public part constitutes the **interface** to objects of the class. The data members and functions declared in the public section can be accessed by any function in the outside world (outside of the class).

Figure 1.9 illustrates the accessibility rules for **private**, **protected** and **public** data member.

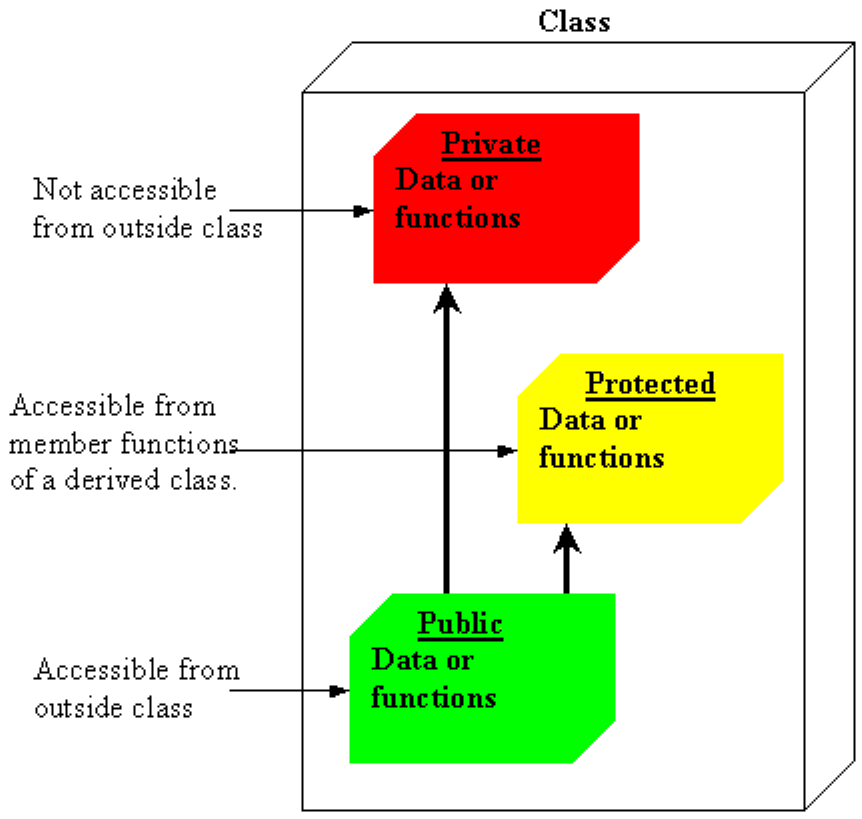

Class
Private
Data or functions
Protected
Data or functions
Public
Data or functions
Not accessible from outside class
Accessible from member functions of a derived class.
Accessible from outside class

Figure 1.9: Private, Protected, and Public Access Specifiers

You can do anything with the private data within the function implementations that are a part of that class. You should keep in mind that the private data is not available outside the class to any program that might need to use the data member. Also, the private data of other classes is hidden and not available within the functions of this class.

Members of a class that are protected can only be accessed by the member functions of the class, friend classes and friend functions of the class. These protected class members can't be accessed from outside the class, so they behave like private class members. Members of a base class that are declared protected are freely accessible in function members of a derived class, whereas the private members of the base class are not.

It is valid to declare variables and functions with a private modifier, and additional variables and functions in the public part also. In most practical situations, variables are declared in only the private part and functions are declared in only the public part of a class definition.

1.8 Member Functions

A message to an object corresponds to a function call. Messages can have parameters. In response to a message, the object executes a **method**, which is similar to a procedure body. It may or may not return a result. A method is a function contained within the class. In object oriented programming, we send messages instead of calling functions.

A **member function**, is a function declared as a member of a class. There are two member functions in the class date shown:

➔ `setdate(int, int, int)`

➔ `showdata()`

Member functions are usually put in the public part of the class because they have to be called outside the class either in a program or in a function.

As a class contains not only a data member but also a function or methods, it must be defined before it is to be used. The declaration of a member function must define its return value as well as the list of its arguments. For example, the member function `void setdate (int, int, int)` has no return value or a void return value, and has three integer arguments.

Member functions can be more complex as they can have local variable, parameters etc. Member functions should not have the same name as a data member.

There are several benefits from restricting access to a data structure to an explicitly declared list of functions. One of the benefit is that a potential user of such a type needs to only examine the definition of the member functions to learn to use it.

1.9 Using the Class

Now that we know how a class is specified, let us see how objects are defined, and once defined, how their member functions are accessed. Let us consider Example 7.

Example 7:

```
// This is a simple program
class exampleclass{              // specify a class
 private:
int object_data; // class data
 public:
member_function1(parameter1, parameter2…)
{
……//assign value to object_data
}
member_function2()
{
……. //display data
}
};
main()
{
 exampleclass object1,object2;//define the objects of class
 object1.member_function1(200);// call member function to
 //assign value 200 to
 // object_data
 object1.member_function2(); //call member function to
 // display data
 object2.member_function1(350);
 object2.member_function2();
}
```

1.9.1 Defining Objects

The first statement in the main program,

```
exampleclass object1,object2;
```

The statement defines two objects, **object1** and **object2**, of the class **exampleclass**. Remember that the specification for the class, **exampleclass**, does not create any objects. It only describes how they will look when they are created in memory. The definition creates objects physically in memory that can be used by the program. Defining an object is similar to defining a variable of any data type. When we define an object, space is set-aside for it in memory.

1.9.2 Calling Member Functions

We communicate with objects by calling their member functions. The next two statements in the main program shown earlier call the member functions:

```
object1.member_function1(200);

object1.member_function2();
```

A member function is always called to act on a specific object, not on the class in general. A member function is associated with a specific object with the dot operator (the period). The general syntax for accessing a member function of a class is:

```
class_object.function_member()
```

The syntax is similar to the way we refer to structure members, but the parentheses indicate that we are executing a member function rather than referring to a data item. The dot operator is called the class member access operator.

The first function call,

```
object1.member_function1(200);
```

assigns the value 200 to object_data of the `object1`. The second call causes the object to display its value. Similar functions are defined and executed for `object2`. Figure 1.10 depicts this.

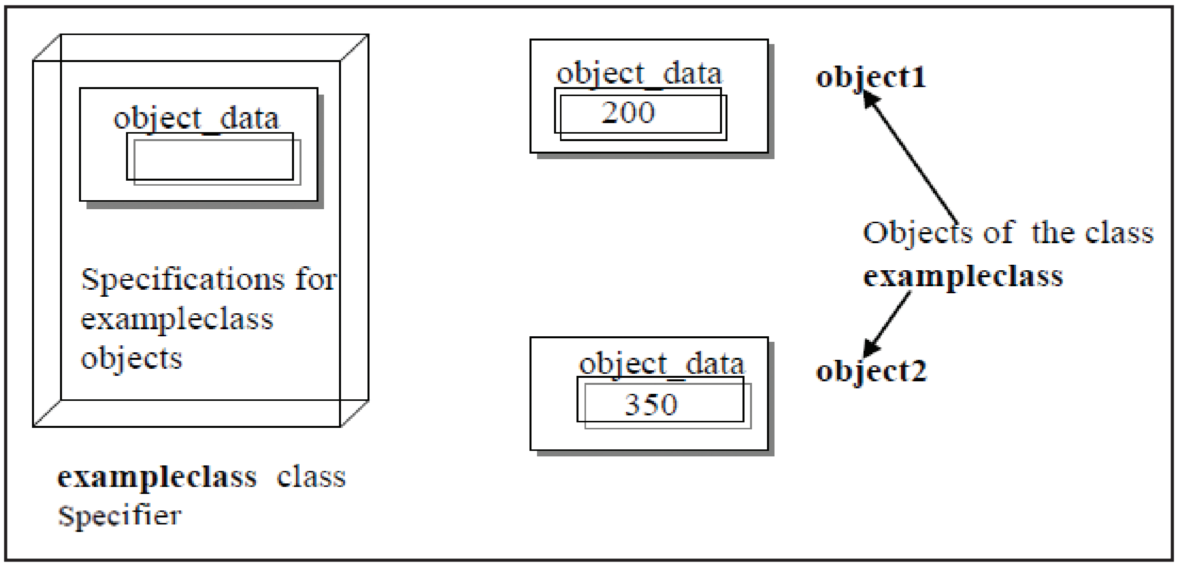

object_data
Specifications for exampleclass objects
exampleclass class Specifier
object_data
200
object1
Objects of the class exampleclass
object_data
350
object2

Figure 1.10: Two Objects of a Class with Different Values

1.9.3 Passing and Returning Objects

Objects can be passed to a function and returned back just like normal variables. When an object is passed as an argument in a function, the compiler creates another object as a formal variable in the called function. It copies all the data members from the actual object. Look at the program given in the Example. This program passes an object to a function.

Example 8:

```
u.get_age(21); // This affects only the local copy
 u.display(); // Outputs 21
}
main()
{
test p;
p.get_age(25);
func(p);
p.display(); // Still outputs 25, value of age unchanged
return 0;
```

```
 }
 u.get_age(21); // This affects only the local copy
 u.display(); // Outputs 21
 }
main()
{
test p;
p.get_age(25);
func(p);
p.display(); // Still outputs 25, value of age unchanged
return 0;
}
```

Just as it is possible for a function to return an integer value it is also possible for the function to return an object. A return statement in a function is considered to initialize a variable of the returned type. The syntax is as shown in the function call:

```
classname object3 = object1.function1(object2);
```

where the function would be defined as,

```
classname fucntion1(classname obj)
{
 classname temp_object;
 .
 .
 return temp_object;
}
```

The function has a local object, `temp_object`, which is returned to the calling function and `obj` is an object of the class `classname`. The program illustrates how an object can be returned by a function.

Example 9:

```
#include<iostream.h>
#include<conio.h>
class test{
private:
char name[80];
public:
void display()
{
cout<<name<<"\n";
}
void getname(char *st)
{
strcpy(name, st);
}
};
// Return object of the type test
test takename()
{
char getstr[80];
test st;

cout<<" Enter your name: ";
cin>> getstr;
st.getname(getstr);
return st;
}
main()
{
test obj;

//Assign returned object to obj
obj=takename();
obj.display();
return 0;
}
```

Passing and returning of objects is not very efficient since it involves passing and returning a copy of the data members.

1.10 Using the Borland C++ Editor

The Borland C++ interface is a simple character based interface supporting different color keys for different functions. In case of monochrome monitors, these color differences are reflected in different shades of grey.

Figure 1.11 shows the Borland C++ editor interface.

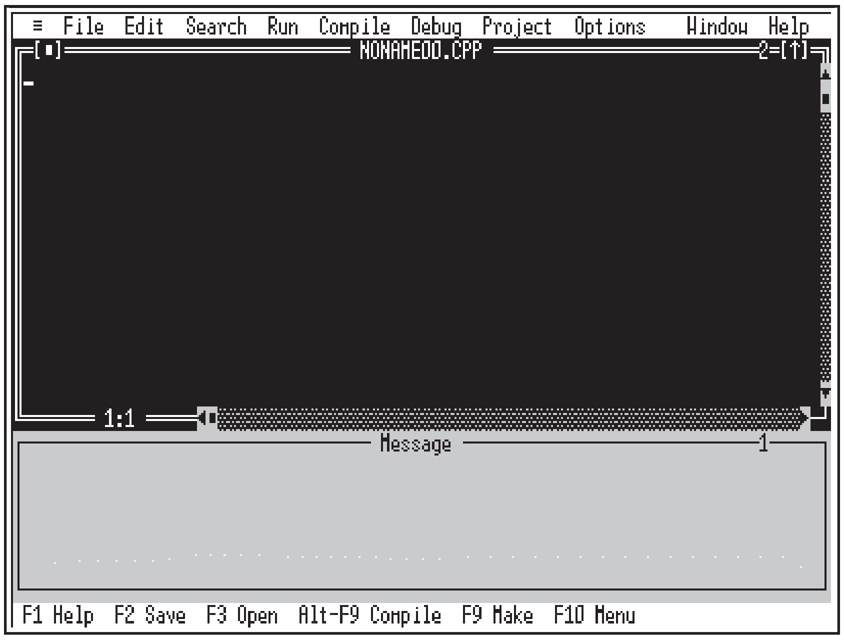
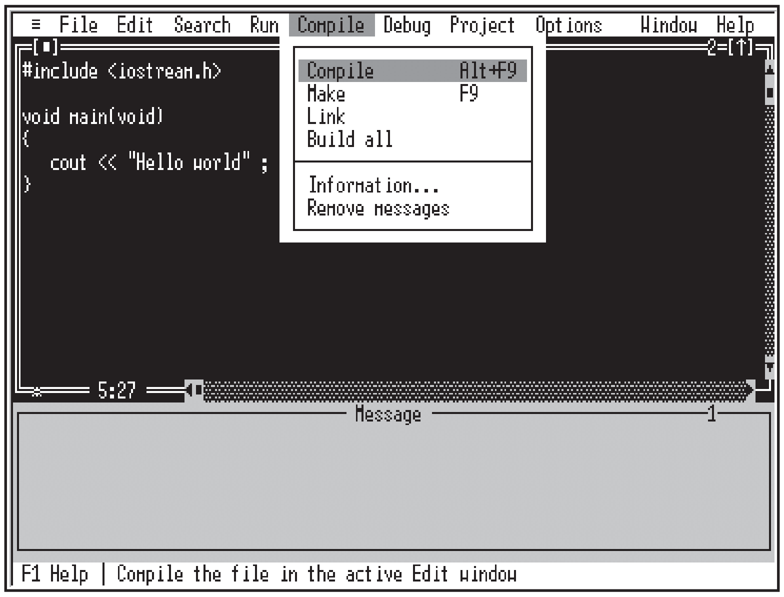

≡ File Edit Search Run Compile Debug Project Options Window Help
NONAME00.CPP
2
1:1
Message
1
F1 Help F2 Save F3 Open Alt-F9 Compile F9 Make F10 Menu

Figure 1.11: Borland C++ Editor Interface

A C++ program is first written in the editor. This is called the source code. This form of the program is written in terms, which are meaningful and understandable to the programmer. These understandable terms are referred to as the C++ syntax. The machine cannot interpret the source code, as it understands only zeros and ones (binary code). Hence, the source code needs to be converted into binary code. For this, the programmer needs to compile the source code, which generates the binary code required for the machine to understand the instructions. Once the binary code for the program has been generated the program needs to be linked into a single executable block. Once this process of linking is complete the program is ready to execute or run.

1.10.1 Compilation

The Borland C++ compiler translates the source code to machine language that produces the actions specified by the source code. If a program is too large to be contained in one file, it can be put in separate files and each of the files can be compiled separately. The advantage of separate compilation is that if code in a file is changed, the entire program need not be recompiled.



```
#include <iostream.h>

void main(void)
{
   cout << "Hello world" ;
}
```

≡ File Edit Search Run Compile Debug Project Options Window Help
Compile Alt+F9
Make F9
Link
Build all
Information...
Remove messages
5:27
Message
F1 Help | Compile the file in the active Edit window

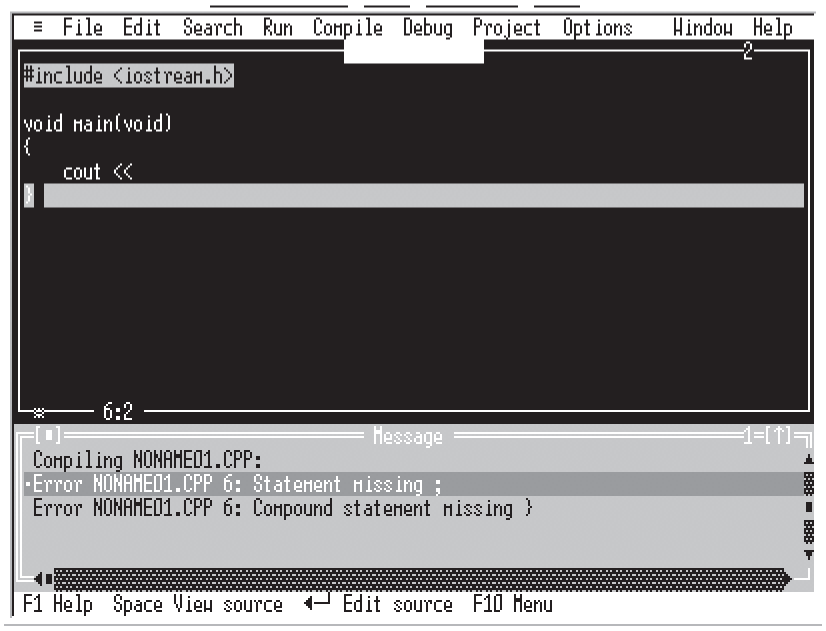
Figure 1.12: Compile Option

The **Compile** option under the **Compile** menu shown in figure 1.12 compiles the active editor file (either a .C or .CPP file) and creates an object file (with the .OBJ extension). While Borland C++ is compiling, a status box pops up to display the status of compilation in terms of lines compiled, errors or warnings applicable. Once the compilation is over, the **Message** window becomes active and lists the errors or warnings generated by the compiler. To return to the code window hit the **Enter** key after selecting the error or warning listed in the **Message** window. To use the compile command, at least one code set window should be open in the editor.

1.10.2 Error Messages

As discussed, error messages and warnings generated by the compiler at run time are displayed in the message window. On selecting the desired error or warning message from the list in the message window (use the arrow keys to scroll and enter to select), the editor takes your cursor to the position of error in your instruction set. Before generating an executable program for a given set of code, all the errors must be rectified. However, a warning can be ignored and the executable code can be generated.

An example of an error scenario is shown in figure 1.13.


≡ File Edit Search Run Compile Debug Project Options Window Help
2
6:2
Message
1=[↑]
F1 Help Space View source ↵ Edit source F10 Menu

```
#include <iostream.h>

void main(void)
{
    cout <<
}
```

```
Compiling NONAME01.CPP:
•Error NONAME01.CPP 6: Statement missing ;
 Error NONAME01.CPP 6: Compound statement missing }
```

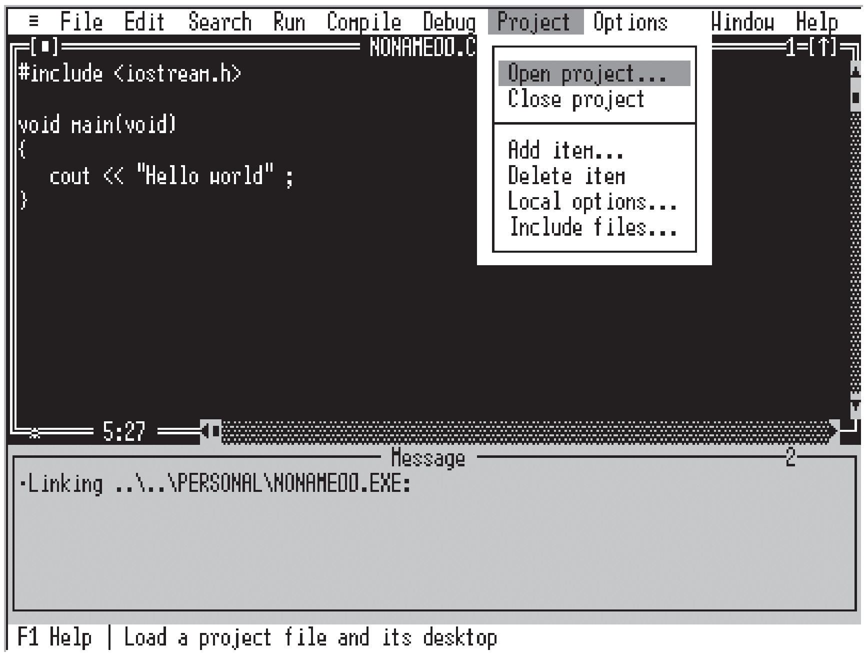
Figure 1.13: Example of Error

In figure 1.13, a code statement has been written without using the statement terminator. This results in an error being generated. This error is shown in the **Message** window as shown in figure 1.13. The line of code containing the compilation error is highlighted in the message window. The active window shown in figure 1.13 is the **Message** window listing the specified error.

1.10.3 Linking

The source code along with support routines from the function libraries and any other separately compiled functions are linked together by the C++ linker into an executable code, which is the *.EXE* file.

The **Link** command in the **Compile** menu links all the different object code files as specified in the **Include Files** option from the **Project** menu (refer Figure 1.14).



```
#include <iostream.h>

void main(void)
{
   cout << "Hello world" ;
}
```

```
•Linking ..\..\PERSONAL\NONAME00.EXE:
```

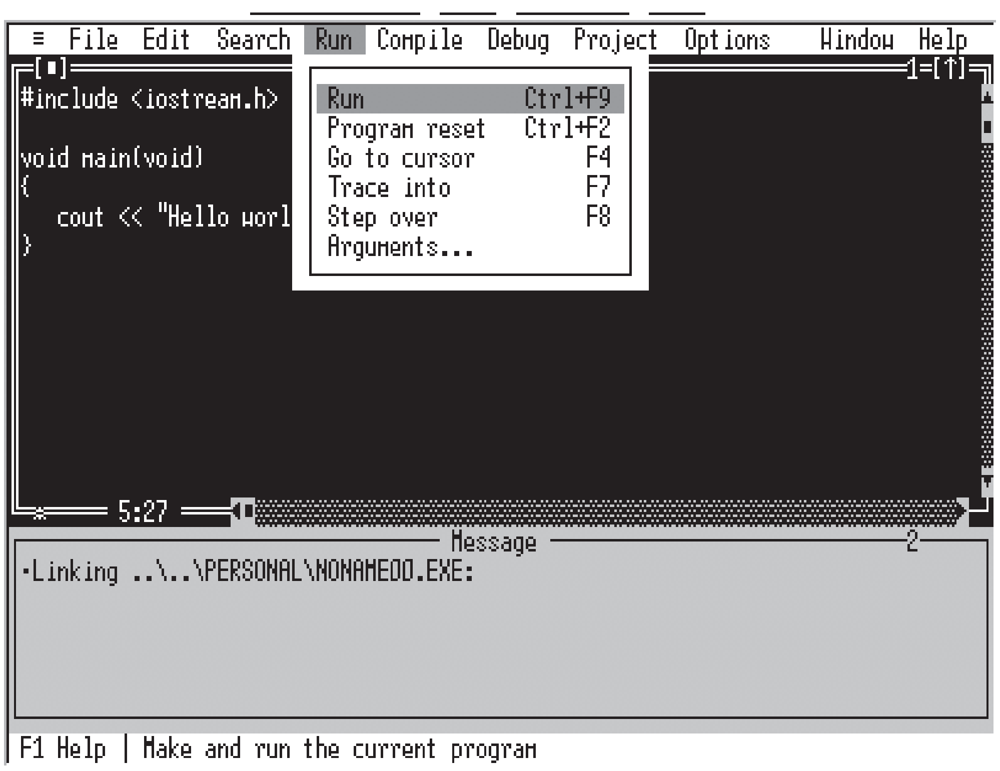
F1 Help | Load a project file and its desktop

Figure 1.14: Project Menu

The compiled code of all the functions defined in the header files is present in their corresponding library files. A library file is identified by the **.lib** extension. The **LIB** sub-directory of the C++ sub-directory contains all the necessary **.lib** files. To use a specific header file, its compiled code should be made available in this sub-directory.

1.10.4 Execution

We can run the executable code of a program by using the C++ system loader. The **Run** option from the **Run** menu carries out the action of compiling, linking and executing the program. As shown in figure 1.15, this can be done by pressing the CTRL + F9 keys as well. Once the program completes execution, the control passes back to the editor.


≡ File Edit Search Run Compile Debug Project Options Window Help
Run Ctrl+F9
Program reset Ctrl+F2
Go to cursor F4
Trace into F7
Step over F8
Arguments...
5:27
Message
F1 Help | Make and run the current program

```
#include <iostream.h>

void main(void)
{
   cout << "Hello worl
}
```

```
•Linking ..\..\PERSONAL\NONAME00.EXE:
```

Figure 1.15: Run Menu

1.11 Check Your Progress

1. In the object oriented way of thinking, software can be organized as a collection of objects that combine both ___________ and __________.
2. Combining data and functions together as a single entity is called ________________.
3. Camel is to animal as object is to ____________.
   a. a member function
   b. a class
   c. an operator
   d. a data item
4. An object is an instance of the class. (True / False)
5. Protecting data from access by unauthorized functions is called ________________.
6. In structures all functions and data, by default, are ____________ but __________ in classes.
7. In a class specifier, data or functions designated private are accessible
   a. to any function in the program
   b. to member functions of that class
   c. only to public members of the class
   d. only if you know the password
8. In C++, a function contained within a class is called _____________.
9. Data items in a class must be private. (True / False)
10. To use a member function, the ____________ connects the object name and the member function.
11. Sending a message to an object is the same as _________________________.
12. C++ is considered a ____________ of C.
13. The #include statement is used to specify a ________________ directive.
14. Every C++ program must have a ________________ function.
15. Header files are files that have the extension ________________.
16. The C++ object corresponding to the standard input stream is ________________.
17. The C++ object corresponding to the standard output stream is ________________.
18. Every code instruction in C++ should be terminated using the statement terminator indicated by the notation ________________.

1.12 Do It Yourself

1. Prepare a list of objects that would be part of an Automobile system.
2. Discuss what the objects in the following list have in common:

   Bicycle, sailboat, car, truck, aeroplane, glider, motorcycle, horse
3. Name the classes the following objects can be grouped into:
   a. File, directory, filename, ASCII file, executable file
   b. Expression, constant, variable, function, statement, arithmetic operator
   c. Sink, freezer, refrigerator, bread, cheese, door, cabinet, mop
4. Write a class specifier, in C++, that creates a class called `Employee` with one private data member, salary, of type `int` and one public function whose declaration is `void pay()`.
5. Write a statement that defines an object called Officer of the Employee class described in question 4.
6. Write a statement that executes the `pay()` function using the Officer object, as described in question 4 and 5.
7. Write a member function called `getinfo()` for the Employee class described in question 4 and 5. This function should return the value of the salary data.
8. If three objects of a class are defined, how many copies of that class's data items are stored in memory? How many copies of its member functions?
9. Write a program that prints your name and address.
10. Write a program to display the following text:

    Twinkle twinkle little star,

    How I wonder what you are,

    Up above the world so high,

    Like a diamond in the sky.
11. Write a program to set the width of the output stream to 20, 10 and 5 for different output statements. Display strings enclosed in square brackets to illustrate the width of the string after altering the stream width.
12. Key in the following program and identify the mistakes in the code:

```
 #include <iostream.h>
void main(void)
{
 cout << "He who laughs last," endl ;
 cout << "Laughs loudest."
}
```

The output of this program should be

```
He who laughs last,
Laughs loudest.
```

Correct the mistakes in the code to give the correct output.

13. Consider the following program:

```
#include <iostream.h>
void main(void)
{
}
```

    a. Can any of the code statements be omitted from the code.

    b. Will the program work?

    c. What will be the output of the program, if any?

14. Consider the following program:

```
 #include iostream.h
 void main(void)
  {cout << "Welcome to the world of C++ programming." };
```

    a. Will the program compile and run successfully?

    b. What will be the output if it does run?

    c. Modify the code to give the following screen output:

```
Welcome to the world of C++ programming.
```

15. Key in the following program:

```
#include <iostream.h>
void main(void)
{
```

cout << “Is there a mistake here ?” ;

a. Compile the program. What are the errors?

b. Correct the program.

16. Consider the following code:

```
void main(void)
{
 cout << “Help ! I keep getting stuck ;
}
```

a. What are the errors in this code?

b. Correct the code to get the following output:

Help ! I keep getting stuck

Summary

- Object-oriented programming allows for the analysis and design of an application in terms of entities or objects. In object-oriented programming, code and data are merged into a single indivisible thing -- an object.

- Similar objects, objects with common properties, are grouped into a class. Each class describes a set of individual objects.

- Each object is said to be an instance of its class. Objects in a class have the same properties and common behavior and common relationships to other objects.

- **Abstraction** consists of focussing on what an object is and does, before deciding how it should be implemented. **Inheritance** is the property that allows the reuse of an existing class to build a new class.**Encapsulation** is the process that allows selective hiding of properties and methods in a class.

- Polymorphism means that object belonging to a 'family' of inheritance-related classes can be passed around and operated on using base class pointer and references.

- Usually the functions in a class are public. The members and operations declared in the public section can be accessed by any function outside of the class.

- In C++, a function declared as a member of a class is called a member function. Member functions are usually given the attributes of public because they have to be called outside the class either in a program or in a function.

- C++ is an Object-Oriented programming language, which was developed as an improvement over the C language.

Session -2

Basics of C++

Welcome to the Session, **Basics of C++**.

This session introduces basics of C++. The session describes diffrent identifiers and keywords in c++. Further, the session describes the various data types and qualifiers. Finally, the session discusses on C++ variables, Operators and type casting.

At the end of this session, you will be able to:

- ➔ Identify the C++ character set
- ➔ Discuss the identifiers and keywords
- ➔ Explain the various data types and qualifiers
- ➔ Identify the C++ variables
- ➔ Identify the C++ operators
- ➔ Understand the precedence and order of evaluation
- ➔ Discuss mixed mode expressions and implicit type conversions
- ➔ Understand type casting and type compatibility
- ➔ Identify C++ shorthand operators

2.1 Introduction

In our previous chapter, we have discussed what a program is and a programming language and the different components of the C++ programming language. With reference to the components of the C++ language, we shall discuss an essential sub-topic referring to data storage in programs during the course of this session.

During the execution of a program, data storage is done by using variables of specific data types.

Let us take an example where you have a program, which accepts your name and salary and calculates the taxation for a given period. This program needs to store the details, which have been input, somewhere in the computer's memory before the program processes the same. These areas where the program stores these details are known as variables. These variables are referred to by their names as defined by the programmer. These names are technically known as **identifiers**. The type of data being stored can be simple alphabets (known as alphabetic) or numbers (known as numeric) or a combination of both (known as alphanumeric). This variation in the type of data being stored requires the programmer to classify memory areas or variables into different data types. This classification makes these memory areas compatible with the specified type of data. Variables are hence, primary memory areas where data is stored during the execution of a program. The primary memory of the computer is used for temporary storage of data.

Hence, variables and data types form one of the most important part of any programming effort. In short, variables are data storage constructs defined to handle numerical, alphabetical or alphanumerical data, which are used for storing data in a temporary form at the time of processing. Depending upon the type of processing the variable is supposed to address, we specify its data type. In other words, the data type defines the type of data which a given variable or constant shall contain. This chapter discusses the various fundamentals pertaining to these components of C++ programming.

2.2 C++ Character Set

Every language has a basic set of alphabets and numbers, which define its vocabulary. We learn these basic elements of any language in our first few years of schooling. Similarly, the C++ language uses a basic set of elements, which is known as **C++ Character Set**.

C++ uses the uppercase letters A to Z, the lowercase letters a to z, the digits 0 to 9, and certain special characters as the building blocks used to form the basic program elements. The special characters used are as listed:

[ ]	( )	.	->	++	--	&
*	+	-	~	!	\	/
%	<<	>>	<	>	<=	>=
==	!=	^	&&	|	||	?:
=	*=	/=	%=	+=	-=	<<=
>>=	&=	^=	|=	,	#	##

C++ uses certain combinations of these characters, such as **\b**, **\n** and **\t**, to represent special conditions such as backspace, newline and horizontal tab respectively. These character combinations are known as escape sequences. The use of these **escape sequences** has been illustrated in Example 1.

Example 1:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 clrscr() ; // This function is used to clear the screen
 cout << "Using the newline escape sequence.\n" ;
 cout << "To use the backspace escape sequence, press enter" ;
 getch() ; // This function is used to pause for user entry
 cout << "\b\b\b\b\b\b\b\b\b\b\bUsed backspace" ;
 cout << "\nUsing the tab sequence\t\t\t\t tab here" ;
 getch() ;
}
```

The output of the program would be:

```
Using the newline escape sequence.
To use the backspace escape sequence, press enter
```

When the enter key is pressed in the keyboard:

```
Used backspace
Using the tab sequence
tab here
```

Example 1 shows the use of the new line escape sequence. To see the action of the backspace escape sequence, press enter after you receive the screen prompt:

```
To use the backspace escape sequence, press enter
```

On pressing enter, the backspace escape sequence is used to move back 11 characters on the screen. Hence, the screen will now display the message,

```
To use the backspace escape sequence, Used backspace
```

where, the text "press enter" will be overwritten by "Used backspace".

While using the tab sequence, the following output is sent to the screen

Using the tab sequence tab here where the display is split by a set of four tabs which are indicated by the \t escape sequence.

Given in table 2.1 is a list of the possible escape sequences supported by C++.

Escape sequence	Character represented
\b	Backspace
\n	Newline
\t	Tab
\r	Carriage return
\f	Form feed
\'	Single quote (apostrophe)
\\	Backslash
\a	Bell
\010	Octal number given by 10
\xhh	Hexadecimal number given by hh

Table 2.1: Escape Sequences in C++

2.3 Identifiers

The names of variables, functions, labels, and various other user-defined references are called identifiers. These identifiers consist of one or more characters. It is necessary that the first character of an identifier is an alphabet (A-Z or a-z) or an underscore (_). The subsequent characters can be alphabets, numbers or underscores.

It is necessary to note the importance of meaningful identifiers in C++ programming. The following factors should be kept in mind while naming identifiers:

➔ **Naming Identifiers**

- A standard naming convention should be followed for all the identifiers throughout a program. This makes referring to the identifiers easier at a later stage. This convention may be defined on several grounds such as attaching a particular prefix or suffix to the specified identifier name.
- The identifier name should adequately define the purpose of its existence within the system. It is not advisable to use single character identifier names such as u, i, j, and so on. for the sake of programming. Such variables may be easier to work with in the immediate frame of work. As the system grows in size, the importance of these variables is often forgotten and future reference to these variables becomes very difficult.

➔ **Scope of identifiers**

The part of the program in which an identifier is valid defines its scope. An identifier is recognized in the code block within which it is declared and all the other enclosed blocks. If an identifier is declared outside any function, its scope becomes **global**, that is, it is valid through the entire program file following its declaration.

2.4 Keywords

Keywords are words that are a part of the C++ language and should be used only in the context as defined. Such names cannot be used as user-defined identifiers as each of these keywords has a specific function to perform in C++. Following is the list of keywords, which have been inherited from the C language:

auto	extern	short	break	float	sizeof
case	for	static	char	goto	struct
continue	if	switch	default	int	typedef
do	while	long	union	double	register
unsigned	else	return	const	enum	signed
volatile	void				

The following are the additional keywords, which have been added in C++:

class	friend	virtual	inline	private	public
protected	this	new	delete	operator	

The actual use and description of all the keywords stated will be covered in their specific contexts.

2.5 Data Types and Variables in C++

The data type is one of the fundamental properties of a variable. It may be considered as the definition of the set of values that a given variable can accept.

C++ data types can be divided into two groups:

➔ Built-in or language defined

➔ User-defined, by using a programming construct. C++, thus allows one to create even complex aggregate data types.

Figure 2.1 depicts data types.

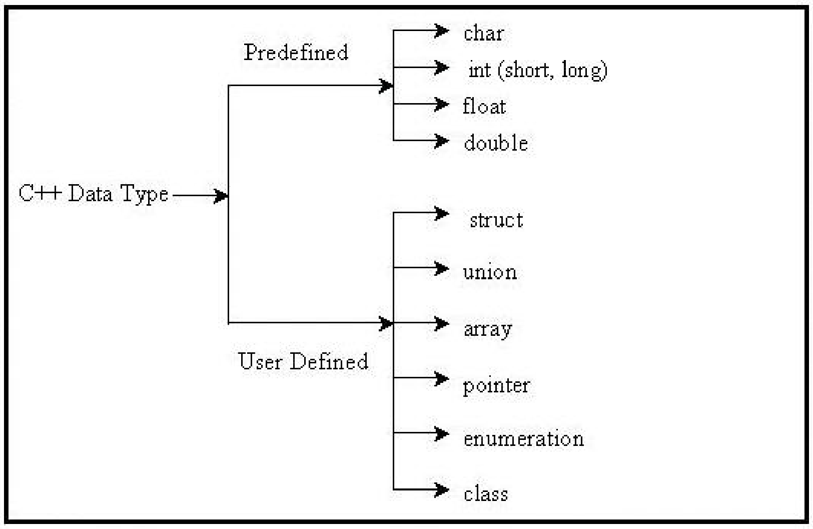


Figure 2.1: Data Types in C++

For the sake of simplicity, we shall only discuss pre-defined / built-in data types right now. The user-defined data types shall be discussed in detail in the following sessions.

2.5.1 Predefined/Built-In Data Types and Associated Variables

Predefined / built-in data types are basic or elementary system provided data types. Built-in data types in C++ are called arithmetic data types. They represent characters, integers and floating-point numbers. There are two particular aspects about these data types. They are:

➔ Some data types have multiple definitions, depending on the precision of computations and the range of numbers required to be computed.

➔ Similarly, our computing needs also require us to use data of varying precision.

This provides you with a dual advantage of achieving greater memory related efficiency on one hand and a higher computational precision in calculations on the other.

Table 2.2 represents a list of the pre-defined data types in C++ along with their characteristics:

Type	Description	Length
char	Single letter character entries only	8 bits
`int`	Integer values only	16 bits
long	Accepts higher range of integer values than the `int` data type	32 bits
float	Supports decimal point notation	32 bits
double	Supports decimal point notation with greater precision than float data type	64 bits
Long double	Similar to a float, with greater precision to the right of the decimal point	128 bits

Table 2.2: Pre-defined Data Types in C++

The structure of a simple variable declaration in C++ is as shown

```
data_type variable_name [ = initial_value ]
```

where the square brackets indicate the assignment of an initial value to a given variable. The initialization of a variable is optional. Thus, we can declare variables as follows:

```
// Declaring integer variable number and initializing it to value 7.
int   number = 7;

//Declaring integer variable number without initializing
int   number ;

// Define num1, num2 and num3 & initialize num3 to value 3.
int   num1, num2, num3 = 3 ;

// Initializing num1, num2, num3 to the value 3 in a single statement
int   num1 = 3, num2 = 3, num3 = 3 ;

// Declaring and initializing character variables
char  character = 'W' ;
char  alpha ;
char  alpha='G', beta='M' ;

//Declaring and initializing float and double variables

float  floating_point_number = 1234.12345678 ;
float  f_number ;

double     double_number = 1234.121213212;
double d_num ;
```

All the pre-defined or built-in data types in C++ have definite domains having assignment limits within which they function.

The program in Example 2 illustrates the use of various data types, their initialization and modification methods:

Example 2:

The program in example 2 illustrates the use of integer, character, float and double variables. These variables are initialized at the time of declaration. The initial values of these variables are displayed on the screen and the user is asked to enter the required modification. The modified values are then assigned to the respective variables and the same are displayed on the screen.

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 int i_num = 123 ;
 char c_val = 'B' ;
 float f_val = 123.8997 ;
 double d_val = 456.890977 ;

 cout << "The current integer value = " << i_num << endl ;
 cout << "Enter new value: " ;
 cin >> i_num ;
 cout << "\nNew value: " << i_num ;

 cout << "\n\n" ;
 cout << "The current character value = " << c_val << endl ;
 cout << "Enter new value: " ;
 cin >> c_val ;
 cout << "\nNew value: " << c_val ;

 cout << "\n\n" ;
 cout << "The current float value = " << f_val << endl ;
 cout << "Enter new value: " ;
 cin >> f_val ;
 cout << "\nNew value: " << f_val ;
 cout << "\n\n" ;
 cout << "The current double value = " << d_val << endl ;
 cout << "Enter new value: " ;
 cin >> d_val ;
 cout << "\nNew value: " << d_val ;
}
```

The program in example 2 illustrates the use of integer, character, float and double variables. These variables are initialized at the time of declaration. The initial values of these variables are displayed on the screen and the user is asked to enter the required modification. The modified values are then assigned to the respective variables and the same are displayed on the screen.

The output of the program in Example 2 is as follows...

```
The current integer value = 123
Enter new value: 998
New value: 998
The current character value = B
Enter new value: R
New value: R
The current float value = 123.899696
Enter new value: 1221.788998955
New value: 1221.78894
The current double value = 456.890977
Enter new value: 332233.78885984984774
New value: 332233.78886
```

2.5.2 Variable Qualifiers

Variable qualifiers are referred to as **variable modifiers**. These modifiers can further be classified into **type modifiers** or qualifiers and **access modifiers** or qualifiers.

➔ **Type modifiers**

We have seen that the precision of a floating-point variable can be extended by declaring it to be of type double. The range of a default floating-point variable is 3.4 * (10**-38) to 3.4 * (10**+38). If we define a float variable to be of type double, then the default range increases to 1.7 * (10**-308) to 1.7 * (10**+308) thus allowing the variable to store data with a higher precision.

Similarly it is possible to extend the limits of an integer variable by using the **qualifier long** in the declaration of the **int** variable. Other such qualifiers available in C++ are **signed**, **unsigned** and **short**. These qualifiers are also referred to as type modifiers. The use of qualifiers has been illustrated in Example 3.

Example 3:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 clrscr() ; // Clear the screen
 // Integer variable limit: -32768 to +32767
 int ids_int;

 // Long integer variable limit: -2147483648 to +2147483647
 long int li_int;

 // Unsigned integer variable limit: 0 to +65535
 unsigned int ui_int;

 // Unsigned short int variable limit: -32768 to +32767
 unsigned short int usi_int;
 ids_int= 32767 ;
 li_int= -2147483648 ;
 ui_int= 65535 ;
 usi_int= -32768 ;
 cout << " \n default signed integer: " << ids_int;
 cout << endl ;
 cout << "\n long integer: " ;
 cout << li_int<< endl ;
 cout << "\n unsigned integer:" << ui_int<<endl;
 cout << "\n unsigned short integer: " ;
```

```
  cout << usi_int<< endl ;
  // Function to get a character entry from standard input getch() ;
}
```

In Example 3, certain qualifiers have been used to extend or limit the integer data type.

After initializing the values to the declared variables, the same have been displayed on the screen using the extraction operator. Now try modifying the values of these variables in such a manner that they go beyond the functional ranges of their corresponding data types. The values thus displayed on the screen will be inaccurate.

Output of the program in Example 3:

```
ids_int: 32767
li_ext_l_int: -2147483648
ui_int: 65535
usi_int: 32768
```

Now let us assume that we change the values which have been assigned to the variables during initialization in Example 3. These values are beyond the ranges specified for these data types. For example if we initialize the values to:

```
ids_int= 32769
li_int= -2147483650
ui_int= 70000
usi_int= -32769
```

The output for the variable initializations will result in the output being shown as junk for the different variables. An instance of the corresponding output is shown:

```
ids_int: -32767
li_ext_l_int: 2147483646
ui_int: 4464
usi_int: 32767
```

You can try the same with other values which are be beyond the range limits specified for the given data types.

➔ **Access modifiers (declaration of constants)**

C++ also offers the facility to use **access modifiers** with respect to the variables defined. The most important of these access modifiers is the constant modifier, which has been discussed in the next section.

C++ also has a feature for declaring constants in addition to the standard set of variables. A constant is a data set that holds a value, which is set when initialized and which cannot be changed later on anywhere during the course of a program. This model follows the same basic construct, which is followed while declaring a variable, the only difference being that the *const* keyword has to be added before the data type.

Constants may be declared as:

```
const float  pi = 3.146 ;

const char   three = '3' ;

const int    number = 5 ;

const doubled_number = 8738478.9898
```

The constants declared can be used as:

```
float f ; // Declaring a variable f of type float

f = pi * number // Using the products of constants pi and number
```

The following are invalid operations on constants.

```
three = '7' ; // will generate the error:

number = 6 ; // cannot modify a const object
```

```
Functionally, constants can be treated almost like normal variables.
The C++ compiler however, performs consistency checks on the usage of
constants to ensure that they are not modified either directly or by side
effects.
```

C++ contains four basic types of constants. These are discussed here.

- **Integer constants**

  One or more digits constitute an integer constant. An integer beginning with the digit 0 (zero) is considered as an octal and can contain the digits 0 through 7 and an integer beginning with 0x or 0X is taken to be hexadecimal. A hexadecimal data value can contain 0 through 9 and the letters a or A through f or F.

  Examples of integer constants are:

  41 – Simple integer constant

  0345 – Octal constant

  0x9F – Hexadecimal constant

- **Floating point constants**

  Floating point constants in C++ follow two formats:

✧ Decimal notation

A decimal notation is one where there is a series of digits representing the whole part of a number, followed by a decimal point, which is followed by a series of digits representing the fractional part. Either one of the parts can be omitted at any given point of time, but never both. In addition, the decimal point cannot be omitted.

Examples of decimal constants are:

12.5677 - Standard decimal value constant

0.8903 - Decimal notation omitting the whole part of a number

12.0 - Decimal notation with no fractional part

We can declare decimal constants as shown.

```
const float f_number=12.5766 ; // Standard decimal value

const float f_value=0.8907; // Omitting whole part

const float f_num=12.0; // No fractional part
```

✧ Scientific notation

This notation comprises of a mantissa in decimal notation. It is followed by the letter e or E, followed by an exponent value indicator. The exponent value indicator consists of an optional plus or minus sign, which is followed by a series of digits, which form a whole number.

Example 4 illustrates the use of scientific constants.

Example 4:

```
#include <iostream.h>


void main(void)

{

 const double d_const_val = 9.493E-99 ;

  cout << d_const_val ;

}
```

Output:

```
9.493e-99
```

- **Character constants**

  Character constants consist of a single character within single quotation marks. The character thus enclosed can be any character in the ASCII character set. It may be represented either directly (eg. 'C') or as an escape sequence. An escape sequence is a multi-character combination, the backslash forming the first character followed by a certain ASCII character. The escape sequences discussed in section 2.2 are examples of character constants.

We can declare character constants as shown:

```
const char c_alpha = 'A' ;

const char c_beta = 'B' ;
```

- **String literals**

String literals/constants consist of double quotation marks containing any number of characters (including none). Escape sequences are also permitted. For instance the representation of an apostrophe within the string literal requires the escape sequence /'.

A string constant contains all the specified characters in a single string and these characters are identified by a single name. This string is terminated by a **NULL** character. The '\0' notation represents the **NULL** character. We cannot assign a **NULL** character value directly to a string constant. Example 5 illustrates the use of string literals or constants.

Example 5 illustrates the use of string literals or constants:

Example 5:

```
#include <iostream.h>


void main(void)

{

 const double d_const_val = 9.493E-99 ;

 cout << d_const_val ;

}
```

Output:

```
Hello world
```

The program in Example 6 illustrates the use of the various concepts discussed in this chapter.

Example 6:

```
#include <iostream.h>
void main(void)
{
 int i_age ;
 char str_name[20] ;
 float f_salary ;
 const double const_tax = 10000.90 ;

 cout << "Enter your name: " ;
 cin >> str_name ;
 cout << endl ;

 cout << "Enter your age: " ;
 cin >> i_age ;
 cout << endl ;

 cout << "Enter your monthly salary: " ;
 cin >> f_salary ;
 cout << endl ;

 cout << "Hello Mr. " << str_name ;
 cout << ". Your age as entered is " << i_age << endl ;
 cout << "Your total tax payable = " << const_tax ;
}
```

Output:

```
Enter your name: Benson
Enter your age: 35
Enter your monthly salary: 5000
Hello Mr. Benson. Your age as entered is 35
Your total tax payable = 10000.90
```

- **Character Strings**

  Suppose we need to store complete words like an employee's name - Richard. This cannot be stored using the variable definitions discussed so far. "Richard" is a collection of various characters namely, 'R', 'I', 'C'.... Such a collection of characters is referred to as a **string**. A string is necessarily enclosed within double quotes (" "). Unlike other programming languages, C++ does not offer an elementary data type to handle sequences of characters, or strings. Traditionally, in C++, the compiler sees a string as a continuous set of individual character variables. Strings can be declared as:

```
char str _ string[6] ;
```

The code will declare a string **str_string** of 5 characters. The reason why this code statement defines a character string of 5 elements and not 6 is because every string in C++ is terminated by a null character, that is, '**\0**' which is included within the length of the character string. In other words, the null character indicates where the string ends. An example of a string declaration is:

```
char str _ string[30] = "Good morning" ;
```

To display the value stored in the string str_string, we need:

```
cout << str _ string << endl ;
```

Given are more examples of declaring strings -

```
//Assignment without mentioning the length of a string during //
declaration

char str _ world[] = "Hello world" ;
```

The declaration defines a string variable **str_world** without mentioning the length of the string. The assignment defines the length of the string automatically, which would be 11 in this case. Note that even spaces are included when determining the length of the string.

The program in Example 7 illustrates the use of character strings in an input and output operation.

Example 7:

```
#include <iostream.h>

void main(void)

{

 char str_name[25] ; // Declaration

 cout << "Enter your name: " ;

 cin >> str_name ; // Accept value into string

 cout << "Hello " << str_name ; // Display value entered

}
```

Output:

```
Enter your name: Jones

Hello Jones
```

2.6 C++ Operators

Operators are a set of characters, which have a specific meaning particular to the language. They do not include special characters, which serve as punctuation marks, for example (; ": ,). This chapter discusses the operators used in C++.

All operators work on **operands**. **Operands** are *variables, constants* or both.

2.6.1 Assignment Operator

In C++, the assignment operator can be used with any valid C++ expression. The general form of the assignment expression is:

```
variable_name = expression ;
```

where the expression can be a simple constant or a complex expression.

Examples of using the assignment operator are:

```
    value = 1 ; // Simple constant

    value = another_value ; // Assigning a variable

    value = value_1 + value_2 ; // Using arithmetic expression
```

2.6.2 Multiple Assignment

Suppose you have to write a program in which you use three variables `var_1`, `var_2`, and `var_3` and the values of all these variables has to be initialized to the same value, say 70. According to what we have discussed so far, the same would require the following code:

```
int var_1, var_2, var_3 ;

var_1 = 70 ;

var_2 = 70 ;

var_3 = 70 ;
```

Note that by this method, we have to use three code statements to initialize the values of the three variables. However, by using the multiple assignment feature, we can initialize all the three variables in a single code statement as:

```
var_1 = var_2 = var_3 = 70 ;
```

However, such initializations cannot be done at the time of declaring the variables.

For example,

```
int var_1 = int var_2 = int var_3 ;
```

The expression will give an error.

2.6.2.1 Arithmetic Operators

Arithmetic operators in C++ are divided mainly into binary and unary types. The binary operators require two operands to function. The unary operator requires only a single operand to function.

2.6.2.2 Binary Arithmetic Operators

In C++, there are five major arithmetic operators, namely the addition, subtraction, multiplication, division and modulus operator. They are represented by + , - , * , / , and % symbols respectively.

All operators discussed are *binary operators* and require two operands to function. Examples corresponding to each of them have been given.

➔ **The addition operator (+)**

The addition operator adds the first operands and the second operand.

Example:

```
int a=10, b=20, result ;
result = a+b ;
```

Here the variable **result** will receive the value of the addition operation between a and b.

➔ **The subtraction operator (-)**

The subtraction operator subtracts the value of the second operand from the first and returns the value of the subtraction operation.

Example:

```
int big=50, small=54, ans ;
ans = big - small;
```

In this example, the value of variable '**small**' will be deducted from the value of variable '**big**' and the result is assigned to the variable '**ans**'.

➔ **The multiplication operator (*)**

The multiplication operator multiplies the values of the two operands (to its left and right) and returns the product value of the multiplication operation.

Example:

```
int mul_1=10, mul_2=20, mul_ans ;
mul_ans = mul_1 * mul_2 ;
```

In this example, the variable '**mul_ans**' will receive the product of the two values '**mul_1**' and '**mul_2**'.

➔ **The division operator (/)**

The division operator divides the first operand by the second and returns the quotient of the division operation. It results in the **Divide By Zero Error** when the second operand is 0.

Example:

```
 int dividend=100, divisor=10, quotient ;
 quotient= dividend/divisor;
```

In this example, the variable quotient will receive the value of the quotient after dividing the value stored by the variable **dividend** by the value stored in the variable **divisor**.

How the following expression will be evaluated ?

x = 140 / 7 / 5

➔ **The modulus operator (%)**

The modulus operator divides the first operand by the second and returns the remainder of the division operation.

Example:

```
int x=20, y=3, ans ;
ans = x% y ;
```

In this example, the variable '**ans**' will receive the remainder after dividing **x** by **y**. Thus the value 2 will be assigned to '**ans**'.

If the first operand in the modulus operation is smaller than the second operand, then what is the output?

Let us consider an example, which shall use the arithmetic operators in a program. We have a program in which the user is required to enter the speed of a vehicle and the time taken for it to travel from one point to another. Based on these entries, the program calculates the distance covered by the vehicle using the formula:

Distance = Speed * Time

The variables used are:

```
f_dist = distance travelled.
f_speed = speed of the vehicle.
```

```
    f_time = time taken by the vehicle to move from point a to b.
```

Example 8:

```
#include <iostream.h>
#include <conio.h>
void main(void)
{
 float f_speed, f_dist, f_time ;

 clrscr() ;
 cout << "Enter the speed of the vehicle: " ;
 cin >> f_speed ;
 cout << endl ;

 cout << "Enter the time taken by the vehicle: " ;
 cin >> f_time ;
 cout << endl ;

 f_dist = f_speed * f_time ;

 cout<<"The distance covered by the vehicle="<< f_dist ;

}
```

Output:

```
Enter the speed of the vehicle: 95.5
Enter the time taken by the vehicle: 1.8
The distance covered by the vehicle = 171.899994
```

Let us consider a case where addition and multiplication operators can be used. We calculate the distance travelled by a vehicle using the formula distance = speed * time travelled. In the same time if the speed of the vehicle is increased the distance covered by the vehicle in the same amount of time will be more. In the following program, the user enters the initial speed of the vehicle and the time. The program then calculates the distance covered by the vehicle in the given time. The program is then supplied with a new value for speed and the distance that will be covered in the same amount of time is calculated.

Example 9:

```
#include <iostream.h>
#include <conio.h>
void main(void)
{
 float f_speed, f_dist, f_time, f_increase ;
 clrscr() ;

 cout << "Enter the speed of the vehicle: " ;
 cin >> f_speed ;
 cout << endl ;

 cout << "Enter the time taken by the vehicle: " ;
 cin >> f_time ;
 cout << endl ;
 f_dist = f_speed * f_time ;

 cout<<"The distance covered by the vehicle = " << f_dist ;
 cout << endl ;

 cout << "\nIncrease in the speed of vehicle: " ;
 cin >> f_increase ;
 cout << endl ;

 f_speed = f_speed + f_increase ; // Increase in speed

 f_dist = f_speed * f_time ;
 cout << "The distance covered by the vehicle = " << f_dist ;
 cout << endl ;
}
```

Output:

```
Enter the speed of the vehicle: 10

Enter the time taken by the vehicle: 1

The distance covered by the vehicle = 10

Increase in the speed of vehicle: 20

The distance covered by the vehicle = 30
```

Now assume you have a program where the user can provide you with the distance travelled and the time taken. You now have to calculate the speed of the vehicle using the formula given in your program.

Speed = Distance / Time

Example 10 provides the solution to this problem.

Example 10:

```
#include <iostream.h>


void main(void)
{
 float f_speed, f_dist, f_time ;

 cout << "Enter the distance travelled: " ;
 cin >> f_dist ;
 cout << endl ;
 cout << "Enter the time taken: " ;
 cin >> f_time ;
 cout << endl ;
 f_speed = f_dist / f_time ;

 cout << "The speed of the vehicle = " << f_speed ;
}
```

Output:

```
Enter the distance travelled: 900

Enter the time taken: 90

The speed of the vehicle = 10
```

2.6.2.3 Unary Arithmetic Operators in C++

The unary operators in C++ use only a single operand. The unary operator is either specified before or after the operand depending on the usage. C++ provides the following unary arithmetic operators:

- ➔ Negation operator
- ➔ Increment operator
- ➔ Decrement operator
- ➔ **Negation Operator (-)**

  The negation operator negates the associated operand and returns the negated value of the associated operand. The negation operator is specified to the left of the operand.

  Examples:

  ```
  int x ;
  ```

  ```
  x = -3
  ```

  Here, the constant value 3 will be negated and assigned to the variable **x**.

  ```
  int new, old=5 ;
  ```

  ```
  new = -old ;
  ```

  Here the value of the variable '**old**' is negated and assigned to the variable '**new**'. Hence, the variable 'new' will be assigned the value -5.

- ➔ **Increment (++) and decrement (--) operators**

  Consider that a school register is automated and with the entry of the details of every student, the student count is incremented by 1. The student count is associated to the variable '***std_count***'. This assignment is done using the following code statement:

  ```
  std_count = std_count + 1 ;
  ```

  For example, the same could be coded as:

  ```
  std_count++ ;
  ```

  ```
  OR
  ```

  ```
  ++std_count ;
  ```

The increment operator consists of two addition signs that follow each other without any spaces in between.

  The decrement operator is the opposite of the increment operator. For instance, in the example, in case there was a student who was leaving the school, then when his details are deleted from the school register, the student count should also decrease by 1. This could be represented as:

```
std_count = std_count - 1 ;
```

While using the decrement operator, the same is represented as:

```
std_count-- ;     OR    --std_count ;
```

The decrement operator consists of two subtraction signs following each other without any spaces in between.

Using increment or decrement operators increases the readability of the code.

The difference between pre- and post- fixing the operator is useful when it is used in an expression. When the operator precedes the operand, increment or decrement operation takes place before using the value of the operand. If the operator follows the operand, increment or decrement operation takes place after using the value of the operand.

Consider the following,

```
a = 10;
b = 5;
c = a * b++;
```

In the expression, the current value of b is used for the product and ***then*** the value of b is incremented. That is, c is assigned 50 (a * b) and then the value of b is incremented to 6.

If however, the expression was

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

the value stored in c would be 60, because b would first be incremented by 1, and then the value stored in c (10 * 6). The use of the increment and decrement operators in both pre- and post-fixing formats has been illustrated in Example 11.

Example 11:

```
#include <iostream.h>
void main(void)
{
 int value ;
 value = 1 ;
 cout << "THE INCREMENT OPERATOR" << endl ;
 cout << "Value of value: " << value << endl ;
 cout << "Pre-fix increment operator (++value): " << ++value ;
```

```
  cout << endl ;
  cout << "Value of value: " << value << endl ;
  cout << "Post-fix increment operator (value++): " << value++ ;
  cout << endl ;
  cout << "Value of value: " << value << endl ;

  cout << "\n\nTHE DECREMENT OPERATOR" << endl ;
  cout << "Value of value: " << value << endl ;
  cout << "Pre-fix decrement operator (--value): " << --value ;
  cout << endl ;
  cout << "Value of value: " << value << endl ;
  cout << "Post-fix decrement operator (value--): " << value-- ;
  cout << endl ;
  cout << "Value of value: " << value << endl ;
}
```

Output:

```
THE INCREMENT OPERATOR
Value of value: 1
Using the pre-fix increment operator (++value): 2
Value of value: 2
Using the post-fix increment operator (value++): 2
Value of value: 3
THE DECREMENT OPERATOR
Value of value: 3
Using the pre-fix decrement operator (--value): 2
Value of value: 2
Using the post-fix decrement operator (value--): 2
Value of value: 1
```

2.6.3 Relational Operators

Often, we need to compare different sets of data. It may include evaluating whether the value of a variable is greater than, less than or equal to the other. These comparisons draw relations between different data values. These relations are represented in C++ using relational operators.

Relational operators are used to test the relationship between two variables, or between a variable and a constant. For example, using the expression a > b it can be checked whether the value stored by variable a is greater than the value stored by the variable b.

Suppose a program has to perform certain steps on the basis of the condition that a is less than 10.

The relation is thus represented as a < 10. C++ has six relational operators. These are defined as shown in table 2.3:

Operator	**Meaning**
==	Equal to
>	Greater than
<	Less than
!=	Not equal to
>=	Greater than or equal to
<=	Less than or equal to

Table 2.3: Relational Operators in C++

Some of the operators listed contain one character whereas others contain two characters. Those consisting of two characters must not have a space between them.

The use of these relational operators will be shown later in this session which incorporate the 'if' construct. Some examples of using the relational operators are:

Equal to operator

```
b == 3 // where the value of b is compared to the constant 3

b == c // where the value of b is compared to the value of c
```

Greater than operator

```
b > 3 // whether the value of b is greater than 3

b > c // whether the value of b is greater than value of c
```

Less than operator

```
b < 3 // whether the value of b is less than 3

b < c // whether the value of b is less than the value of c
```

Greater than or equal to operator

```
b >= 3 // whether the value of b is greater than or equal to 3

b >= c // whether the value of b is greater than or equal to

 //value of c
```

Less than or equal to

```
b <= 3 // whether the value of b is less than or equal to 3
b <= c // whether the value of b is less than or equal to value
 //of c
```

Not equal to

```
b != 3 // whether the value of b is not equal to 3
b != c // whether the value of b is not equal to value of c
```

These operators are of special use in decision-making.

2.6.4 Logical Operators

Symbols that are used to combine or negate expressions containing relational operators are referred to as logical operators.

Suppose we need to evaluate a candidate applying for admission to a computer course. The applicant is to be considered if the qualification of the candidate is grade **12** and has attained the age of **18** years. To code such an expression, you would use a logical operator, called **AND**. The **AND** operator is represented by the symbol && (double ampersand). The code fragment used is:

```
equal == 12 && age == 18
```

The **OR** logical operator is used when at least one of the two conditions evaluated must be true in order if the compound condition has to be true. Two consecutive vertical bars, || are used to represent the **OR** operator in C++.

Continuing with the previous example, we can also consider an applicant if the qualification is grade **10** or if the candidate has attained the age of 21 years. The code required is:

```
qual == 10 || age == 21
```

The **NOT** logical operator is represented by a single exclamation mark '!'. This operator is used to reverse the logical (true or false) value of the expression to its right. For example we cannot consider a candidate whose has **not** attained the age of 18 years. Hence, the code:

```
(! (age >= 18) ) // age is not equal to or more than 18
```

Logical operators are also used in conjunction with decision constructs.

2.7 Precedence and Order of Evaluation

The C++ language follows the standard precedence for the basic arithmetic operators. Precedence rules help in removing ambiguity of the order of operations performed while evaluating an expression.

For example, given an expression like:

```
2 * 3 / val_1
```

It becomes difficult for one to assess which arithmetic operation is going to be performed first. We can hence, use the rules pertaining to the precedence of operators to define the manner in which the expression is going to be evaluated. According to precedence of operator both '*' and '/' **have the same precedence**. Since, the associativity (discussed later) is from left to right the multiplication operation is executed before the division operation. If we want to evaluate the division operation in the expression first and then multiply it with 2, the following code is required:

```
2 * (3 / val_1)
```

We have used the brackets to modify the order of evaluation of the operations in the given expression. Also important for this purpose is the associativity of the operators. Associativity defines the direction in which the expression is evaluated when an operator is involved. The precedence and associativity of all the operators including those introduced by C++ are summarized in table 2.4. The list is in the descending order of precedence.

Operator	**Associativity**
( ) [ ] . →	Left to right
- ! & * ~ ++ --	Right to left
+ -	Left to right
<< >>	Left to right
< <= > >=	Left to right
= = !=	Left to right
&	Left to right
^	Left to right
|	Left to right
&&	Left to right
| |	Left to right
?:	Right to left
= += -= *= /= %= &= |= ^= <<= >>=	Left to right

Table 2.4: Order of Precedence in C++

Operators with the highest precedence are performed first. For instance, in the expression

i + u * v ;

In the expression the multiplication operation is performed the addition takes place. Parentheses or brackets can be used to override the precedence. For example, in the following expression the addition is performed first.

(i + u) * v ;

When operators of the same precedence appear in the same expression, they are usually evaluated from left to right. For example, the following expression illustrates how the associativity law is carried out among the operators with same precedence.

i * u / v ;

The expression is evaluated from left to right as the precedence of the operators are same.

Step 1: *Multiply i by u*

Step 2: *Product of i * u divided by v*

Example 12 illustrates precedence of operators in C++:

Example 12:

```
#include <iostream.h>
#include <conio.h>
void main(void)
{
 int val_1 = 10, val_2 = 20, val_3 = 2, result_val ;

 clrscr() ;
 result_val = val_1 + val_2 * val_3 ;
 cout << "val_1 + val_2 * val_3 = " << result_val ;
 cout << endl ;

 result_val = (val_1 + val_2) * val_3 ;
 cout << " (val_1 + val_2) * val_3 = " << result_val ;
 cout << endl ;
}
```

Output:

```
val_1 + val_2 * val_3 = 50
(val_1 + val_2) * val_3 = 60
```

2.8 Mixed Mode Expressions & Implicit Type Conversion

The C++ programming language offers flexibility to the programmer. Consider the following statement:

```
x = 5 * 6.7 ;
```

This arithmetic expression contains one operand of type `int` and another of type float. The result is an integer. This is known as implicit type conversion. The program in Example 13 illustrates mixed-mode calculations:

Example 13:

```
#include <iostream.h>


void main(void)
{
 int val_1 = 2, ans_val ;
 float val_2 = 12.34, result_val ;


 result_val = val_1 * val_2 ;
 cout << val_1 << " * " << val_2 << " = " << result_val << endl ;


 ans_val = val_1 * val_2 ;
 cout << val_1 << " * " << val_2 << " = " << ans_val << endl ;
}
```

Output:

```
2 * 12.34 = 24.68
2 * 12.34 = 24
```

Note the type conversions, which have occurred in the output for the given program.

A mixed mode expression is one in which the operands are not of the same type. In this case, the operands are converted, before evaluation, to maintain compatibility between data types. Table 2.5 shows the conversion:

Operand 1	Operand 2	Result
Char	Int	Int
Int	Long	Long
Int	Double	Double
Int	Float	Float

Operand 1	Operand 2	Result
Int	Unsigned	Unsigned
Long	Double	Double
Double	Float	Double

Table 2.5: Type Conversions in C++

C++ permits integer values to be stored into variables that have been declared as floating point variables. The number thus entered is merely converted to its floating-point equivalent; its value is not changed. Similarly, C++ also allows programmers to store floating point values into variables declared to be of type integer. In this case, however, the value is changed because the number is truncated, that is, its fractional portion is lost. For example, consider the following program segment:

```
int x, y ;
float a, b ;

x = 3 ;
a = 987.654 ;

y = a ;
b = x ;
```

Here, the integer 3 is assigned to the integer variable x and the floating point number 987.654 is assigned to the floating point variable a. After the value of a has been stored into y, displaying the value of y shows that it has been truncated to 987. However, the final value of b is 3.0, which is equivalent to 3, the original value.

Hence, y = 987 and b = 3.0.

Consider the following code:

```
char ch = 'A' ;
ch = ch + 32 ;
```

In this example, the value 'A' is assigned to the variable '***ch***'. The ASCII code of 'A' is 65. In the second line, the value 32 is added to the value of '***ch***', thus indicating reference to the ASCII code value 97. The ASCII code 97 represents the character '**a**'. Hence, the '***ch***' now contains the value '**a**'.

Any character type data in C++ is handled by the compiler as an ASCII value ranging from 0 to 255. The English alphabet set ranging from 'A' to 'Z' has the ASCII value 65 to 90. Similarly, the alphabets 'a' to 'z' have the value 97 to 122. So if 'C' is subtracted from 'A' the result will be ASCII value 2 which represents the character '•'.

```
float f = 10.0 ;
int  i = 0 ;
```

```
i = f / 3 ;
```

In this example the constant 3 will be converted to type float and then floating point division will take place, resulting in 3.333333, but since the variable i is an integer, the value will be automatically truncated to 3 and the fractional part will be lost (implicit conversion from float to `int`).

This illustrates that C++ is not a strongly typed language. The implicit conversion, thus occurring, is also called silent conversion since the programmer is not aware of these conversions. The flexibility of the C++ language, to allow mixed type conversions implicitly, saves a lot of effort on the part of the programmer, but at times, it can give rise to bugs in the program.

2.9 Type Casting

Implicit type conversions, as allowed by the C++ language, can lead to errors creeping into the program. Therefore, explicit type conversions may be used in mixed mode expressions. This is done by type casting by using the (***type***) operator as follows:

(*type*) expression

For example,

A floating point variable ***a*** needs to be type cast as an integer. The syntax to follow would be:

```
float a ;
```

```
(int)a
```

The ***expression*** is converted to the given ***type*** as per the rules stated. The use of the cast operator is as follows:

```
float a = 10.0, b= 3.0, c ;
```

```
c = a / b ;
```

This expression will result in 3.333333 being stored in c. If we need to divide only the integer values (values to the left of the decimal point) of two floating point numbers, then the following expression with the type cast can be used:

```
c = (int)a / (int)b ; OR   c = (int) (a / b) ;
```

The type -casting operator can be used both on constants as well as on variables. This has been illustrated in the following code fragment:

```
(int) 3.24258
```

Cast is performed only after the entire expression within the parentheses is evaluated:

For example,

```
    (double) (3 * 4 / 7)

    (float) (a + 5)
```

First, the expression (3 * 4 / 7) is first evaluated and then it is converted to type double. Similarly, in the second statement, first the expression a + 5 is evaluated and then it is converted to type float.

2.10 C++ Shorthand Operators

While discussing the increment and decrement operators, we have seen how the C++ syntax reduces the coding required in routine programming constructs. In addition to the increment and decrement operators, the C++ syntax also supports some other operators, which help in reducing the code required in several routine programming constructs. These programming constructs are referred to as shorthand operators. Let us first illustrate the need for these operators. Suppose a program has to perform a set of steps during which it must increment the value of a variable by 5 every time it executes. Let us assume the variable here is ***test_var***. The syntax as discussed so far would be:

```
test_var = test_var + 5 ;
```

The increment operator cannot be used in this case because the increment operator is used to increment the value by 1. We need to increment our value by 5 in this case. Hence, we can use the shorthand operators provided by C++ for the purpose. This case would require the addition shorthand operator (+=) to be used. The syntax to follow would thus be:

```
test_var += 5 ;
```

Note the exclusion of the variable ***test_var*** to the right side of the equal to sign in the given example. Every shorthand operator consists of an arithmetic operator (+, -, /, *, %) and an assignment operator (=). These two are combined, by ensuring there are no spaces between the two, to form a single operator. This operator requires a variable to be specified to its left, to which the modified value is assigned. On the right side, the value by which the variable to the left is to be modified is specified. In other words, this operator takes the value to the left of the operator and modifies it by the value specified to its right. Due to this feature of updating, these shorthand operators are also referred to as the updating assignment operators.

The shorthand operators available in C++ have been listed in table 2.6 with their respective examples:

Statement	**Is equivalent to**
a += b ;	a = a + b ;
c -= d ;	c = c – d ;
e *= f ;	e = e * f ;
g /= h ;	g = g / h ;
i %= j ;	i = i % j ;

Table 2.6: Shorthand Operators in C++

Examples of using the shorthand operators in C++ are:

```
int a = 20 ;
int b = 5 ;

a += b; // Increments ‘a’ by the value stored in ‘b’.
a += 5 ;// Increments ‘a’ by the value 5, hence, value of a = 25.

a *= b ;// Multiplies the value of ‘a’ by ‘b’.
a *= 5 ;// Multiplies the value of ‘a’ by 5a.
 //Hence, value of a = 100.
```

2.11 Check Your Progress

State whether the following are TRUE or FALSE

1. The \b escape sequence is used as a tab insertion operator for output strings.
2. The names of variables, functions, labels and various other user-defined references are called identifiers.
3. The C++ language has in-built support for string and boolean data.
4. The arithmetic addition operator is a binary operator.
5. The increment and decrement operators in C++ can be used to increment or decrement the values of the associated variable by user specified values.
6. Multiple assignment can be done for variables at the time of declaration.
7. The AND logical operator is represented by the || notation in C++.
8. C++ is a strongly typed language and does not support mixed mode expressions or implicit type conversion.

2.12 Do It Yourself

1. Write a program to accept your name, age and salary and store these details in variables. Display the same on the screen leaving three lines in the middle of the input area and the display area.

   Your output should look like this:

   Enter your name: James

   Enter your age: 21

   Enter your salary: 5000

   Your name is Jones

   Your age is 21

   Your salary is 5000

2. Write a program to accept your first name and your surname as two different data elements.

   Display these elements on the screen in the following display format:

   Enter your first name: Jonathan

   Enter your surname: Longperson

   Display the input in the following format:

   Hello Jonathan   Longperson

   Hint: Use escape sequences.

3. Consider the following program

```
#include <iostream.h>
void main(void)
{
 int case ;
 case = 10 ;
 cout << case ;
}
```

   a. Will the program compile successfully?

   b. If the program does not compile make corrections to the same.

4. Consider the following program

```
#include <iostream.h>
void main(void)
{
 int i_number ;

 cout << "Enter a number: " ;
 cin >> i_number ;
 cout << "You entered: " << i_number ;
}
```

Given is a list of inputs and their corresponding expected output:

Input value	Required output
255	You entered: 255
32768	You entered: 32768
-50000	You entered: -50000
2147483600	You entered: 2147483600

a. Verify whether your program gives you the required output.

b. If not, in a single statement modification, correct your program to give the required output for the mentioned cases.

5. Write a program to accept your name, place of birth and monthly salary. Store these details in their respective variables. Display the monthly salary for all the users which is fixed at a constant of 10000. Your program interface should be as follows:

```
Enter your name: Armen
Enter your place of birth: America

Hello Armen from America
Your monthly salary is: 10000.
```

Hint: Use constants for storing the monthly salary.

6. Consider the following program:

```
#include <iostream.h>
void main(void)
```

```
{
 const int i_const ;
 cout << “Value of i_const” << i_const << endl ;
}
```

a. What is wrong with the program?

b. Correct the errors to give the following output:

Value of i_const = 900

7. What is the error in the following program ?

```
#include <iostream.h>
void main(void)
{
 int int_value = 19 ;
 cout << “Value of int_value = “ << intvalue ;
}
```

8. Key in the following program.

```
#include <iostream.h>
void main(void)
{
 int val_1, val_2, val_3 ;
 val_1 = 1 ;
 val_2 = 2 ;
 val_3 = 3 ;
 cout << “val_1 = “ << val_1 << “, val_2 = “ << val_2 << “, val_3 = “ << val_3 << endl ;
}
```

a. What is the output of the program?

b. Modify the given program to add the values of val_1 and val_2 and print the sum on the screen.

9. Using the arithmetic operators provided in C++, write a program that declares and assigns values to the variables val_1, val_2, val_3, and then does the following:

a. Doubles the value of val_1.

b. Multiplies the value of val_2 by itself.

c. Halves the value of val_3.

d. Prints the results of the preceding operations.

10. Consider the following program.

```
#include <iostream.h>
void main(void)
{
 int a = 90 ;
 a++ 5 ;
 cout << "The value of variable a = " << a ;
}
```

Is the program given correct ? If not, modify it to give the following output.

```
 The value of variable a = 95
```

11. Consider the following program:

```
#include <iostream.h>
void main(void)
{
 int a = 0, b ;
 b = 12 ;
 cout << "Output = " << b / a ;
}
```

What is the output of the program?

12. Write a program to accept a value from the user and increment the value by 1 and display the same on the screen. The display and increment should be done in a single code statement.

13. Write a program to accept three values val_1(`int`), val_2(double) and val_3(float). Perform the following operations on these values:

a. Add the three values.

b. Subtract val_3 from val_2 and add the result to val_1.

c. Multiply the values val_1 and val_2. Subtract val_3 from the product thus obtained.

d. Multiply the values val_1 and val_3 and divide the product by val_2.

e. Display the results of the transactions on the screen.

Summary

- ➔ The areas where the program stores data are known as variables. These variables are referred to by their names as defined by the programmer, which are known as **identifiers**.
- ➔ The C++ character set consists of the uppercase letters A to Z, the lowercase letters a to z, the digits 0 to 9, and certain special characters.
- ➔ C++ uses certain combinations of these characters, such as \b, \n and \t, to represent special conditions such as backspace, newline and horizontal tab respectively. These character combinations are known as **escape sequences**.
- ➔ When an identifier is declared outside any function, its scope becomes global. Which means it is functional through the entire program file following its declaration.
- ➔ Keywords are words that are a part of the C++ language and should be used only in the context as defined.
- ➔ The data type is one of the fundamental properties of a variable. It may be considered as the definition of the set of values that a given variable can accept.
- ➔ A string is necessarily enclosed within double quotes “ “.
- ➔ Variable modifiers can be classified into type modifiers or qualifiers and access modifiers or qualifiers.
- ➔ Operators are the basic tools used by C++ to provide the facility of handling large and complex calculations.
- ➔ The precedence of operators in C++ follows the standard arithmetic model of representing calculation.

Session - 3

Flow Control Statements

Welcome to the Session, **Flow Control Statements**.

This session introduces conditional constructs in C++. The session describes selection, iteration, and jump statements. Further, the session describes the selection conditional constructs, for e.g., If, If else, If else If, Nested if, and so on.

At the end of this session, you will be able to:

➔ Explain the selection constructs

- The if statement
- The if...else statement
- The if...else if...else statement
- Nested if
- The switch statement
- Nested switch

➔ Identify the iteration constructs

- The while loop
- The do...while loop
- Nested while and do...while loops
- The for loop
- Multiple initializations/increments in for loop
- Nested for loops

➔ Understand simple control statements

3.1 Introduction

Control flow statements are used when it is required to change the flow of the program after a test or decision is taken. These control flow statements thus specify the order in which computations are carried out. The C++ language offers a number of control flow statements. These statements are fundamentally classified as *decision constructs or branching statements* and *iteration constructs*.

3.2 Decision Constructs

The standard method for executing a set of program instructions is the sequential model. This method follows a serial pattern of execution; it executes a given code statements one after the other following the order in which they have been written. However, very often, instead of executing statements in a sequential manner, one needs to select among two or more statements to execute. The choice of the instructions to be performed depends upon the value of a variable or the value of an expression. In C++, there are fundamentally three kinds of conditional statements. These are indicated by the keywords, *if*, *if-else* and *switch*.

Given in figure 3.1 is an illustration describing the flow of control in sequential statements and selection constructs.

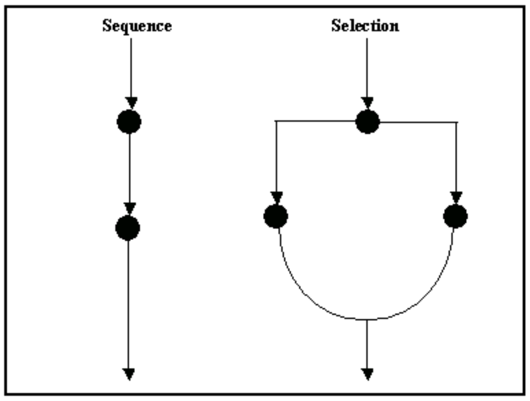


Figure 3.1: Flow of Control in Conditional Constructs

3.2.1 Simple Selection (if)

Let us now discuss a simple ‘ **if** ’ decision statement. The general **if** construct is represented as follows:

```
if (condition)
 Statement_1;
Statement_2;
Or
if (condition)
{
Statement_1;
......
}
Statement_2;
```

The basic **if** statement allows program to execute a single statement, or a block of statements enclosed between braces, if a given condition is true.

If the **condition** specified evaluates to TRUE (that is, non-zero), the **statement_1** given after the if**(condition)** is executed. However, execution of **statement_2** takes place irrespective of whether **statement_1** is executed or not.

The value of variable **grade** must be evaluated. If the value is 'A', then the screen must display the message "Excellent ! Keep it up." The following is the code fragment required for the purpose:

```
if (grade == 'A') // Using the equal to relational operator
 cout << "Excellent ! Keep it up." ;
```

In this example, if the specified expression evaluates to TRUE (that is if value of **grade** is '**A**'), the program displays the output,

```
Excellent ! Keep it up.
```

Similarly, when we need to display a message - "**Good score !!!**", if a student has scored 90 marks or more in science. The required code fragment would look like:

```
// Using the greater than or equal to relational operator
if (eng_marks >= 90)
     cout << "Good score !!!" ;
```

Now, suppose we need to perform a set of steps, which will be executed in case a student was not present for an examination. Present is indicated by the character value 'P'. The code fragment required for the purpose would be:

```
if (std_attd != 'P') // Using the not operator
 cout << "The student was not present for the exam" ;
```

A common mistake is to use the = (equal to) assignment operator as conditional operator in '**if**". The equal to relational operator will compare the values of the two operands. The assignment operator on the other hand will assign the value of the right-hand side operand to the one on the left. The compiler will not indicate an error in this case as any expression, including an assignment expression, is considered as a valid testing expression. It will however, display a warning message.

So far, the statement to be performed in case of the condition evaluating to TRUE is a single or simple statement. This statement can also be a compound statement, which may include more than one statement. For this, the block of statements must be enclosed within curly braces.

The general compound ‘ **if** ’ construct is represented as follows:

```
if (condition)
{
 Processing statement 1 ;
       Processing statement 2 ;
      }
```

For example, suppose we need to assign grades to students based on their scores. If the student has a score, which is equal to or greater than 70, we require to display the message “**Excellent. Grade: A**” and assign the character value ‘**A** ’ to the variable **grade**. The program required is illustrated in Example 1.

Example 1:

```
#include <iostream.h>
#include <conio.h>
void main(void)
{
 char grade ;
 int score ;
 cout << "Enter the score: " ;
 cin >> score ;
 cout << endl ;
 if(score >= 70)
 {
 grade = 'A' ;
 cout << "Excellent. Grade: " << grade ;
 }
}
```

In this example, the use of a compound if selection construct has been illustrated. The user enters the score. If the score is greater than or equal to 70, the value ‘**A**’ is assigned to **grade** and the message “**Excellent. Grade: A**” is displayed.

The output of the program in Example 1 is:

```
Enter the score: 70
Excellent. Grade: A
```

3.2.2 Two-way Selection (if-else, if-else-if)

In Example 1, we have a very limited scope for making decisions. A process is executed, if the score is greater than or equal to 70. How do we specify a student's grade if he/she has scored more than 60 but less than 70? A real grading system will obviously have more than just one condition. Hence, the simple selection construct with the '**if**' statement alone will not suffice in such a requirement. We need to specify a course of action incase the first condition is not met with. This will be possible with the '**if-else**' and '**if-else-if**' conditional statements.

We thus need to use the two-way selection construct, which is represented as follows:

```
if (expression)
 Processing statement 1
else
 Processing statement 2
```

Here, if the expression to be evaluated is false, then **processing statement 1** is ignored and **processing statement 2** is executed. Statement(s) included under **processing statement 2** is also commonly referred to as the **else** branch statement(s). Hence, the two statements are mutually exclusive and both can be either simple or compound statements.

For example, if the score is not equal to or greater than 70, we need to assign the value '**B**' to **grade** and display the message "Average. Grade: B". Hence, this represents a two-way selection construct. The two-way selection construct is a more elaborate form of the simple selection statement. The use of the two-way selection construct has been illustrated in Example 2.

Example 2:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 char grade ;
 int score ;
 clrscr() ;
 cout << "Enter the score: " ;
 cin >> score ;
 cout << endl ;
 if(score >= 70)
 {
 grade = 'A' ;
 cout << "Excellent. Grade: " << grade ;
 }
 else
 {
 grade = 'B' ;
 cout << "Average. Grade: " << grade ;
 }
}
```

Output:

```
Test case 1
Enter the score: 75
Excellent. Grade: A
Test case 2
Enter the score: 56
Average. Grade: B
```

Referring to the program in Example 2, can a program be developed which should take care of any input marks provided by the user. The program should categorize the marks input by the user into a suitable grade, that is, the program should take care of the upper and lower bounds while defining the grades. For example,

```
. . .
 if(score >= 70 && score <=100)
 {
 grade = 'A' ;
 cout << "Excellent. Grade: " << grade ;
 }
. . .
```

The program in Example 2 performs a standard two-way decision process whereby depending on the user input, the program decides the action that has to be taken. Though the two-way selection construct provides a solution to our elementary requirements, it is not absolutely satisfactory. This construct creates an 'either-or' scenario and limits us to a maximum of two choices. In case a greater number of choices are required, we need to use the '**if-else-if**' selection construct, which is a concatenation of several two-way selection statements. The general '**if-else-if**' selection construct is represented as follows:

```
if (expression)
 Processing statement 1
else if(expression) // Use of else if construct
 Processing statement 2
else
 Processing statement 3
```

Here, the execution follows the same pattern as in the standard **if…else** selection construct. However, the else section of the decision construct includes yet another if statement.

Continuing with the grading example, if the score is equal to or greater than 70, we assign the value 'A' to grade and display the message "`Excellent. Grade: A`". Otherwise, we assign the value 'B' to grade and display the message "Average. Grade: B". However, if the score is below 35, then we need to assign the value 'F' to grade and display the message "`Failed. Grade: F`".

The code required for the purpose is given in Example 3.

Example 3:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 char grade;
 int score;
 clrscr();
 cout << "Enter the score: ";
 cin >> score;
 cout << endl;

 if(score < 35)
 {
 grade = 'F' ;
 cout << "Failed. Grade: " << grade ;
 }

 else if(score < 70)
 {
 grade = 'B' ;
 cout << "Average. Grade: " << grade ;
 }
 else
 {
 grade = 'A' ;
 cout << "Excellent. Grade: " << grade ;
 }
}
```

The output of the program in Example 3 is:

Test case 1

```
Enter the score: 34
Failed. Grade: F
```

Test case 2

```
Enter the score: 56
Average. Grade: B
```

Test case 3

```
Enter the score: 75
Excellent. Grade: A
```

3.2.3 Nested 'if' Statement

In addition to the standard decision processing discussed so far, we often face problems where related decision-making is required, that is, certain processing decisions are dependent on yet another set of decisions.

While dealing with related decision structures, we need to use the **nested 'if'** selection construct. An **'if... else'** construct can contain another set of **'if...else'** statements. This is known as the **nested 'if'** construct. The general **nested 'if'** selection construct is represented as follows:

```
if (expression 1)

{

      if (expression 2)

      {

            Processing statement 1 ;

            Processing statement 2;

}

}
```

For example, in the grading system discussed so far, if the grades assigned are also based on the total attendance of the student in addition to his/her marks obtained, then we would need a nested if selection construct. The grading structure would thus have to be developed on the basis of table 3.1:

Marks	Attendance %	Grade
>=80	Not Applicable	A
>=60	>=80	A
>=60	<80	B
>=35	>=80	C
>=35	<80	P
<35	Not Applicable	F

Table 3.1: Grading input

According to the table, if the marks are greater than or equal to 80, then the percentage of attendance is not considered and the student is assigned the grade 'A'.

If the marks obtained is greater than or equal to 60 and the attendance percentage greater than or equal to 80, then the student is assigned grade A and so on. The program segment for this would be written as:

```
if(marks >= 60)
{
 if(att_percent >= 80)
 {
       grade = 'A' ;
 }
 else if(att_percent < 80)
 {
       grade = 'B' ;
 }
 }
```

Hence, the full program required for this problem is given in Example 4.

Example 4:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 int marks, att_percent ;
 char grade ;

 clrscr() ;

 cout << "Enter MARKS: " ;
 cin >> marks ;
 cout << endl ;

 cout << "Enter ATTENDANCE PERCENTAGE: " ;
 cin >> att_percent ;
 cout << endl ;
```

```
if(marks >= 80)
{
grade = 'A' ;
}
else if(marks >= 60)
{
if(att_percent >= 80)
{
      grade = 'A' ;
}
else if(att_percent < 80)
{
      grade = 'B' ;
}
}
else if(marks >= 35)
{
if(att_percent >= 80)
{
      grade = 'C' ;
}
else if(att_percent < 80)
{
      grade = 'P' ;
}
}
else if(marks < 35)
{
 grade = 'F' ;


 }
 cout << "Grade: " << grade ;
}
```

The output of the program in Example 4 is:

Test case

```
Enter MARKS: 60
Enter ATTENDANCE PERCENTAGE: 80
Grade: A
```

3.3 switch Statements

Suppose we want to extend the grading example given in the previous sections to include a feature of displaying some output depending on the grade assigned.

If grade is 'A', display, "Excellent, keep up the good work."

If grade is 'B', display, "Average, can do better."

If grade is 'C', display, "Poor, must work hard."

If grade is 'P', display, "Passed and promoted. Must work hard."

If grade is 'F', display, "Failed. Detained for an academic year."

The additional code to be added to the program in Example 4 is given in the Example 5.

Example 5:

```
 ...
if(grade == 'A')


{
 cout << "Excellent. Keep up the good work." ;
 }
 else if(grade == 'B')
 {
 cout << "Average. Can do better." ;
 }
 else if(grade == 'C')
 {
 cout << "Poor. Must work hard." ;
 }
 else if(grade == 'P')
 {
 cout << "Passed and promoted. Must work hard." ;
 }
 else if(grade == 'F')
 {
 cout << "Failed. Detained for 1 academic year." ;
 }
}
```

Note that the number of 'if...else...if' decision constructs shown in the code fragment. Constructing and understanding such a code construct becomes more and more complex, as the number of these constructs increase. Alternatively, C++ provides us with the 'switch...case' construct, which can be used to handle such complex decision making scenarios with greater simplicity. This also increases the readability of the program code..

The '**switch**' statement is a multi-way decision-making construct, which tests whether an expression matches one of a number of constant values, and branches accordingly. The '**switch**' statement is a neater alternative to using else-if statements where multi-way decision-making is involved. The general format is as follows:

```
switch (expression)
{
case value_1      :     Processing statements
                               break ;

case value_2      :     Processing statements
                         break ;

default           :     Processing statements
}
```

The expression must be enclosed in parentheses, and the body of the switch statement must be enclosed in braces. Statements can be single or multiple statements with or without braces. The values with **case** should be constants. The expression is evaluated and the resultant value is compared with each of the values given with **case**. If any of the values match, the statements following that case till **break**, are executed. If none of the case values match the result of the expression, then the statements following **default**, which is optional, are executed. The keyword **break**, is used to indicate the end of a sequence of processing statements under a particular **case**.

The previous code fragment could thus be represented as:

Example 6:

```
. . .
 switch(grade)
 {
 case ‘A’: cout << “Excellent. Keep up the good work.” ;
            break ;
 case ‘B’: cout << “Average. Can do better.” ;
            break ;
 case ‘C’: cout << “Poor. Must work hard.” ;
            break ;
 case ‘P’: cout << “Passed and promoted. Must work hard.” ;
            break ;
 case ‘F’: cout << “Failed. Detained for 1 academic year.” ;
            break ;
 default   : cout << “Invalid grade encountered.” ;
            break ;
 }
. . .
```

Note that the clarity of representing the multi-way decision structure in the case.

Hence, the switch statement can be used instead of the if...else construct when multi-way decision making is involved.

Several programs offer a facility to list out a set of options for the user to choose from. The user is then asked to enter his choice. For example, we need to specify our destination choice, while booking airline tickets. For the sake of simplicity, let us assume that the following destinations are listed:

```
1.    Delhi
2.    Mumbai
3.    Chennai
4.    Calcutta
5.    Bangalore
6.    Exit
Choose your destination (1-5) or enter 6 to Exit:
```

Such a method of representing a multiple-choice construct is referred to as a menu structure type program. The structure can thus be represented using the 'switch...case' construct as shown in Example 7.

Example 7:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 int dest_choice ;

 clrscr() ;

 cout << "              1. Delhi" << endl ;
 cout << "              2. Mumbai" << endl ;
 cout << "              3. Chennai" << endl ;
 cout << "              4. Calcutta" << endl ;
 cout << "              5. Bangalore" << endl ;
 cout << "              6. Exit" << endl ;
 cout << "\n" ;
 cout << "Choose your destination (1-5) or enter 6 to exit: " ;
 cin >> dest_choice ;
cout << "\n\t\tDestination chosen: " ;


      switch(dest_choice)
 {
           case 1: cout << "Delhi" ;
                  break ;
     case 2: cout << "Mumbai" ;
                  break ;
case 3: cout << "Chennai" ;
                  break ;
           case 4: cout << "Calcutta" ;
                  break ;
           case 5: cout << "Bangalore" ;
                  break ;
           case 6: clrscr() ;
                  break ;
           default: clrscr() ;
     cout << "Invalid entry. Valid (1-6)" ;
                  break ;
      }
 }
```

Output:

```
1. Delhi
2. Mumbai
3. Chennai
4. Calcutta
5. Bangalore
6. Exit
Choose your destination (1-5) or enter 6 to exit: 3
Destination chosen: Chennai
```

In this example, if the option is 1, then the first '**cout**' statement will be executed and the control will pass on to the next statement after the switch. Otherwise, the rest of the '**case**' statements will be evaluated in the same way. If none of them matches, then the last block of statements after '**default**' will be executed.

The '**break**' statement is essential for the proper logical working of the switch structure. It causes the program flow to exit from the switch structure after the correct case statements are executed. The break can be omitted in which case the control jumps over through to the next '**case**' statements, until a '**break**' is encountered. This may be logically wrong, although syntactically correct. Therefore, omitting the break in the first case statement in the example will cause both the '**case 1**' and '**case 2**' statements to be executed, irrespective or whether that case value is satisfied or not.

Omission of the '**break**' statement can be used when the same operation is to be performed for a number of cases. For example, to find out whether a character input is a vowel, a consonant, or a space, we can omit '**break**' for a number of case statements. Example 8 illustrates the use of this feature.

Example 8:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
      char ch ;
      cout << "Enter a vowel / consonant / space: " ;
// getche() - Function to accept any one character as input
      ch = getche() ;
      cout << endl ;

switch (ch)
      {
```

```
case ‘a’:
       case ‘e’:
       case ‘i’:
       case ‘o’:
       case ‘u’: cout << “You entered a vowel” << endl;
                 break ;
       case ‘ ’: cout << “You entered a space” << endl ;
                 break ;
       default:  cout << “You entered a consonant” << endl ;
                 break ;
      }
}
```

The output of the program in Example 8 is:

Output 1

```
Enter a vowel / consonant / space:
You entered a space
```

Output 2

```
Enter a vowel / consonant / space: a
You entered a vowel
```

In this example, if ‘**ch**’ is ‘**a**’, then control starts from the first ‘**case**’ else if it is ‘e’, it starts from the second ‘**case**’ and so on. Control falls through till the first ‘**break**’ statement is encountered and is then transferred to the next statement after the ‘**switch**’.

Note that this program uses the **getche()** function which is a ready made function provided by the C++ library. It is used for accepting any one character as input. This is used particularly when the program should also accept spaces as input. Input through ‘**cin**’ does not accept spaces. Example 9 illustrates the use of ‘**cin**’.

Example 9:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 int marks, att_percent ;
 char grade ;

 clrscr() ;

 cout << "Enter MARKS: " ;
 cin >> marks ;
 cout << endl ;

 cout << "Enter ATTENDANCE PERCENTAGE: " ;
 cin >> att_percent ;
 cout << endl ;

 switch(marks >= 80)
 {
 case 1: grade = 'A' ;
      break ;
 case 0: switch(marks >= 60)
      {
           case 1: switch(att_percent >= 80)
                {
case 1: grade = 'A' ;
                           break ;
                     case 0: grade = 'B' ;
                           break ;
                }
                break ;
           case 0: switch(marks >= 35)
                {
```

```
                             case 1: switch(att_percent >= 80)
                                     {
                                          case 1: grade = 'C' ;
                                                  break ;
                                          case 0: grade = 'P' ;
                                                  break ;
                                     }
                                     break ;
                             case 0: grade ='F' ;
                                     break ;
                       }
                       break ;
         }

  }

cout << "Grade: " << grade ;
}
```

Output of the program in Example 9 is:

```
Enter MARKS: 67

Enter ATTENDANCE PERCENTAGE: 76

Grade: B
```

The program in Example 9 uses a nested switch construct to enable the user to develop a set of selection and sub-selection statements.

Suppose we need to write a program, which accepts the values of two operands from the user. A menu structure is then displayed which provides a set of options for processing these operands.

The program in Example 10 provides an example of using the switch decision construct in a simple menu-based program.

Example 10:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 float op_1, op_2, result_val ;
 int option ;

 clrscr() ;

 cout << "Enter value for operand_1: " ;
 cin >> op_1 ;
 cout << endl ;
 cout << "Enter value for operand_2: " ;
 cin >> op_2 ;

 clrscr() ;
 cout << endl ;

 cout << "              (1) Add          " << endl ;
 cout << "              (2) Subtract     " << endl ;
 cout << "              (3) Multiply     " << endl ;
 cout << "              (4) Divide " << endl ;
 cout << "                  (0) Exit        " << "\n\n";
 cout << "       Enter option: " ;
 cin >> option ;
 cout << endl ;
switch(option)
 {
```

```
 case 1: result_val = op_1 + op_2 ;
      cout<<"\t"<<op_1<<"+" <<op_2<<"="<<result_val ;
      break ;
 case 2: result_val = op_1 - op_2 ;
      cout<< "\t"<<op_1<<" - " <<op_2<<" = " << result_val ;
      break ;
 case 3: result_val = op_1 * op_2 ;
 cout<<"\t"<<op_1<<" * "<< op_2<<"="<< result_val ;
      break ;
 case 4: result_val = op_1 / op_2 ;
 cout <<"\t"<<op_1<<" / "<<op_2 <<" = " <<result_val ;
      break ;
 case 0: cout << "Exit chosen!!! " ;
      break ;
 default: cout << "Invalid choice. Valid between: 0 and 4" ;
      break ;
 }
}
```

Output:

Output 1

```
Enter value for operand_1: 12

Enter value for operand_2: 12
```

In this screen the values for the two operands are accepted from the user. Suppose the user enters 12 as values for both the operands.

Output 2

```
 (1) Add
  (2) Subtract
  (3) Multiply
  (4) Divide
 (0) Exit
Enter option: 1
```

In this screen, the user chooses the operation to be performed on the two operands entered. In this example, the user has chosen 1, that is, **Add**. Consequently, the two operands will be added and the sum is displayed as shown the output 3.

Output 3

```
 (1) Add
 (2) Subtract
 (3) Multiply
 (4) Divide
 (0) Exit
Enter option: 1
12 + 12 = 24
```

The program in Example 8 can also be written using the if…else…if multi-way decision construct.

3.4 Basics of Loops

There are two essential elements to a loop: the **statement** or the block of statements that forms the body of the loop that is to be executed repeatedly, and a loop condition of some kind that determines when to stop repeating the loop.. Any loop construct needs to follow a standard algorithm.

Step 1: - Initialize a counter variable.

Step 2: - Evaluate condition

Step 3: - Increment/decrement the value of counter variable

3.5 while and do…while Loops

The **while and do…while** loops are two of the basic iteration constructs supported by C++.

3.5.1 The while loop

The **while** loop in C++ has the following syntax:

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

The use of the while loop has been shown in the following code fragment:

```
int ctr = 0;
while(ctr <= 1)
```

```
{
 cout << "\nHello world" ;
 ctr++ ;
 }
```

This code fragment prints the string "Hello world" on the screen twice. The loop is executed until the value of **ctr** is lesser than or equal to 1. Hence, the construct can be explained as follows:

1. Expression is evaluated, that is, **ctr <= 1**.
2. If the condition is true, the statements specified within the while construct are executed and the control goes to evaluate the condition (**ctr<=1**). This continues till the condition is false.
3. If the condition is false, the execution continues with the program statements following the loop.

The use of a **while** loop is illustrated in Example 11.

The program in Example 11 will print the numbers 0 to 10 on successive lines.

Example 11:

```
#include <iostream.h>
#include<conio.h>
void main(void)
{
 int x = 0 ;
 clrscr() ;
 while(x <= 10)
 {
 cout << x++ << endl ;
 }
 getch();
}
```

In the program if the value of the variable x is not incremented, then the program goes into an unending series of iterations. This is called an infinite loop.

In addition to the processing, the program in Example 1 also mentions the use of the **clrscr()** and the **getch()** functions.

1. The **clrscr()** function clears the current text window and places the cursor in the upper left-hand corner of the screen. The function prototype definition is found in the header file **conio.h**.
2. The **getch()** function reads a single character directly from the keyboard, without echoing it to the screen. This function is used to pause processing. The processing continues after the user presses

any key on the keyboard. The function prototype definition is found in the header file **conio.h**.

You can return to the C++ editor using the CTRL + Pause key combination. After you return to the C++ environment, you will need to reset the program by using the CTRL + F2 key combination. This is necessary to stop the execution of the program completely.

Example 12 shows an infinite while loop code fragment:

Example 12:

```
...
x = 0
while(x <= 10)
{
 cout << x << endl ;
}
...
```

This code fragment will continue to print “0” on successive lines indefinitely until the execution is interrupted.

In a practical scenario when we need to calculate the grade of all the students, we have to run the same algorithm for all the students. This can be implemented using one of the loop structures. A grade allocating example has been illustrated in Example 13.

Example 13:

```
#include <iostream.h>
#include <conio.h>
void main(void)
{
 int marks, att_percent ;
 char grade, cont = ‘Y’ ;
 while(cont != ‘N’ && cont != ‘n’)
 {
 clrscr() ;
 cout << “Enter MARKS: “ ;
 cin >> marks ;
 cout << endl ;
cout << "Enter ATTENDANCE PERCENTAGE: " ;
 cin >> att_percent ;
 cout << endl ;
 switch(marks >= 80)
 {
 case 1: grade = 'A' ;
      break ;
 case 0: switch(marks >= 60)
      {
 case 1: switch(att_percent >= 80)
                 {
 case 1: grade = 'A' ;
                            break ;
                       case 0: grade = 'B' ;
                            break ;
                 }
                 break ;
            case 0: switch(marks >= 35)
                 {
                 case 1: switch(att_percent >= 80)
                 {
                           case 1: grade = 'C' ;
                                break ;
                           case 0: grade = 'P' ;
                                break ;
                 }
                 break ;
                 case 0: grade ='F' ;
                      break ;
                 }
                 break ;
```

```
        }
 }


cout << "Grade: " << grade << endl ;
cout << "Continue (Y/N): " ;
cin >> cont ;
}
}
```

Output:

Test case 1

```
Enter MARKS: 70
Enter ATTENDANCE PERCENTAGE: 90
Grade: A
Continue (Y/N): Y
```

As you enter any value other than 'N' / 'n', when asked whether you want to continue, you can once again enter the marks and attendance percentage of the student and the grade is generated. The output is thus:

Test case 2

```
Enter MARKS: 56
Enter ATTENDANCE PERCENTAGE: 98
Grade: C
Continue (Y/N): n
```

When either 'N' or 'n' is entered, the program automatically terminates.

Hence, in the case of the **while** loop, the loop condition is evaluated **before** each iteration, that is, before the loop is entered.

3.5.2 The do...while Loop

In the **do...while** loop the concept remains roughly the same except for a slight difference between the two. The while... loop construct evaluates the condition before the loop body is entered whereas the do... while loop evaluates its condition after executing the loop at least once.

The '**do...while**' loop is thus useful in conditions where a certain set of processing statements needs to be performed at least once.

The general format of the '**do...while**' loop is as follows:

```
do
{
 Processing statements
 } while (expression) ;
```

Example 14 illustrates the use of the '**do...while**' loop construct.

Example 14:

```
// code fragment
...
ctr = 11 ;
do
{
 cout << "Hello world" << endl ;
} while(ctr <= 10) ;
...
```

The code fragment prints the string "Hello world" once on the screen. Note the following points:

- ➔ The loop is executed the first time without evaluating the test condition which states that ctr should be less than or equal to 10.
- ➔ The loop will execute first and then condition will be evaluated.

Hence, the output for the program would be:

```
Hello world
```

Example 15 illustrates the scenario where we need to accept an integer value, then add the value of its digits and print the total on the screen.

Example 15:

```
#include <iostream.h>

void main(void)
{
 int n, i_sum = 0, right_digit ;

 cout << "Type an integer value: " ;
 cin >> n ;
 cout << endl ;
 do
 {
 right_digit = n % 10 ; // To extract rightmost digit
 i_sum += right_digit ;
 n /= 10 ;  // Move next digit into rightmost position
 }
 while(n>0);//No more digits to extract if value of n = 0
 cout << "The sum of the digits is: " << i_sum ;
}
```

Output:

```
Type an integer value: 123
The sum of the digits is: 6
```

We will now look into working of nested loops structures.

3.5.3 Nested while and do…while Loops

Both the **while** and **do…while** loops can also be represented in a nested format to enable multiple layers of iteration in a program.

The standard structures of representing the ***while*** and ***do…while*** loops in the nested format are shown in Examples 16 and 17 respectively.

Example 16:

```
while  (condition_1)
{
 while  (condition_2)
 {
statements
 }
}
```

Example 17:

```
do
{
 do
 {
      statements
      }while  (condition_1)  ;
} while  (condition_2);
```

Suppose we need to enter marks of several subjects for number of students, it can be done using nested loop structure as shown in Example 18.

Example 18:

```
#include <iostream.h>
#include <conio.h>
void main(void)
{
 int class_no, marks, tot_studs, roll_no ;
 char choice ;
 clrscr() ;
 do
 {
      cout << "Enter class number: " ;
      cin >> class_no ;
      cout << endl ;
if(class_no >0)
     {
 cout << "Enter TOTAL STUDENTS IN CLASS: " ;
      cin >> tot_studs ;
      cout << endl ;
      roll_no = 1 ;

      while(roll_no <= tot_studs)
      {
      cout << "Enter MARKS for roll number " ;
 cout << roll_no << ":" ;
      cin >> marks ;
      cout << endl ;
      cout << "MARKS ENTERED for roll number " ;
 cout << roll_no << " = " << marks << endl ;
      roll_no++ ;
      }
     }
     cout << "Enter marks for another class (Y/N) ? " ;
     cin >> choice ;
     clrscr() ;

 } while(choice != 'N' && choice != 'n') ;
}
```

Output of the program in Example 18 for a particular set of inputs is:

```
Enter class number: 5
Enter TOTAL STUDENTS IN CLASS: 5
Enter MARKS for roll number 1:65
MARKS ENTERED for roll number 1 = 65
Enter MARKS for roll number 2:67
MARKS ENTERED for roll number 2 = 67
```

```
Enter MARKS for roll number 3:54
MARKS ENTERED for roll number 3 = 54
Enter MARKS for roll number 4:76
MARKS ENTERED for roll number 4 = 76
Enter MARKS for roll number 5:89
MARKS ENTERED for roll number 5 = 89
Enter marks for another class (Y/N) ? N
```

Here, the marks of students in different classes are accepted from the user. First the program accepts the marks for the students in one class according to their roll numbers. After the marks for one class are accepted, the user is asked if he would want to continue entering marks for another class. If the user enters any value other than 'N' or 'n', the user is allowed to continue entering the marks for the next class. Note that the class entered cannot be 0 (zero).

3.6 Iteration Using 'for' Loop

Let us look how another loop construct works.

3.6.1 Defining the 'for' Loop

The for loop is primarily used for executing a statement or block of statements a predetermined number of times, but it can be used in other ways, as we shall see.

We control a for loop using three expressions separated by semicolons, which are placed between parentheses following the keyword for as shown.

```
for (initializing_expression ; condition ; iteration_expression)
 {
      // Loop statements
     }
 // Next statement
```

➔ *Initializing_expression*: This expression is executed once, at the beginning of the loop. It is typically used to initialize one or more loop variables.

➔ *condition*: This expression is evaluated at the beginning of each loop iteration. If it is true the loop continues, and if it is false execution continues with the statement after the loop.

➔ *Iteration_expression*: This expression is evaluated at the end of each loop iteration. It is typically used to modify the values of variables initialized in the first expression.

Example 19 illustrates the use of the **for loop**.

Example 19:

```
...
int x;

for(x = 1; x <= 5; x++)
{
 Processing statements ;
}
...
```

Omitting expressions in the '**for' loop** is illustrated in Example 20.

Example 20:

```
...
int x;

for(x = 1; x <= 5; x++)
{
 Processing statements ;
}
...
```

Each of the expressions of the for loop is separated by a semi-colon (;). There is no semi-colon used after expression 3. Even if the first and/or the third expressions are omitted, the declaration essentially requires the use of two semi-colons as expression delimiters. The **processing statement** as shown is the body of the loop, which may consist of either simple or compound statements.

Let us consider the Example discussed in Example 11. The program in Example 21 prints the values 0 to 10 using the for loop construct.

Example 21:

```
#include <iostream.h>
#include <conio.h>
void main(void)
{
 int num ;
 clrscr() ;
 for(num = 0; num <= 10; num++)
 {
 cout << num << endl ;
 }
}
```

The output of the program in Example 21 is the same as the output shown for the program in Example 11.

If we need to accept 10 numbers from the user and print their sum. Example 22 illustrates the program required for the purpose.

Example 22:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 int i_num, i_count, i_total = 0 ;

 clrscr() ;
 for(i_count = 1; i_count <= 10; ++i_count)
 {
 cout << "\nEnter a number: " ;
 cin >> i_num ;
 i_total += i_num ;
 }
 cout << "\nThe sum total of the numbers entered: " << i_total ;
}
```

The program in Example 22 asks the user to enter 10 numbers. The variable **i_count** is used as an control variable and is incremented after each input. The variable **i_total**, which has been initialized to the value 0, is used to add and store the result of addition. As the user inputs a number, the number is added to the previous value of **i_total**. Using the **for loop** all the ten numbers are added and the result is displayed.

3.6.2 Multiple Initializations and Increments in the 'for' Loop

While dealing with applications, we often come across situations where processing decisions are based on multiple data entities. In other words, we often need to consider the state of multiple data components of a program at the same time, for a single process.

Consider maintaining the score of a cricket team during a match. The score of each batsman is added to the total score of the team. A team consists of 11 players. Hence, we need to keep a record of the total number of players, the score of each player and the total score of the team. Hence, we have to initialize and increment multiple values in this process.

The general format for multiple initialization is:

for(var1 = 1, var2 = 1,… varN = 1; condition; var1++,var2++, …, varN++)

For example, we need to display a series of numbers, say 0 to 10 in ascending and descending order on the same screen. In other words we need to print the series 0, 1, 2, 3, 4,….10 and the series 10, 9, 8, 7, ….0. We thus need to maintain two variables, one for the ascending series and another for the descending series.

The program for the example has been illustrated in Example 23.

Example 23:

```
#include <iostream.h>

#include <conio.h>


void main(void)

{

 int x, y ;


 for(x=0, y=10; x <= 10 && y >= 0; ++x, --y)

 {

 cout << "x = " << x << "        " << "y = " << y << endl ;

 }

}
```

Given in Example 23, is a compound test expression, which uses the logical and operator to define the union of the two expressions. If and only if both these expressions evaluate to true, will the test condition evaluate to true, else it will fail and the loop execution will be terminated.

Suppose we need to accept the score of 11 batsmen in a team. After accepting the score for each batsman, we need to display the total team score and continue with further data entry. The code required is given in Example 24.

Example 24:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 int bats_score, team_score, bats_no, runs_scored ;
 char c_quit ;

 clrscr() ;
 for(team_score = 0, bats_no = 1; bats_no <= 11 && c_quit != 'Y' && c_quit
!= 'y'; bats_no++, team_score += runs_scored)
{
 cout << "\nTEAM SCORE: " << team_score ;
 cout << "\t\tBATSMAN " << bats_no << " - RUNS SCORED: " ;
 cin >> runs_scored ;
 if(bats_no < 11)
 {
      cout << "Quit entering data (Y/N) ? " ;
      cin >> c_quit ;
 }
 }
 cout << "\nTOTAL TEAM SCORE: " << team_score ;
}
```

Output:

```
TEAM SCORE: 0 BATSMAN 1 - RUNS SCORED: 56
Quit entering data (Y/N) ? n

TEAM SCORE: 56 BATSMAN 2 - RUNS SCORED: 56
Quit entering data (Y/N) ? n

TEAM SCORE: 112 BATSMAN 3 - RUNS SCORED: 76
```

```
Quit entering data (Y/N) ? n
TEAM SCORE: 188 BATSMAN 4 - RUNS SCORED: 56
Quit entering data (Y/N) ? n
TEAM SCORE: 244 BATSMAN 5 - RUNS SCORED: 56
Quit entering data (Y/N) ? n
TEAM SCORE: 300 BATSMAN 6 - RUNS SCORED: 65
Quit entering data (Y/N) ? y
TOTAL TEAM SCORE: 365
```

When the complexity of our processing needs increases, we often need to evaluate the truth-value of expressions consisting of multiple values.

3.6.3 Nested 'for' loops

In cases where a complex set of sub-related iteration constructs are required, we can use the **nested for loop** representation. When one for loop is included in the loop body of another for loop, the construct thus formed is called a nested for loop.

The general format for using the nested for loop is:

for (initializing_expression ; condition ; iteration_expression)

{

    for (initializing_expression ; condition ; iteration_expression)

    Processing statement ;

}

Example 25 illustrates the use of nested for loops:

Example 25:

```
. . .
for(i = 1; i < max1; i++)
{
:
:
 for (j = 0; j <= max2; j++)
 {
:
:
 }
}
. . .
```

Example 26 shows the entire code for implementing nested ‘**for**’ loop.

Example 26:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 int i, j, k ;
 i = 0 ;

 clrscr() ;
 cout << "Enter number of rows: " ;
 cin >> i ;
 cout << endl ;

 for(j = 0; j < i; j++)
 {
 cout << "\n" ;
 for(k = 0; k <= j; k++)
       {
 cout << "*" ;
 }
 }
}
```

Output:

```
Enter number of rows: 10
*
**
```

```
***
****
*****
******
*******
********
*********
**********
```

The program in Example 26 accepts the number of rows on which it prints the pre-defined character. The program then prints asterisks depending on the line number being processed; i.e.; line 1 will have 1 asterisk, line 2 will have 2 asterisks and so.

3.7 Simple Control Statements

While discussing the switch construct, we saw that the **break** statement is used to terminate the execution of a set of statements. In addition to the break statement, there are other statements, which force loops to behave differently.

3.7.1 Comma Operator

We have seen how the comma can be used as a value **separator** to differentiate between multiple values in the multiple initialization/increment format of the **for** loop. The comma can also be used as an **operator** in C++.

The sequence of expressions with a comma operator is evaluated one at a time from left to right. It then returns as a result, the value of the last expression.

In the expression:

expr1, expr2, expr3

The first expression to be calculated is expr1, the expr2, and finally expr3, whose value is given to the whole expression. The intermediate results of the first two sub-expressions are simply ignored.

For example,

b = 5, a = (++b), a + b

In the example, the value of the expression is 12. When the last sub-expression; that is, **a + b** is calculated, **b** has the value of 6 and **a** has the value of 6.

One reason for using this operator in place of a sequence of statements is to write the code in a shorter and more readable format. It also helps one avoid having to use additional variables to hold intermediate results. An expression containing the comma operator can appear in place of any other ordinary expression.

3.7.2 The 'break' and 'continue' Statements

The break statement used within a loop causes the loop to be terminated and forces the execution to proceed with the statements following that loop. It is used to provide an early exit from all the three forms of iteration constructs discussed.

The use of the break statement has been illustrated in the program in Example 27:

Example 27:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 int i, num ;

 clrscr() ;

 for(i = 0; i < 10; i++)
 {
 cout << "Enter any number between 1 & 10: " ;
 cin >> num ;
 cout << endl ;
if(num == 5)
     break ;
 }
 cout << "\nBreak statement executed" ;
}
```

Output:

```
Enter any number between 1 & 10: 2
Enter any number between 1 & 10: 5
Break statement executed
```

The program in Example 27 will accept 10 values from the user. In case the user wants to terminate the program execution before entering all the 10 value, he needs to enter 5. On reading 5, the **break** is executed and the **for loop** terminates displaying the message - "`Break statement executed`".

We can also use the break statement while working with nested loops. The **break** statement here causes an exit only from the current loop, that is, the one that was being executed. Example 28 illustrates the use of **break** statement.

Example 28

```
. . .
while(x <= 10)
{
 while(n < 20)
 {
 .
 .
 break ;
 }
}
. . .
```

While working with repetitive tasks, we often come across cases, which are considered as "exceptions" and cannot be processed in the same manner as other cases. In such cases, we are simply required to jump the rest of the processing and go on to the next case to be processed. The **continue** statement is an example of such an action which causes the next iteration of the enclosing loop to begin. When this statement is encountered in a loop, the rest of the statements in the loop are skipped and the control passes to the condition, which is evaluated and if true the loop is entered again.

Consider the program in Example 29.

Example 29:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 int i;
 clrscr() ;
 for(i = 1; i <= 100; i++)
 {
 if(i % 9 == 0)
```

```
  {
        continue ;
  }
  cout << i << "\t" ;
  }
}
```

Output:

```
1 2 3 4 5 6 7 8 10 11
12 13 14 15 16 17 19 20 21 22
23 24 25 26 28 29 30 31 32 33
34 35 37 38 39 40 41 42 43 44
46 47 48 49 50 51 52 53 55 56
57 58 59 60 61 62 64 65 66 67
68 69 70 71 73 74 75 76 77 78
79 80 82 83 84 85 86 87 88 89
91 92 93 94 95 96 97 98 100
```

The program in Example 29 prints all the numbers from 1 to 100 which are not divisible by 9.

Example 30 illustrates the use of the **continue** and **break** constructs in yet another scenario. The program listed accepts a set of numbers from the keyboard and the prints the sum of the positive numbers entered.

Example 30:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 int i_num, i_total = 0 ;

 clrscr() ;
 do
 {
 cout << "Enter a number (0 to quit): " ;
 cin >> i_num ;

 if(i_num == 0)
      break ;
 else if(i_num < 0)
      continue ;

 i_total += i_num ;
 } while(1) ;
 cout << "The sum of all the positive values entered: " << i_total ;
 getch() ;
}
```

Output:

```
Enter a number (0 to quit): 45
Enter a number (0 to quit): 45
Enter a number (0 to quit): 87
Enter a number (0 to quit): 0
The sum of all the positive values entered: 177
```

In the program in Example 30, the **do...while** iteration construct is used to conduct the entire processing required for the problem. The termination condition used is a dummy constant indicating that the expression will always evaluate to true. The expression evaluates whether the number entered by the user is zero. This condition is evaluated within the loop by using the **if-else-if** selection construct. If the value entered is zero, then the control is passed onto the statement immediately following the loop where the sum total of the positive numbers entered is displayed on the screen. The sum total of the positive numbers entered is updated using the variable i_total. If the number entered is negative, the **continue** statement skips the update of the sum total and causes the next iteration of the loop to begin.

3.7.3 `exit()` Function

The function **`exit()`** is used to terminate a program immediately. An **`exit()`** is used to check if a mandatory condition for a program execution is satisfied or not. The genral form of an **`exit()`** is:

`exit(int return_code)`

where the **`return_code`** is optional. **Zero** is generally used as a **`return_code`** to indicate normal program termination. Other values indicate some sort of error. The program in Example 31 illustrates the use of the **`exit()`** function.

Example 31:

```
#include <iostream.h>
#include <conio.h>
#include <process.h>

void main(void)
{
 int val = 1 ;
 char c_entry ;
 clrscr() ;
 while(val <= 50)
 {
 cout << "Val = " ;
 cin >> val ;
 cout << "Enter E to exit system immediately " ;
 cin >> c_entry ;
 if(c_entry == 'E' || c_entry == 'e')
       exit(0) ;
 }
 cout << "Exiting system..." ;
}
```

Output:

```
Val = 45
Enter E to exit system immediately T
Val = 3

Enter E to exit system immediately E
```

3.8 Check Your Progress

State whether TRUE or FALSE

1. The **if…** decision statement represents a two-way decision construct.
2. The break keyword is used to exit an **if…** decision statement.
3. It is essential to use delimiting brackets while specifying both the simple and compound processing code blocks.
4. The ***for*** loop is used as a complex iteration construct, including variable initialization, validation expression and variable modification.
5. In the ***do…while*** iteration construct, the condition is checked before the contents of the loop are executed.
6. The comma can be used both as an ***operator*** and a ***separator*** in C++.

Fill in the blanks

1. The________decision construct is used as an alternative to the multi-way **if…else…if** decision model.
2. In the switch decision construct, if none of the case conditions is true, the code specified under the________section of the construct will be executed.
3. The iteration constructs supported by C++ are________ ,________and.
4. The while loop checks the validation expression ________the loop execution begins.
5. The_______operator is used to separate values in multiple initializations/increments while using the for loop.

3.9 Do It Yourself

1. Write a program, which enables a user to input an integer. The program should then evaluate the entry and state whether it is evenly divisible by 5. The value of the integer needs to be restricted within the range of 5 and 20000.

2. Write a program which accepts two numbers and tells whether the product of the two numbers is equal to or greater than 1000.

3. Write a program, which accepts an integer and determines whether it is evenly divisible by both 6 and 7.

4. Write a program to accept a number and determine whether this entry is within a given range, say 1 to 10. This range should also be accepted from the user at the time of execution.

5. Write a program to accept three positive integers representing the sides of a triangle, and determine whether they form a valid triangle or not. Note that the sum of any two sides of a triangle must always by greater than the third side.

6. Write a program to accept two numbers, **num1** and **num2** and then determine whether the product of the two numbers is greater than or equal to num1.

7. Write a program to accept 2 values from the user. Write a menu, which lists the following binary operations:

1. Evaluate greater than or lesser than

2. Add

3. Subtract

4. Multiply

5. Divide

0. Exit

Subtraction should be allowed only when the first value entered is greater than the second value.

Division should be allowed only when the second value is not zero.

Display the results of the respective operations.

8. ARAMCO Industries International needs a program to tell us whether an employee is eligible for medical allowance or not.

According to the rules of the company, the senior employees are not eligible to medical allowance. Instead they are given a special medical insurance coverage. All other employees are given medical allowances based on their grades.

The program should accept the grade of the employee and display the percentage of the basic salary, which is offered as medical insurance.

The following table displays the medical insurance coverage for each grade.

Technical	Managerial	Allowance %
1	6	10
2	7	15
3	8	18
4	9	20
5	10	22

The grades from 11 to 15 are applicable to senior employees who are covered under the special medical insurance scheme. If the grade entered is that of a senior employee, the following message should be displayed:

An employee under this grade is covered by the special medical insurance scheme.

9. Write a program to accept 2 numbers. Calculate the difference between the two values. If the difference is equal to any of the values entered, then display the following message:

   Difference is equal to value <number of value entered>

   If the difference is not equal to any of the values entered, display the following message:

   Difference is not equal to any of the values entered

10. Write a program to calculate the discount applicable to a customer in a departmental store. The customer is eligible for a discount if his purchase is over 2000. If the customer is a regular customer (indicated by R) and the payment is made in cash (indicated by C), the discount applicable is 20%. If the customer is a regular customer and the payment is made by a credit card (indicated by D), the discount applicable is 15%. If the customer is not a regular client, the payment must be made in cash only and the discount applicable is 10%. Calculate and display the total amount payable by the customer.

11. Write a program to perform the following steps:

- Accept user input in the range of 0 to 3.
- If the user enters 0, then display the message, "**BASIC is easy**".
- If the user enters 1, then display the message, "**FORTRAN is fun**".
- If the user enters 2, then display the message, "**PASCAL may be structured**".
- If the user enters 3, then display the message, "**But C++ is the one**".
- If the user enters a value beyond this range, display the message "**Invalid entry. valid: 0 to 3.**"

Summary

- The if…else decision statement represents the two-way decision construct.
- The if…else…if…else statement is used to represent the multi-way decision construct.
- An '**if…else**' construct can contain another set of '**if…else**' statements. This is known as the **nested** '**if**' construct.
- The switch statement is used as an alternative to the multi-way if…else…if…else statement.
- The keyword break, is used to indicate the end of a sequence of processing statements under a particular case.
- Iteration constructs are used to perform repetitive tasks with a single initiation.
- The **while { }** loop checks the validation expression before entering the loop.
- The **do{ … } while (condition)** loop executes the loop at least once
- Both the while and do…while loops can also be represented in a nested format to enable multiple layers of iteration in a program.
- The **for** loop is used as a complex iteration construct, including variable initialization, validation expression and variable modification.
- The **exit** statement is used to immediately terminate the execution of a program.

Session - 4

Functions, Pointers, and Arrays

Welcome to the Session, **Functions, Pointers, and Arrays**. This session defines and describes functions, pointers, and arrays.

At the end of this session, you will be able to:

- Identify what functions do and their structure
- Discuss the arguments of a function
- Discuss return from the function and type of a function
- Identify function declaration and function prototype
- Understand the sizeof() operator
- Discuss call by value and call by reference
- Explain recursive functions and Identify storage classes
- Discuss functions in multifile programs
- Express function declaration for extern functions
- Discuss and use Pointers
- Identify single dimensional arrays
- Describe the process of initializing an array
- Identify multidimensional arrays

4.1 Introduction

Most software applications consist of several programs bundled together to give the required output. These programs can be identified as modules and these modules are represented as functions in C++. In other words, the use of functions involves breaking up of large and complex applications into smaller and manageable sub-tasks. Let us analyse the need of breaking a program into several functions.

4.2 Breaking a Program into Several Sub-tasks

Suppose we need to accept a student's marks for 6 subjects, calculate the total score and print the same along with the percentage of the student. For this, the following code will be required.

Example 1:

```
#include <iostream.h>
#include <conio.h>
void main(void)
{
:
:
:
 cout << "Enter marks for paper 1: " ;
 cin >> i_pap1 ;
 cout << endl ;
:
:
:
:
 cout << "Enter marks for paper 6: " ;
 cin >> i_pap6 ;
 cout << endl ;
 i_tot = i_pap1 + i_pap2 + i_pap3 + i_pap4 + i_pap5 + i_pap6 ;
 f_per = (i_tot*100)/600 ;
:
:
 cout << "Total score of student on 600 = " << i_tot << endl ;
 cout << "Final percentage of student: " << f_per << endl ;
 cout << "Average variance with respect to batch: " << avg_var;
}
```

The code fragment accepts the students score for six papers, calculates the total score, percentage and the average variance as compared to the average batch percentage.

Another method of representing the program is to logically break up these requirements into smaller programming units giving the same output. Hence, we can break up the program into the following logical sub-tasks:

1. Accept user input for the student's marks for 6 subjects.
2. Calculate total score and printing it on the screen.
3. Calculate percentage of the student and printing this on the screen.
4. Calculate the average variance of the student.

4.3 Benefits of Using Functions

First, let us list out the benefits of using functions.

1. The use of functions allows us to break up large tasks into smaller and easily manageable units.
2. The use of functions provides us a way of simplifying tasks that would otherwise be repetitive. A function can be defined as a set of instructions that need to be executed when it is invoked. This concept should not be confused with that of using a loop. A loop is program code, which is executed repeatedly for a specified number of times from one point within a program. If this loop is required again, later in the program, then the entire code would have to be rewritten there.

The usage of functions has been shown in Example 2.

Example 2:

```
#include<iostream.h>
void main(void)
{
:
 func1() ;
 func2() ;
:
}
func1()
{
:
}
func2()
{
:
}
```

In the code fragment given, two functions namely, **func1()**, **func2()** have been declared which are called in **main()**. The actual code for these functions is written separately after **main()**.

The program component main() is also a function. In fact, no C++ program can run without this function as it represents the starting point of all processing activities within a program.

4.4 Structure of a Function

Every function, which is defined, follows a definite pattern of representation or declaration. A function has the following distinctive attributes:

1. A name that identifies it
2. The type of the value returned by the function
3. The function's parameters and their data types
4. Processing statements

All these attributes need to be indicated properly while defining a function. The general structure to be used while defining a function is as follows:

```
return_type function_name(parameter_list)
{
     Processing statements
}
```

The function_name is the name that we use to call the function for execution in a program.

The return_type sets the data type of the value that is to be returned by the function, and can be any legal data type – including any data that we have created ourselves. If the function does not return a value, the return type is specified by the keyword void.

The parameter_list identifies what can be passed to our function from the calling function, and specifies the type and name of each parameter. A parameter name is used within the body of the function to access the item of the data that was passed by the calling function. It may be the case that a function doesn't have any parameter, which is indicated by an empty parameter list, or by the keyword void. A function that has no parameters and does not return a value would therefore, have the header:

```
void function_name()
```

Or

```
void function_name(void)
```

Before using functions, we need to realise that every function within a program represents an independent entity. Functions as explained are used to perform a set of independent processing statements. Hence, different functions represent the different sub-sections of a program. To be able to reach out to these different sub-sections of a program, we need a method by which these functions can be instructed to start processing. The method which invokes the function is referred to as "**calling**" a function.

> The program component main() is also a function. In fact, no C++ program can run without this function as it represents the starting point of all processing activities within a program.

As shown the first is the calling function, that is, the function from within which we are trying to access a sub-section of the program. The second is the "**called**" function, which is the actual sub-section or function of the program, which is called. For example, in the following code fragment, **main()** is the calling function and **func1()** is the called function.

```
main()
{
 func1() ;        Calling function
}

func1()
{
:                 Called function
 }
```

The function definition is followed by a left curly brace or opening brace '{', indicating the start of the function body. The **function body** consists of the processing statements within the braces. The end of the function is indicated by a right curly bracket or closing brace '}'.

Example 3 illustrates how the function **first_function()** can be called from **main()**.

Example 3:

```
#include <iostream.h>
#include <conio.h>
void first_function() ; // declaring function

void main(void)
{

 first_function() ; // calling function
}

void first_function() // defining function
{
 cout << "Writing my first C++ function !!!" ;
}
```

Output:

```
Writing my first C++ function !!!
```

4.4.1 Arguments

Just as a function can return data, it can also receive data from its calling function. The difference between arguments and parameters is quite subtle. A parameter appears in a function definition and specifies the data type expected by the function. An argument is the actual value passed to the function when we call it. For example,

```
void func1 (int parm1, int parm2)
```

The function arguments or parameters are indicated within parenthesis to the right of the **function_name**. When the parameters are specified, this declaration needs the specification of the parameter data type(s). There can be more than one parameter being passed and the data types of each of these parameters needs to be clearly specified **separated by a comma**.

The specification of the parameter variable names within the brackets is optional. If nothing is specified, then it is considered that there are no arguments associated with the function.

Function arguments or parameters are important in establishing a relationship between the various functions being used in the program. These arguments need to be specified in the function definition. Example 4 illustrates the use of function parameters or arguments.

Example 4:

```
#include <iostream.h>
void convert_measure(int) ; // Defining a function prototype
void main(void)
{

 int i_inches ;

 cout << "Enter measure in inches: " ;
 cin >> i_inches ;
 cout << endl ;
 convert_measure(i_inches) ; // Calling or invoking a function
}
void convert_measure(int i_inches)
{
 int i_feet = i_inches / 12 ;
 cout << "Measure equivalent in feet = " << i_feet ;
}
```

The program in Example 4 accepts a measure in inches and then converts it into feet. The measure in inches is accepted into the variable **i_inches** in the **main()** function. The **main()** function then calls the **convert_measure()** function.

The **convert_measure()** function has been defined as receiving an argument of type **int**. The data type of the variable **i_inches** is **int** and is passed to the function **convert_measure**. The **convert_measure()** function converts the inches entered, into feet and then displays the result on the screen.

Prototyping a function is the process of introducing a function to a program. It indicates what the particular reference represents in the program. It is essential to define a function prototype in a C++ program before actually being able to use the function.

In example 4, a single parameter, named **i_inches**, has been used as an argument for the function **convert_measure()**. Now suppose, our requirements change. We no longer limit the user to convert from inches to feet. Depending on his choice, we also offer him the feature to convert the measure entered into centimetres. The user thus needs to specify the measure in inches and also his choice before the **convert_measure()** function is called. This would require two arguments to be passed to the function **convert_measure()**.

Example 5 illustrates the use of multiple function arguments.

Example 5:

```
#include <iostream.h>
#include <conio.h>
void convert_measure(char, int) ;

void main(void)
{
 int i_inches ;
 char c_choice ;

 clrscr() ;

 cout << "Enter measure in inches: " ;
 cin >> i_inches ;
 cout << endl ;

 cout << "Convert to (F)eet or (C)entimetres ? " ;
 cin >> c_choice ;
 cout << endl ;

 convert_measure(c_choice, i_inches) ;
}
void convert_measure(char c_chc, int i_inch)
{
 int i_feet ;
 float f_centis ;
 switch(c_chc)
 {
```

```
 case ‘F’: i_feet = i_inch / 12 ;
 cout << “Measure in feet = “ << i_feet << endl ;
 break ;
 case ‘C’: f_centis = (2.5 * i_inch) ;
      cout << “Measure in centimetres = “ << f_centis <<
 endl ;
      break ;
 }
}
```

Output:

```
Enter measure in inches: 12
Convert to (F)eet or (C)entimetres ? F
Measure in feet = 1
```

The program in Example 5 accepts user input in the form of two elements, namely, **c_choice** and **i_inches**. Both these variables are passed to the function **convert_measure()**. The function performs the conversion calculation and displays the converted equivalent of the measure entered depending on the choice entered by the user (C for Centimetres and F for Feet).

4.4.2 Return from a Function

A function can return a value to its calling function. The data type of the return value of the function is specified to the left of the **function_name**. So far we have seen variables being passed to functions and functions returning values of the data type **void**. **void** is a keyword, which indicates that no value is being returned. We shall now see how a function can be made to return values to its calling function.

```
int func1() ;


void main(void)
{
 int ret_val ;
:
:
 ret_val = func1() ;
}
```

```
int  func1()
{
:
:
 return 1 ;
}
```

In the code fragment, the function main calls a function named **func1()**. This function returns the value 1 to the calling function using the **return** statement.

The program in Example 6 illustrates the return of values to the calling function.

Example 6:

```
#include <iostream.h>
#include <conio.h>
int convert_measure(int) ;

void main(void)
{
 int i_inches, i_feet ;
 clrscr() ;
 cout << "Enter measure in inches: " ;
 cin >> i_inches ;
 cout << endl ;

 i_feet = convert_measure(i_inches) ;
 cout << "Measure equivalent in feet = " << i_feet << endl ;
}
int convert_measure(int i_inches)
{
int i_feet ;
 i_feet = i_inches / 12 ;
 return(i_feet) ;
}
```

Output:

```
Enter measure in inches: 12
Measure equivalent in feet = 1
```

The program in Example 6 contains the definition of the function **convert_measure()** which is called by **main()**. It returns the result of the conversion to the function l**main()**. The keyword return is used to **return** the value to the function **main()**.

Note that the data type of the return value is defined as **int**. Even if no return data type was specified, the function would still be considered as returning the default return data type - **int**.

The **return** keyword is used to exit immediately from the current function. It returns control to its calling function by returning a value to the variable named `i_feet`. The general structure of the return keyword is as follows:

```
return (value) ;
```

The use of the return keyword has been illustrated in the program in Example 6.

The program in Example 7 defines a function **call_function()**, which returns no value (**void**). In the body of **call_function()**, the **return** keyword is used to immediately terminate the execution of the function and to return control to its calling function, which is **main()**. Note that there is no value being returned to **main()** as there is no argument which follows the return statement in this program.

Example 7:

```
#include <iostream.h>
#include <conio.h>
void call_function(void) ;

void main(void)
{
 clrscr() ;
 cout << "Before call_function()" << endl ;
 call_function() ; // Calling or invoking a function
 cout << "After call_function()" << endl ;
}
```

```
void call_function(void)
{
 cout << "Within call_function" << endl ;
 return ; // Returns execution control to calling function
}
```

Output:

```
Before call_function()
Within call_function
After call_function()
```

Let us consider that the same function - **call_function()**, is now defined as returning a value of type **int** (default return value). The program should thus be constructed as shown in Example 8.

Example 8:

```
#include <iostream.h>
#include <conio.h>

//Returning int (default return type) data
int call_function(void);

void main(void)
{
 int ret_val ;
 clrscr() ;
 // Calling or invoking a function
 ret_val = call_function() ;
 cout << "Value returned from call_function() = " << ret_val ;
}

int call_function(void)
{
 return 1 ; // Returns execution control to calling function
}
```

Output:

```
Value returned from call_function() = 1
```

In the program, the function **call_function()** has been defined as returning a value of type - **int**. In the function body, the return statement is followed by a constant value 1 which is returned to the calling function **main()**.

The program in Example 8 returns a constant. The same return statement can be modified to return a value contained in a variable. The program should thus be constructed would be like program in Example 9 except for the return statement in the function call_function().

Example 9:

```
#include <iostream.h>
#include <conio.h>

//Returning int (default return type) data
int call_function(void);

void main(void)
{
 cout << "Value returned from call_function() = " << ret_val ;
}

int call_function(void)
{
 int ret_val=1;

 return ret_val;//Returns execution control to calling function
}
```

Output:

```
Value returned from call_function() = 1
```

The function **call_function()** assigns a value '1' to an integer variable **ret_val**. The value in **ret_val** is then returned to the calling function, **main()**. The value which is returned by **call_function()** is however not being used by the calling function. Example 10 illustrates how to use values returned from functions.

Example 10:

```
#include <iostream.h>
#include <iostream.h>
void main(void)
{
 int ret_val ;
 // Returning int (default return type) data
 int call_function(void) ;
 // Calling or invoking a function
 ret_val = call_function() ;
 cout << "The returned value = " << ret_val << endl ;
 ret_val = ret_val + 1 ; // Using the returned value
 cout << "After using the returned value: "<< ret_val<< endl ;
}

int call_function(void)
{
 int ret_val = 1 ;
 return ret_val ;
 // Returns execution control to calling function
}
```

Output:

```
The returned value = 1
After using the returned value: 2
```

The **call_function()** returns the value in variable **ret_val** to the **main()** function. The same value is then assigned to the variable **ret_val** defined in **main()**. 1 is added to **ret_val** and the result is displayed to illustrate the use of the returned value. The examples, which have been discussed so far, have shown how to return and further use variables of the type **int**.

The program in Example 11 illustrates how to return a value of type **char**.

Example 11:

```
#include <iostream.h>
void main(void)
{
 char char_ret_val ;
 char call_function(void) ;
 char_ret_val = call_function() ;
 cout << "Value returned = " << char_ret_val ;
}
char call_function(void)
{
 return('G') ; // Returning a character constant
}
```

This program shows how to return a character constant 'G' to the calling function.

4.5 Scope of Variables Within Functions

The scope of a variable determines what parts of the program have access to it. A variable's scope depends on where that variable is declared. Let us consider the following scenario. The identity card of a college student is applicable only within the college, issuing it. A student holding an identity card of college A cannot enter college B. The student would thus need to have an identity card for college B too.

The area where a respective college identity card is applicable can be referred to as its **scope of influence**.

The scope of a variable is similar to the scope of influence of the identity card as discussed. Just as the scope of influence of an identity card refers to the college in which it can be used, the scope of a variable refers to the range within a program where that variable has meaning.

So far in all the code fragments we have used a single function – **main()** and all the data variables have been declared within this function. However, programs normally require the use of multiple functions (other than **main()**). The scope of variables is classified into two basic categories, namely **local and global variables**.

Variables declared within functions are called **local variables**. Local variables thus have meaning only within the function in which they are declared. Example 12 illustrates the declaration of a local variable **l_var** within function **main()**.

Example 12:

```
void main(void)
{
 int l_var ;
func1() ;
:
:
}
func1()
{
:
:
}
```

In this case, the variable **l_var** is accessible only in function **main()** and is not accessible within **func1()** which is declared outside the function **main()**.

Variables, which are not declared within a function, are called **global variables**. They are accessible from all the functions within a given program. Example 13 illustrates the declaration of a global variable **g_var**.

Example 13:

```
int g_var ;
void main(void)
{
 g_var = 2 ;
 int call_func1() ;
}

int call_func1()
{
 g_var = 1 ;
:
:
}
```

In the code fragment, the variable **g_var** has been declared outside the functions, **main()** and **call_func1()**.

Thus it is hence, from both the functions. Hence, all the functions defined in main() and outside main() can access the global variables defined in the program.

Example 14 illustrates the use both local and global variables n used.

Example 14:

```
#include <iostream.h>
#include <conio.h>

void using_globalvar(void);



int gi_area; // Declaring a global variable

void main(void)
{
 int li_len, li_wid; //Declaring local variables

 cout << "Enter the length of a rectangle: " ;
 cin >> li_len ;
 cout << "\nEnter the width of a rectangle: " ;
 cin >> li_wid;

 // Assigning value to a global variable
 gi_area= li_len * li_wid;

 using_globalvar() ; //Function call
 }
void using_globalvar(void)
{
cout << "\nArea = " << gi_area; // Global variable
 getch() ;
 return ;
}
```

Output:

```
Enter the length of a rectangle: 12

Enter the width of a rectangle: 12

Area = 144
```

In Example 14, **gi_area** has been declared as a **global** variable. The function, **using_globalvar()** has been used to show the use of global variables over multiple functions in a program. The length and width of the rectangle as entered by the user are stored in local variables **li_len** and **li_wid** respectively. The global variable, **gi_area** has been assigned the area of a rectangle based on the entries made by the user. This variable is accessed by the function **using_globalvar()** and its value is then displayed on the screen. The control is then returned to the calling function – **main()**.

4.6 Scope Rules for a Function

We have seen that variables are influenced by rules defining their **scope**. Similarly, functions also follow scope rules depending on where the function is defined.

If a function is declared within a function body, its scope is limited to the function within which it is defined. Such a function becomes locally accessible.

Example 15 illustrates the scope rules pertaining to functions:

Example 15:

```
void main (void)
{
 void fun1 (void) ; // This function is local to main ()
 fun1 () ;
 fun2 () ; // Erroneous function call
}


void fun1 (void)
{
 void fun2 (void) ; // This function is local to fun1 ()
 fun2 () ;
}
```

The program in Example 16 illustrates a local function.

Example 16:

```
#include <iostream.h>

void main(void)
{
 void local_to_main(void) ;
 local_to_main() ;
 local_to_local() ; // Erroneous call to function
}

void local_to_main(void)
{
 void local_to_local(void) ; // Declaring a local function
 cout << "This is function local_to_main" ;
 local_to_local() ;
}

// This function is local to function local_to_main()
void local_to_local(void)
{
 return ;
}
```

The program in Example 16 will generate the following compiler error:

```
"Function 'local_to_local' should have a prototype in the function main()"
```

This error is due to a call to function **local_to_local()** from function **main()**. This is because the prototype of the function is defined in function **local_to_main()**. The function **local_to_local()** is thus local only to function **local_to_main()** .

This scenario can corrected by changing the scope definition of the function **local_to_local()**. The function can be defined as a global function to allow accessibility throughout the program. The same is illustrated in Example 17.

Example 17:

```
#include <iostream.h>

void local_to_local(void) ; // Defining global function
void main(void)
{
 void local_to_main(void) ;
 local_to_main() ;
 local_to_local() ;
}

void local_to_main(void)
{
 cout << "This is function local_to_main" << endl ;
 local_to_local() ;
}

void local_to_local(void)
{
 cout << "This is function local_to_local" << endl ;
 return ;
}
```

If a function is declared outside any function, then it becomes accessible to all of the functions within the program. Such functions are referred to as global functions. Function ***local_to_local()*** in the program is a global function.

4.7 Formal Parameters and Default Arguments

When a function is called, the parameter specified is known as an **actual** parameter or argument. This is the value, which is transmitted to the called function. The **formal** parameter is the name of the corresponding **actual** parameter in the called function. The program in Example 18 illustrates what is a formal parameter.

Example 18:

```
#include <iostream.h>


void main(void)
{
 void test_function(int) ;
 test_function(1) ; // actual parameter
}
void test_function(i_recd_value) // formal parameter
int i_recd_value ;
{
 cout << "Value passed = " << i_recd_value ;
 return ;
}
```

Output:

```
Value passed = 1
```

In the program illustrated in Example 18, the called function is **test_function()**. It has been declared as receiving a parameter of type **int** from its calling function. In the calling statement, the constant value 1 is passed to the function. This value has to be received by the called function in a variable. This receiving variable named **i_recd_value** is called the formal or dummy parameter for the function **test_function()**. The declaration of the formal parameter must either be done within the parentheses following the function name or between the function definition and the opening brace of the function body. Undeclared formal parameters are assumed by C++ to be of type **int**. There should be no semi-colon or statement terminator between the function name specification statement and the variable declaration.

A **default argument** is a value, which is automatically assigned to a formal variable, if the actual argument from the function call is omitted. This default argument is given in the function prototype. The use of a default argument is illustrated in Example 19.

Example 19:

```
#include <iostream.h>

int called_function(int i1 = 1, int i2 = 1, int i3 = 1, int i4 = 1) ;

void main(void)
{
 called_function(10, 10, 10, 10) ; // Call with parameters
 called_function() ; // Call without parameters using default parameters
}

int called_function(int i1, int i2, int i3, int i4)
{
 cout << i1 << endl << i2 << endl << i3 << endl << i4 ;
 return 1 ;
}
```

A default value for a particular argument cannot be given unless all default values for the arguments to its right are given. This is quite logical, as if there are some arguments missing in the middle then the compiler would not know the arguments, which have been specified and which should be taken as default.

```
void called_function(int i1=1, int i2=1, int i3=1, int i4=1); //Valid
void called_function(int i1, int i2, int i3, int i4=1) ; // Valid
void called_function(int i1=1, int i2=1, int i3=1, i4) ; // Invalid
```

Default arguments are useful when the arguments have the same value most of the time.

4.8 The sizeof Operator

The **sizeof** operator is a unary compile-time operator. It returns the length, in bytes of the variable it precedes. Variables of different data types require different amounts of memory for storing data, for example, char variables require 1 byte, `int` variables require 2 bytes, float variables require 4, double 8 and so on. To ascertain the storage space required for a variable, we need to use the **sizeof** operator.

Example 20 illustrates the use of the **sizeof** operator.

Example 20:

```
#include <iostream.h>
void main(void)
{
float f;
cout << "Size of float = " << sizeof f ;
cout << "\nDouble = " << sizeof(double) ;
}
```

Output:

```
Size of float = 4
Double = 8
```

In the example in Example 20, we see that the program is used to display the size required to store the variable **f** and computes the size required to store a variable of type **double**.

As shown in Example 20, to compute the size of a data type, the type name has to be enclosed in parentheses. This is not necessary for variable names, although enclosing them does not make a difference. The **sizeof** operator becomes useful when we need to define storage areas in memory at run time on the basis of data entered.

4.9 Recursive Functions

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. This process is used for repetitive computations in which each action is stated in terms of a previous result. Many iterative (that is repetitive) problems can be written in this form.

In order to solve a problem recursively, two conditions must be satisfied

- **the problem must be written in a recursive form**.
- **the problem must include a stopping condition**.

The program in Example 21 calculates the factorial of a positive integer by *using recursion*.

Example 21:

```
# include <iostream.h>
long int calculate (int);

main ()
{
 int number;
     cout << "\n NUMBER = " ;
     cin >> number ;
     cout << endl ;

     /* Calculate and display the factorial */
     cout << "\n" << number << "! = " << calculate (number) ;
}

long int calculate (int num)
{
if  (num  <=  1)
           return (1);
      else
           return(  num  *  calculate  (num  -  1));
}
```

In **main()** a number is accepted and then a function calculate() is called. This function is a recursive function, that is, it calls itself **recursively**, with an actual argument (**num - 1**) which decreases in count for each successive call. The recursive calls terminates when the value of the actual argument becomes **1**.

When the program is executed, the function calculate() will be accessed repeatedly, once in main() and then (n-1) times within itself, though the user will not be aware of this.

4.9.1 Execution of a Recursive Function Call

Recursive functions are placed on a **stack** until the condition that terminates the recursion is encountered. The intermediate values are stored in the memory and the final result is calculated only at the end of the recursion process using these stored values.

A **stack** is a last-in, first-out (LIFO) data structure. A LIFO data structure is one in which each successive data item is "**pushed down**" upon preceding data items. The data items are later removed (that is, they **are** "**popped**") from the stack in **reverse order**, as indicated by the **last-in**, **first-out** designation. Figure 4.1 illustrates the PUSH and POP operation of a stack.

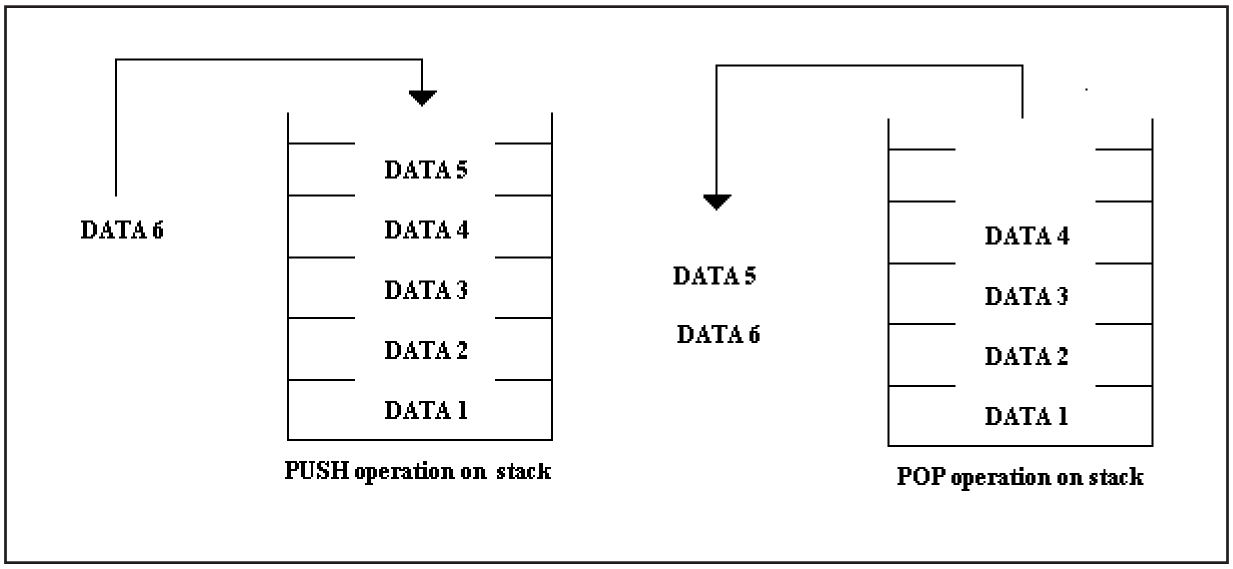


Figure 4.1: Stack (Last In First Out)

The function calls are then executed in reverse order, that is, as they are "**popped**" off the stack. Thus, when evaluating a factorial recursively, the function calls will proceed in the following order:

n! = n * (n - 1)!

(n - 1)! = (n - 1) * (n - 2)!

(n - 2) = (n - 2) * (n - 3)!

....

....

2! = 2 * 1!

1! = 1

The data will be pushed down the stack in the order as shown earlier.

The actual values will then be returned in the following reverse order:

1! = 1

2! = 2 * 1! = 2 * 1 = 2

3! = 3 * 2! = 3 * 2 = 6

4! = 4 * 3! = 4 * 6 = 24

....

....

n! = n * (n - 1)! = ...

This reversal in the order of execution is characteristic of all functions executed **recursively**.

If a recursive function contains local variables, a **different set of local variables will be created for each call to the function**. The names of the local variables will of course, always be the same, as declared within the function. However, the variables will **represent a different set of values each time the function is executed**.

Each set of values will be stored on the stack, so that they will be available as the recursive process "**unwinds**", that is, as the various function calls are "**popped**" off the stack and executed.

4.10 Storage Classes

Storage classes define the manner in which data is stored in a given variable. They define how storage is allocated to a variable. So far, we have discussed two types of variables namely, local and global. Both these classifications define the manner in which the data is stored in their corresponding variables.

Depending on this basic classification of data storage in variables, there are four basic storage classes, which can be identified as:

- ➔ auto
- ➔ static
- ➔ extern
- ➔ register

4.10.1 Auto Storage Class Variables

Auto storage class variables are created each time a function executes and disappear when the function terminates. In other words, the duration of an automatic variable is the duration of the function in which it is declared and the scope is the function's code. It can be accessed only in that function. Local variables are by default classified as **auto** variables, because they are allocated automatically when a function is executed and deallocated automatically when the function terminates. Hence, automatic variables cannot retain their values over a number of function calls to that function.

To declare a local variable as automatic, the keyword **auto** is placed at the head of the declaration, as in

```
auto int i_value ;
```

Since auto is the default storage class for local variables, this keyword is usually omitted. It should also be noted that global variables cannot be declared as **auto**.

4.10.2 Static Storage Class Variables

Static variables also have the same scope as automatic variables, that is, they can be accessed only within the function in which they have been declared. Unlike automatic variables, static variables retain their values over a number of function calls. The life of a static variable starts, when the first time the function, in which it is declared, is executed and it remains in existence, till the program terminates.

To declare a local variable as static, the keyword static is placed at the head of the declaration, as shown.

```
static int i_value ;
```

If a static local variable is assigned a value the first time a function is called, that value remains there when the function is called the next time. The use of the static storage class has been illustrated in the program in Example 22. This example also illustrates the difference between **automatic** and **static** variables.

Example 22:

```
#include <iostream.h>
#include <conio.h>
void print_auto(void)

{
 static int i = 0 ;
 auto j = 0;
 cout << "\nValue of i before incrementing = " << i ;
 cout << "\nValue of j before incrementing = " << j ;
 i += 10 ;
 j += 10 ;
 cout << "\nValue of i after incrementing = " << i ;
 cout << "\nValue of j after incrementing = " << j ;
}
```

```
void main(void)
{
 clrscr() ;
 print_auto() ;
 cout<<"\n";
 print_auto() ;
 cout<<"\n";
 print_auto() ;
}
```

Output:

```
Value of i before incrementing = 0
Value of j before incrementing = 0
Value of i after incrementing = 10

Value of j before incrementing = 10
Value of i before incrementing = 10

Value of j before incrementing = 0

Value of i after incrementing = 20
Value of j before incrementing = 10
Value of i before incrementing = 20
Value of j before incrementing = 0
Value of i after incrementing = 30

Value of j before incrementing = 10
```

4.10.3 Extern Storage Class Variables

Different functions of the same program can be written in different source files and can be compiled together. A global variable is a variable whose value can be accessed and modified by any function in the program.. However, the scope of a global variable is limited to only those functions, which are defined in the physical source file in which it is declared. If a global variable declared in a source file is required in a function defined in another source file, then external variables are used. External variables are declared with the keyword **extern**. Any changes to the **extern** variables done by a function of a file are reflected in the functions of all other files that use the variable. An example has been given in Example 23.

Example 23:

```
/* FILE 1.CPP

VARIABLE g IS GLOBAL, AND THEREFORE, CAN BE USED ONLY IN FUNCTIONS DEFINED
IN FILE 1.CPP; i.e.; main() and fn1().

*/

int g = 0 ;

void main(void)

{

:

}

void fn1()

{

:

}

/* FILE 2.CPP

IF VARIABLE g FROM FILE 1.CPP IS REQUIRED IN FUNCTIONS DEFINED IN FILE 2.CPP,
THEN IT HAS TO BE DECLARED AS extern.

*/

extern int g ;

void fn2()

{

:
}
void fn3()
{
:
}
```

If a variable is declared as **extern** in a function, then its scope is local to that function. If it is declared outside a function, such as in the example, then its scope is global in that source file.

4.10.4 Register Storage Class Variables

The register storage class can be specified only for local variables. The computer contains a small number of registers, or individual storage units. These are similar to the primary memory except that they are part of the computer's microprocessor itself. Hence, operations on registers are run faster than those involving memory. The register storage class is thus used in programs for which speed is important.

A variable declaration with the register storage class is a "request" to the compiler that the variable be assigned to a register rather than to a location in memory.

The standard declaration construct for a register variable is:

```
register int number ;
```

Since a register variable is not located in memory, it is illegal to apply the address operator (&) to a register variable. This is based on the assumption that all register variables are stored in registers, although this may not be the case (depending on the availability of registers). In other words, a register is assigned by the compiler, only if it is free, otherwise it is taken as an automatic variable.

4.11 Functions in Multi-file Programs

Programs can be composed of multiple files. Such programs could make use of lengthy functions, where each function may occupy a separate file. As with variables in multifile programs, functions can also be defined as **static** or **external**. The scope of the external function is through all files of the program and it is the default storage class for functions. Static functions are recognized only within the program file and their scope does not extend outside it. The function header will look like:

```
      static fn_type fn_name (argument list)
or
      extern fn_type fn_name (argument list)
```

The keyword extern is optional as it is the default storage class.

4.12 Definition of a Pointer

Any value in the memory can be accessed in two ways:

➔ By referring to its **variable name**.

➔ By referring to the **memory address of its location**.

So far, we have used the first method of accessing a value in memory, that is, by referring to its variable name. We will now see how memory address can be accessed directly and its usefulness in programming.

To implement the model of reference used by pointers, we need to learn what are **memory address.** Every variable in the program has an address in memory – this is the location in the memory of the computer where the data is stored. Similarly, in order to execute, the functions that the program uses must be located somewhere in memory; so a function has an address, too. These address depend on where the program is loaded into memory when we choose to run it.

When we refer to a variable by its name, we indirectly refer to a particular memory address for the specified data value.

While using a pointer, the address of a given variable is directly used. It is this address, which is then used for further manipulations and calculations on the variable. In other words, by using the address of a variable, we can work with the value stored there, as we would work with the variable itself.

A variable, which holds data of type char, is fundamentally different from a variable, which holds data of type **int**. A pointer is a variable that we can use to store a memory address. The address stored in a pointer usually corresponds to the position in memory where a variable is located, but it can also be the address of a function. A pointer variable is defined using the asterisk (*) operator. A variable holding the data of the type integer needs two bytes of space to store the value of the variable. Referring to figure 4.1, an integer variable A is defined with a value of 10. A pointer to an integer is then defined and is assigned the address of the variable A. Since the A is an integer variable it occupies two bytes of storage.

Figure 4.2 depicts a representation of a pointer.

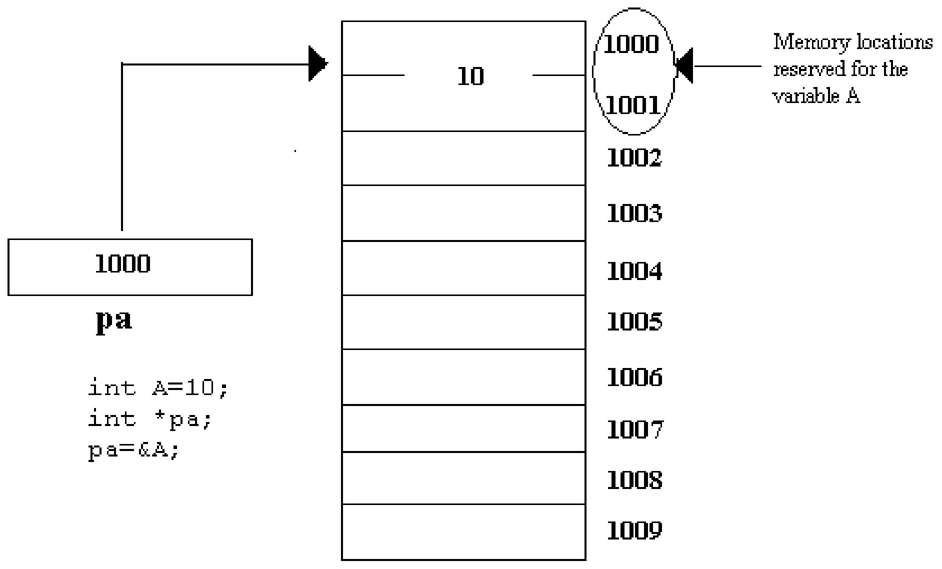


Figure 4.2: Defining a Pointer

The following code fragment defines an integer pointer named **p_int**.

```
int *p_int ;
```

Pointer variables are declared like any other variable, except that they are preceded by the asterisk (*) notation.

For example,

```
    int *i _pointer ;

    char *c_pointer ;
```

Consider the following,

```
    char v, *pv // statement 1

    v = 'A' ; // Statement 2

    pv = &v ; // Statement 3
```

Figure 4.3 depicts a pointer reference.

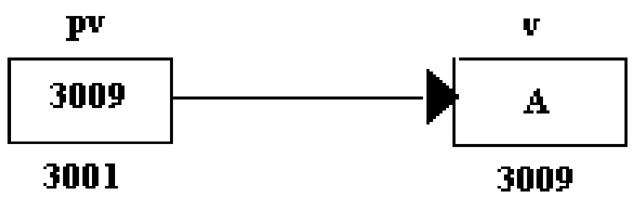


Figure 4.3: Pointer Reference

- ➔ In statement 1, v is declared as a **char** type variable and ***pv** as a **pointer variable**. This variable (***pv**) is declared to hold the **address** of a variable of type **char**.

- ➔ In statement 2, the value '**A**' is assigned to the variable **v**.

- ➔ In statement 3, the pointer variable **pv** is initialized with the **address** (not the value) of the variable **v**.

  Two special characters ' * ' and ' **&** ' have implications in pointer operation. * operator returns the value stored at the address that it. & operator returns the address of the variable it precedes. For example.

  int ***pv** = **&v** can now be explained as, pv contains the address of the variable v.

  The following code fragment assigns the address of variable **i_code** to the pointer variable **p_int**.

```
p _int = &i _code ;
```

4.12.1 Understanding lvalues and rvalues

The **lvalue** of a variable is the address of where the variable is stored in memory. The **rvalue** of a variable is what is stored at that memory address. Hence, **lvalue** (left value) is the memory address while **rvalue** is the content stored at the **lvalue** memory address.

From the compiler's point of view, the lvalue of a variable cannot be modified once it has been assigned. This is because the lvalue is the only way the compiler can find where the variable is stored in memory. The rvalue on the other hand, is the value that the variable contains. You can change the rvalue of a variable whenever you need to.

4.12.2 Fundamentals of Pointers

Within the memory of computer every data item stored, occupies one or more **contiguous memory cells** (adjacent words or bytes). The number of memory cells required to store a data item depends on the **type** of data item. For example,

- a single **character** will typically be stored in **1** byte (8 bits) of memory
- an **integer** usually requires **2** contiguous bytes
- a **floating** point number may require **4** contiguous bytes
- a **double**-precision quantity may require **8** contiguous bytes

When a **pointer** is pointing to any of these data types, it generally refers to the **first byte** of the "**contiguous**" bytes.

Suppose **v** is a variable that represents some particular data item. The compiler will automatically assign memory cells for this data item. The data item can be accessed if we know the location (memory address) of the first memory cell.

Figure 4.4 shows the memory representation for different statements of program code with pointers. Each statement in the figure is explained.

- **int *ptr;:**

  ***ptr** is declared as a pointer variable which will point to an integer variable. At this point ***ptr** has just been declared and is not pointing to anything. Variable ***ptr** itself is located at an address **0xfff1** in the memory.

- **int custcode = 15286:**

  This statement is a combination of declaration cum initialization. **custcode** is defined as an **int** type variable having the value **15286**, and is located at an address **0xfff4** in the memory.

- **ptr = &custcode;:**

  The pointer variable ***ptr** which was earlier declared, is now initialized to the **address of custcode**, which is **0xfff4**. In other words, **ptr** is now pointing indirectly to the value **15286** through its address **0xfff4**.

- **cout << "Custcode = " << custcode** ;:

  The value of custcode 15286, is accessed through the variable 'custcode', and not through its memory address.

- **cout << "Custcode = " << *ptr** ;:

  The value of custcode is now accessed not through the variable 'custcode' but through its **memory address 0xfff4**, which is stored in the pointer variable **ptr**.

- **cout << "Address is " << &custcode** ;:

  The memory address **0xfff4** of the variable **custcode** is printed using '**&custcode**'. The preceding **ampersand** sign indicates the **address** of the succeeding variable, and **not** its value. The variable '**custcode**', without the preceding ampersand indicates the value **15286** stored in it.

- **cout << "Address is " << ptr** ;:

  The memory address **0xfff4** of the variable **custcode** is printed using **ptr**, without the asterisk (*), that is, the value contained in **ptr**.

Figure 4.4 illustrates the concept that is discussed earlier.

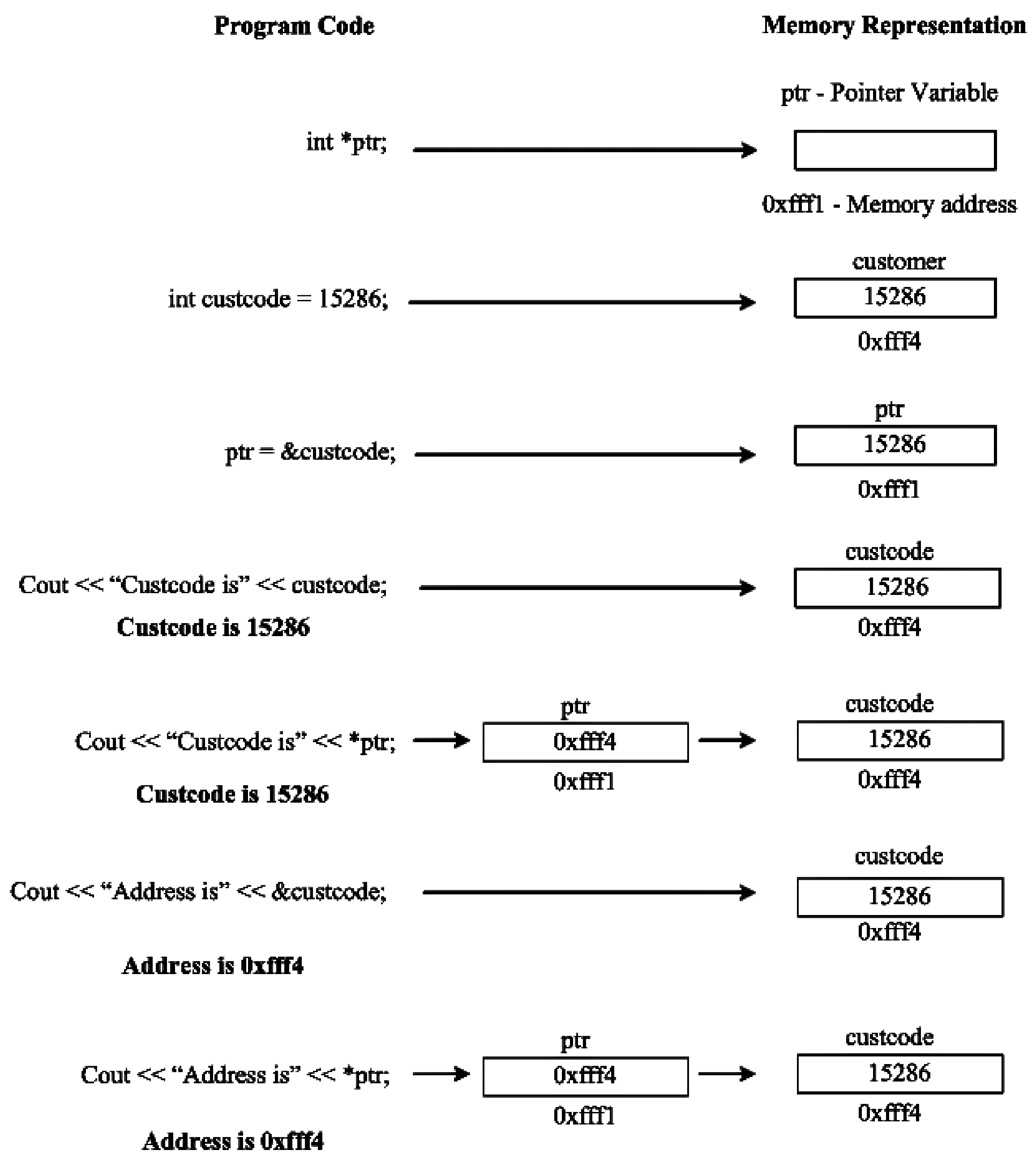


Figure 4.4: Memory Representations of Variables

4.13 Call by Value and Call by Reference

We have seen that data can be exchanged between functions by using parameters. These parameters are represented as variables. The method using only value of the variables to exchange data is known as **call by value**. In this case any change made to this variable in the called function (also referred to as child function) is **not reflected** back in the calling function (also referred to as parent function). The change remains **local** to the child function.

An example of **call by value** is given in the following code fragment.

Example 24:

```
void main(void)
{
 int data ;
:
:
 call_function(data) ;
}
void call_function(int data)
{
 cout << data ;
}
```

In the code fragment, we see that that the function **main()** calls the function **call_function()** using the integer variable **data** as the parameter.

This case can also be represented using **call by reference** where the parameter passed to the **call_function** is the address of the variable **data** rather than the value of the variable itself.

The same may be represented as given in the following code fragment.

Example 25:

```
void main(void)
{
 int data ;
:
:
 call_function(data) ;
}
void call_function(int data)
{
 cout << data ;
}
```

In the code fragment, the address of the variable **data** is passed to the function **call_function()** which accepts the address in a pointer variable **p_data**.

Hence, in **call by reference**, instead of the variable, its address is passed as an argument, and any changes made to the value at this address in the child function is instantly reflected back in the parent function.

Consider the program given in Example 26:

Example 26:

```
#include<iostream.h>

void funct1(int u, int v); // function declaration
void funct2(int *pu, int *pv); // function declaration

void main(void)
{
 int u = 1;
 int v = 3;
 cout << "\nBefore funct1 u = " << u << ", v = " << v ;
 funct1(u, v);
 cout << "\nAfter funct1 u = " << u << ", v = " << v ;
 cout << "\nBefore funct2 u = " << u << ", v = " << v ;
 funct2(&u,&v); /* addresses of u and v are passed */
 cout << "\nAfter funct2 u = " << u << ", v = " << v ;
}
void funct1(int u, int v)
{
 u = 0;
 v = 0;
 cout << "\nWithin funct1 u = " << u << ", v = " << v ;
 return;
}
void funct2(int *pu, int *pv)
{
 *pu = 0 ;
 *pv = 0 ;
 cout << "\nWithin funct2 *pu = " << *pu << ", *pv = " << *pv ;
 return ;
}
```

Output:

```
Before funct1 u = 1, v = 3
Within funct1 u = 0, v = 0
After funct1 u = 1, v = 3
Before funct2 u = 1, v = 3
Within funct2 *pu = 0, *pv = 0

After funct2 u = 0, v = 0
```

This program contains two functions, called **funct1** and **funct2**. The first function, that is, **funct1**, receives two integer variables as arguments. These variables are originally assigned the values 1 and 3 respectively. The values are then changed to 0, 0 within **funct1**.

However, the new values are not recognized in **main()**, because the arguments were passed by value, that is, any changes made to the arguments in the function **funct1** are local to the function in which the changes occur.

Now consider the second function, **funct2**. This function receives two pointers to integer variables, as its arguments. The contents of the pointer addresses are reassigned the values 0, 0. Since the addresses are recognized in both **funct2 and main()**, the reassigned values are reflected in **main()**. Therefore, both the variables u and v now contain 0.

A diagrammatic representation of the program discussed in Example 26 is illustrated in Figure 4.5.

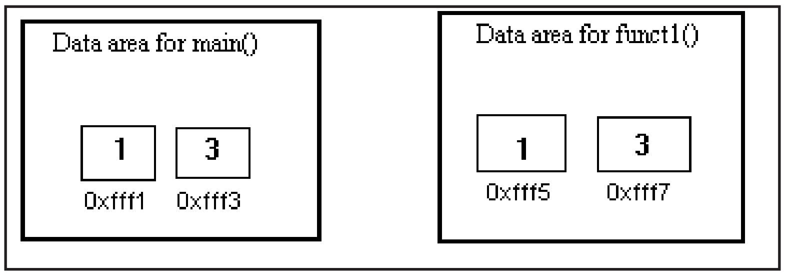


Figure 4.5: Pointing Variables Value

After **funct1** is called, separate copies of '**u**' and '**v**' are created at different memory locations **0xfff5** and **0xfff7** respectively.

Figure 4.6 illustrates the situation after the execution of **funct1()**.

After control passes back to main from funct1:

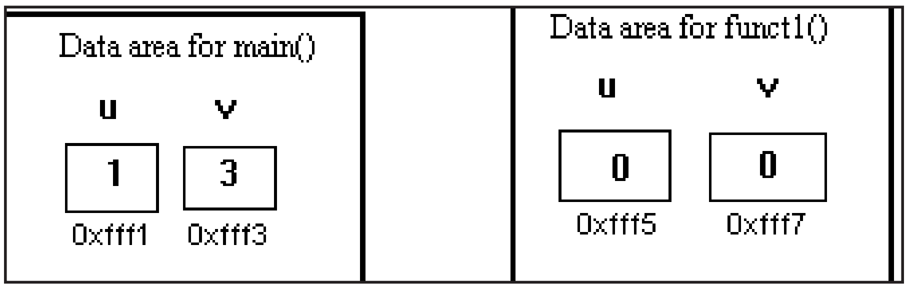


Figure 4.6: Memory Structure After Code Execution

After control passes back to **main()**, the changed values of variables **u** and **v** at addresses **0xfff5** and **0xfff7** inside **funct1()** have no effect on the values of **u** and **v** at addresses **0xfff1** and **0xfff3** in **main()** because these are local to **main()**.

The values of u and v inside funct1 have been changed to 0.

Figure 4.7 shows the outcome after control passes back to main from funct1.

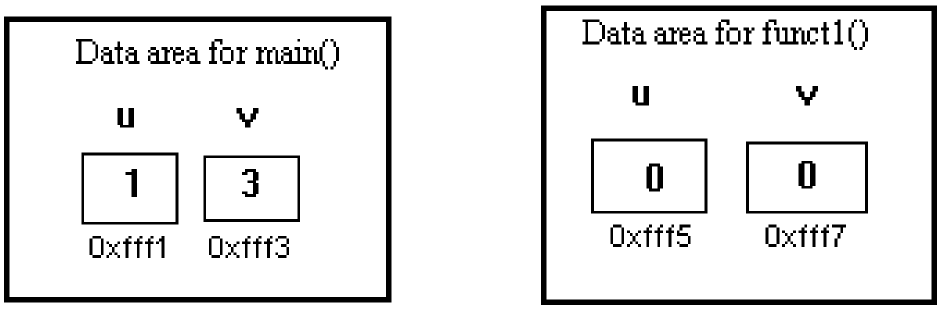


Figure 4.7: Operation From func1()

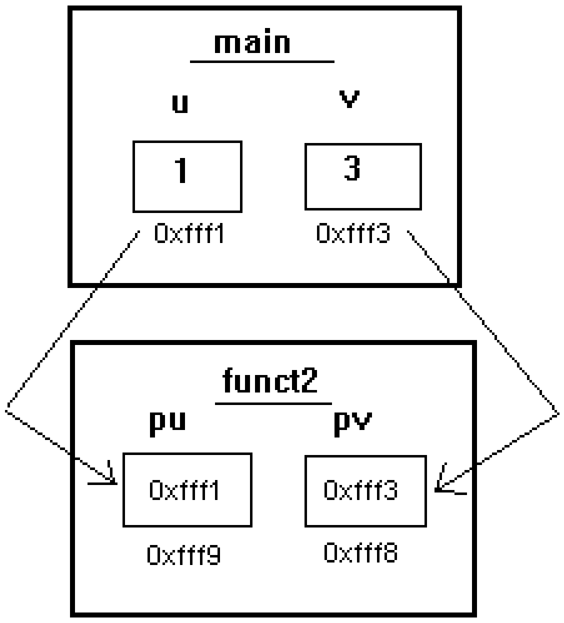


Figure 4.8: Address Reference of Variables

After funct2 is called from **main(), as shown in figure 4.8**, the addresses or **u** and **v** are passed using the **&** sign which are received in the pointer variables **pu** and **pv** respectively. This means that **pu** is now indirectly pointing to the value 1 and **pv** is now indirectly pointing to the value 3.

Figure 4.9 shows the outcome after funct2 is executed.

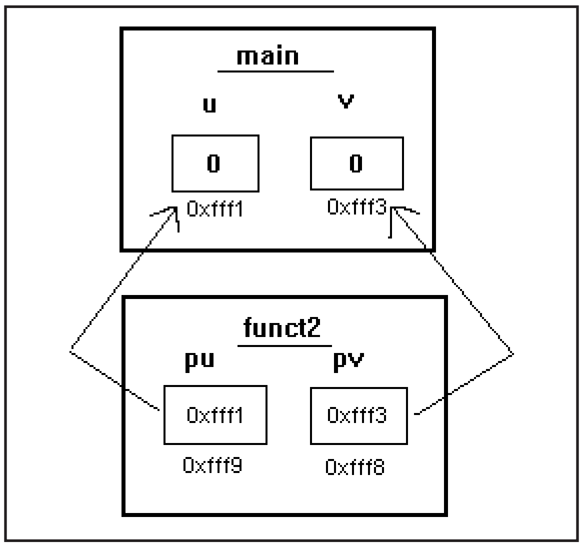


Figure 4.9: Call by Reference Mechanism

After funct2 is executed, since the physical addresses of u and v are passed from main() to funct2, any changes made in funct2 to the values at these addresses will automatically be reflected back in main(). This is known as call by reference.

4.14 Introduction to Arrays

If a large amount of data were to be stored, it would be a long and tedious task to think of a separate variable name for each data element. Consider the case if we were to store the score of the player for 25 different innings. Our code would thus look somewhat like:

```
int i_score1, i_score2, i_score3, …. i_scoreN

i_score1 = 45 ;

i_score2 = 35 ;

i_score3 = 50 ;

i_score4 = 0 ;

:

:

:

i_score25 = 45 ;
```

In the example given, note that declaring 25 different elements to store 25 different scores becomes a very tedious task. It is an equally painstaking job to try to assign 25 different values to 25 different variable names. Using this method of data management would make the job of programming an extremely difficult task and this method of storing data may serve the purpose as long as the number of data elements to be stored is known. However, this method does not allow for the common case where there are an indeterminate number of values, which have to be operated upon or if we need to add large number of values.

The use of arrays provides a solution to this problem. Arrays form an important part of any programming language structure and are hence, included in every high-level programming language.

4.14.1 What is an Array?

An array is a list of variables that are all of the same type and are referenced through a common name. An individual variable in the array is called an array element. Arrays are thus used to store a large variety of common data under a single reference. For example, suppose a batsman scores the following runs in 8 innings:

Score
34
54
34
54
65
34

21

0

To store these scores in the order of their occurrence, a serial number should be attached to assert the order of precedence. This order can be indicated by means of a subscript. For example,

Score1 = 34

Score2 = 54

Score3 = 34

Score4 = 54

Score5 = 65

Score6 = 34

Score7 = 21

Score8 = 0

and so on.

To convert the scenario into C++ syntax, we need to use the array name (user-defined) followed by the square braces '[ ]'. The subscript as shown defines the position of the corresponding elements in the array. This subscript is put into the square braces to populate the array. The general form for an array is

```
      type_specifier variable_name [elements];
```

where:

type_specifier	**Can be C++ data type (char, int, double, etc.)**
variable_name	**Assigns name to the array variable**
elements	**Specifies the number of data units in the array**

When you define an array variable, you must necessarily supply the number of elements you want the array to have.

Thus the scores of the player over 8 innings will be represented as follows:

```
void main(void)
{
// Declaring an array of type integer
 int score[8] ;
// Populating the array
 score[0] = 34 ;
 score[1] = 54 ;
 score[2] = 34 ;
 score[3] = 54 ;
 score[4] = 65 ;
 score[5] = 34
 score[6] = 21
 score[7] = 0
:
:
}
```

In the code fragment, the array has been declared of type integer because we want to store the scores, which are integer values. While defining/declaring an array, we basically create a series of memory locations reserved for values of the integer data type (in this case 8 such memory locations would be reserved). In other words, an array declared to be of type `int`, cannot contain elements that are not of type integer.

An important point to note is that, while assigning the subscript to a particular value in an array in C++, the assignment starts from the number 0 and not 1 as in most other programming languages.

4.15 Initialization of an Array

If we want to represent a set of five numbers, say (**35,40,20,57,19**), using an array variable **number**, then we may declare the variable **number** as shown in figure 4.10:

```
          int number[5];
```

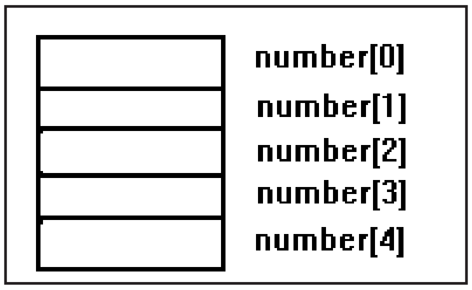


Figure 4.10: Array Index

The values to the array elements can be assigned as follows:

```
    number[0] = 35;
    number[1] = 40;
    number[2] = 20;
    number[3] = 57;
    number[4] = 19;
```

This would cause the array **number** to store the values as shown in figure 4.11:

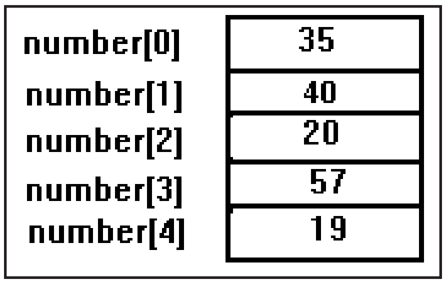


Figure 4.11: Array Initialization with Value

These elements may be used in programs just like any other variable. For example,

```
a = number[0] + 10;
```

```
Hence, a = 45, since number[0] = 35
```

```
number[4] = number[0] + number[2];
```

Hence, number[4] is now replaced with the new value 55 as number[0] = 35 & number[2] = 20

Given are examples which illustrate how to initialize arrays at the time of declaration.

```
        int digits[4] = {3,78,17};

        float x[4] = {-89.7,0,29.34,-2.0};
```

Consider the following code fragment:

```
        int x[5] = {3, 5, 7, 6, 4} ;
```

x, is a 5 element array holding the values 3, 5, 7, 6 and 4. The first element is referred to as **x[0]**, the second as **x[1]** and so on. The last element will be **x[4]**. The subscript associated with each element is shown in square brackets. Thus, the value of the subscript for the first element is **0**, the value of the subscript for the second element is **1** and so on. For an n-element array, the subscripts always range from **0** to **n-1**.

4.16 Character Arrays

This type of array is generally used to represent a string. Each array element will represent one character within the string. Thus, the entire array can be thought of as an ordered list of characters.

Note that an n-character string will require an (**n+1**) element array, because the **null character (\0)** is automatically placed at the end of the string.

For example, suppose the string "**graphics**" is to be stored in a one-dimensional character array called **var**. The definition would be:

```
        char var[10] = "graphics";
```

Since "**graphics**" contains 8 characters, **var** will be a **9-element array**. Thus, **var[0]** will represent the letter **g, var[1]** will represent the letter **r** and so on as represented in **table**.

Note that the last, that is, the 9th array element, **var[8]** will represent the **NULL** character **('\0')**, which signifies the end of the string.

The storage structure for the string "graphics" has been illustrated in table 4.1.

Element Number	Subscript Value	Array Element	Corresponding Data Item
1	0	var[0]	g
2	1	var[1]	r
3	2	var[2]	a
4	3	var[3]	p
5	4	var[4]	h
6	5	var[5]	i
7	6	var[6]	c
8	7	var[7]	s
9	8	var[8]	/0

Table 4.1: Storage Structure For String Graphics

4.17 Using Arrays for Input

The standard input stream **cin** is used to receive input from the standard input device, that is, the keyboard. Consider a very simple case where we need to accept a name and display a welcome name. The program required for the purpose is given in Example 27.

Example 27:

```
# include <iostream.h>
void main(void)
{
 char name[10];
 cout << "\nEnter your first name: " ;
 cin >> name ;
 cout << endl ;
 cout << "\nHello " << name << " Welcome to Arrays!" ;
}
```

Output:

```
Enter your first name: SARAH
Hello SARAH Welcome to Arrays!
```

The memory representation of the input "SARAH", is shown:

S	A	R	A	H	/0
0	1	2	3	4	5

The first letter **S** will be stored in **name[0]**, the second letter **A** in **name[1]** and so on. Note that the **NULL** character is appended at the end, that is, in **name[5]** automatically, denoting the end of the string, **name**. Alternatively, C++ also provides a set of standard string manipulation functions. The use of the **gets()** function to receive input of a character array has been discussed in the following section.

4.18 Using gets with Arrays

We can use **gets()** to read a string. The name of the array representing the string is used as the argument of this function. The array name is used without specifying any index value. On its return from **gets()**, the array will hold the string input from the keyboard. The **gets()** function will continue to read characters until a carriage return is encountered. The use of **gets()** with arrays has been illustrated in Example 28.

Example 28:

```
#include <iostream.h>
#include <stdio.h>
void main(void)
{
 char str_char[100] ;

 cout << "Enter a string\n" ;
 gets(str_char) ; // Receiving input from the keyboard
 cout << "You entered " << str_char ; // Output to screen
}
```

Output:

```
Enter a string
Hello world !
You entered Hello world !
```

This program accepts a string from the user into the character array **str_char** using the **gets()** function and then displays the same on the screen using a simple **cout** statement. Hence, **gets()** is an alternative available to accept array inputs from the keyboard apart from **cin**.

4.19 Multidimensional Arrays

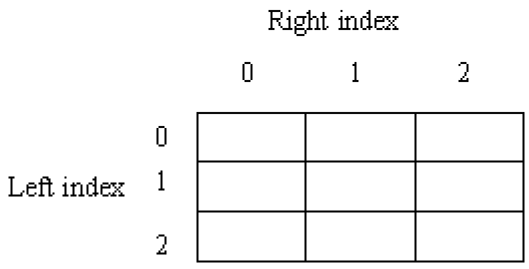
Multidimensional arrays are defined in the same manner as one-dimensional arrays, except that a separate pair of square brackets is required for each subscript. Thus, a two dimensional array will require two pairs of square brackets, a three dimensional array will require three pairs of square brackets and so on.

As the number of dimensions of an array increases, the complexity associated with referring to the data elements of that array also increases.

To create a 2-dimensional array **arr_1**, the following code would be required:

```
    int arr_1[3][3] ;
```

C++ would then generate the following matrix structure:



where the horizontal rows are indicated by the first subscript and the vertical columns are indicated by the second subscript.

Consider the following example:

Example 29:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
int i, j, matrix[3][4] ; // Matrix is defined as a 2-D array of
 // 3 rows and 4 columns

 clrscr() ;
 for(i=0; i<3; i++)
 {
 cout << "Enter numbers for row " << (i+1) << endl ;
 for(j=0; j<4; j++)
 {
       cin >> matrix[i][j] ;
 }
 }
cout << "You entered... " ;
 for(i=0; i<3; i++)
 {
 cout << "\nRow " << i+1 << "\t" ;

 for(j=0; j<4; j++)
 {
       cout << matrix[i][j] << "\t" ;
 }
 }
 getch() ;
}
```

Output:

```
Enter numbers for row 1
1
2
3
4
Enter numbers for row 2
5
6
7
8
Enter numbers for row 3
```

```
9

10

11

12

You entered...

Row 1 1 2 3 4

Row 2 5 6 7 8

Row 3 9 10 11 12
```

So far we have discussed 2-dimensional arrays from the perspective of using integer arrays. Matrix multiplication can be implemented using 2-dimensional arrays. Characters arrays can also be defined in a 2-dimensional format with slight variations. This has been discussed in the following section.

4.19.1 2-Dimensional Character Arrays

Suppose you want to create an array that contains a list of names. Obviously, this list will be a list of character strings. Example 30 shows how to initialize a 2-dimensional character array.

Example 30:

```
/* This program prints the five ocean names which are stored as 5
 strings, in a 2-D array. */
#include <iostream.h>
#include <conio.h>
void main(void)
{
 int i;
// Declaring a character array of 5 elements each of length 10
 char oceans[5][10] = { "Pacific", "Atlantic",
                  "Indian",
                  "Arctic",
                  "Antarctic"
             };
clrscr();
 cout << "\n\nPrinting the five oceans ......\n" ;
 for (i=0; i<5; i++)
 {
 cout << "\n\tOCEAN NUMBER" << i+1 << " is " << oceans[i] ;
 cout << endl ;
 }
 getch();
}
```

Output:

```
Printing the five oceans ......
OCEAN NUMBER 1 is Pacific
OCEAN NUMBER 2 is Atlantic
OCEAN NUMBER 3 is Indian
OCEAN NUMBER 4 is Arctic
OCEAN NUMBER 5 is Antarctic
```

In this program, we have defined an array, **oceans[5][10]** so that it can hold **5** names of up to a maximum of **10** characters each. We then initialized the array oceans by assigning the values of the 5 oceans, that is, "Pacific", "Atlantic", "Indian" and so on. The program then prints the values in the array **oceans**.

If we want to print only the **Indian** ocean, then the actual subscript value needs to be specified, which is,

```
cout << oceans[2] ;
```

Note that the five strings have a subscript value ranging from **0 to 4**.

Each string is enclosed in double quotes. Doing this causes the compiler to generate the code for character string data and place it in the proper array elements. As the compiler places a null termination character at the end of each name during initialization, the elements in the array can be used as character strings.

The values used to initialize the array comprise what is called an initialization list. This list contains the values the array will hold immediately after it has been defined. An opening and closing brace must appear before the first item and after the last item in the list.

The 2-D array in Example 30 can also be defined as

```
char oceans[][10] = { "Pacific",
            "Atlantic",
                 "Indian",
                 "Arctic",
                    "Antarctic"
              } ;
```

The first square brackets, as discussed earlier, denote the number of strings to hold, and the second square brackets denote the maximum width of each of these strings. If the number of strings that will be initialized is not mentioned (first bracket), the compiler automatically allots as much space in the memory as required by the initialization made.

Suppose we want to store the scores of all the batsmen in a cricket team for a given match. The scores must be stored along with the batsman number. The program thus required is illustrated in Example 31.

Example 31:

```
#include <iostream.h>
#include <conio.h>

void main(void)
{
 int b_count, b_scores[11][2] ;
 clrscr() ;
 for(b_count = 0; b_count < 11; b_count++)
 {
 cout << "Enter score for player " << b_count+1 << ": " ;
 cin >> b_scores[b_count][1] ;
 b_scores[b_count][0] = b_count ;
 cout << endl ;
 }
for(b_count = 0; b_count < 11; b_count++)
 {
 cout << "Score for player " << b_count+1 << ": " << b_scores[b_count][1] ;
 cout << endl ;
 }
}
```

Output:

```
Enter score for player 1: 90
Enter score for player 2: 67
Enter score for player 3: 54
Enter score for player 4: 2
Enter score for player 5: 0
Enter score for player 6: 6
Enter score for player 7: 7
Enter score for player 8: 9
Enter score for player 9: 10
Enter score for player 10: 1
Enter score for player 11: 90
Score for player 1: 90
Score for player 2: 67
Score for player 3: 54
Score for player 4: 2
Score for player 5: 0
Score for player 6: 6
Score for player 7: 7
Score for player 8: 9
Score for player 9: 10
Score for player 10: 1
Score for player 11: 90
```

4.20 Pointers and Arrays

There is a close connection between pointers and array names. Indeed, there are many situation where you can use an array name as though it were a pointer.

4.20.1 Array of Pointers

We can create an array of pointers. Each of the pointers in this array will point to a similar data type. Consider the code fragment given. It defines an array of pointers.

```
int *pt[7];
```

The code fragment defines an array pt, which consists of seven pointers. All these pointers can be directed to point to integer variables. For example if we want the sixth pointer in the array to point to an integer variable var the following code is to be written.

```
pt[5]=&var;
```

A note should be taken that the array elements will hold the addresses of integer variables.

Like other arrays, an array of pointers can be initialized. We will see this in the example given.

Example 32:

```
#include <iostream.h>
#include <conio.h>
void main(void)
{
char *song[]= { " Humpty dumpty sat on a wall\n",
                " Humpty dumpty had a great fall\n",
                " Not all the kings horses and men\n"
 };
cout<<song[2];
}
```

Output:

Not all the kings horses and men

In the program, we define an array of character pointers. Each pointer in the array song points to a string. The first pointer points to the string " Humpty dumpty sat on a wall". Therefore, we see that the output of the cout statement is " Not all the kings horses and men" because it points the address of the third string.

4.21 Check Your Progress

Fill in the blanks

1. __________________are important in establishing a relationship between the various functions being used in a program.
2. __________________is the process of introducing a function to a program.
3. The default return data type of a function is__________________.
4. All variables are classified into__________________and__________________depending upon their scope.
5. The sizeof operator is a__________________compile time operator.
6. Every function after performing a set of specified instructions must return the control back to its calling function. (True / False)
7. The ______________ is a C++ keyword, which indicates that the corresponding function does not return any value.
8. Data passed between functions is referred to as __________________or __________________
9. Variables declared within functions are called ______ variables.
10. Variables, which are not declared within any specific function, are called _______ variables.
11. The dummy or formal parameter is the name of the corresponding parameter in the called function. (True / False)
12. A default argument is a value, which is automatically assigned to a formal variable, if the actual argument from the function call is omitted. (True / False)
13. To ascertain the storage space required for a variable, we need to use the ___________ operator.
14. ____________ is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.
15. A ______ is a last-in, first-out (LIFO) data structure.
16. Automatic variables cannot retain their values over a number of function calls to that function. (True / False)
17. Static variables retain their values over a number of function calls. (True / False)
18. External variables are declared with the keyword ____________.
19. Programs cannot be composed of multiple files. (True / False)
20. The ______operator is used to return the memory storage address of the associated variable.
21. A pointer variable is defined using the______ operator.
22. In call by value, a variable is passed as an argument to a function, which in turn creates another

copy of it in the memory.(True / False)

23. In call by reference, the parameter passed to the called function is the __________ of the variable data rather than the value of the variable itself.

Fill in the blanks

1. Pointers are used to refer to values using their memory storage addresses.
2. The statement *ptr is used to refer to the value stored in the pointer variable ptr.
3. A separate name has to be used for each element while storing data in an array.
4. Arrays can store only integer type of data.
5. Subscripts specifying the storage model for arrays begin from 0 rather than 1.
6. All types of arrays are terminated with the NULL character.
7. The gets() function is used as an alternative to cin for accepting string data from the user.

4.22 Do It Yourself

1. Write a program, which should accept the user details like name, age, gender, and salary. The program should display the same data after the user finishes entering them. Use functions to accept and display each data.

2. Write a menu-based program, which should take two numbers as the input. Depending on the users choice the program should add, subtract, multiply, or divide the numbers. In case of division, the program should display appropriate error message if the second number input is zero. Use functions to achieve the mathematical operations.

3. Write a program that reverses any number given by the user. Use recursive function to extracts digits.

4. Write a program, which accepts two numbers as the values for two variables. Using pointers swap the values of these variables and display the result.

5. Write a program to accept a number from the user. The program should print the string “Learning C++ Is Fun” on the screen as many times as the user desires.

6. Modify the program in session 7, question 2, to give a discount of 10% if the number of passengers is greater than ten. Also, it should allow multiple bookings and the program should terminate only when the user desires to do so. For this, an option to quit should be given after each booking.

7. Write a program to accept a set of values and calculate the sum of values entered. Maintain a running count displayed on the screen, which should indicate the sum of the values entered so far. The program should stop accepting any further values if zero (0) is entered.

8. Write a program to accept the scores of all the players in a cricket team. There are in all 11 players in a cricket team. Maintain a running total, which displays the total score of the team depending on the score of each player.

9. Write a menu-based system to list the following menu.

    1. Draw rectangle.

    2. Draw square.

    3. Draw pyramid.

    0. Exit

    Print the respective figure after accepting the values required for the dimensions. Print the figures using the asterisk notation.

Summary

➔ Functions allow us to break up large tasks into smaller and easily manageable ones.

➔ The structure of a function has the following distinctive characteristics:

a. A name that identifies it

b. The type of the value returned by the function

c. The function's parameters and their data types

d. Processing statements.

➔ The function from within which a call is made to another function is known as the calling function.

➔ A function can be made to return values to its calling function by using the return statement.

➔ Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied.

➔ Automatic variables cannot retain their values over a number of function calls to that function.

➔ Static variables retain their values over a number of function calls.

➔ External variables are declared with the keyword extern.

➔ The register storage class can be specified only for local variables.

➔ Pointers are an alternative to using variables while accessing data stored in memory.

➔ An array is a list of variables that are all of the same type and are referenced through a common name. An individual variable in the array is called an array element.

➔ A two dimensional array is represented by a matrix of rows and columns.

Onlinevarsity
your e-way to learning

Visit

Frequently Asked Questions

@

www.onlinevarsity.com

Session - 5

Function Overloading

Welcome to the Session, **Function Overloading**.

This session introduces friend functions, function overloading, operator overloading and copy constructors.

At the end of this session, you will be able to:

- ➔ Explain the concept of functions with default arguments
- ➔ Define and use Friend functions with friend classes
- ➔ Describe function overloading
- ➔ Describe Operator Overloading
- ➔ Describe overloading of the Assignment Operator
- ➔ Describe Copy Constructors
- ➔ Describe conversion functions which help in conversion
  - from Basic types to User-Defined types
  - from User-Defined types to Basic types
  - between Objects of different Classes
- ➔ Identify operators that cannot be overloaded

5.1 Introduction

In a program, to perform a specific set of actions, functions are invoked. When we define a function we are actually specifying how an operation is to be done.

In this session, we will discuss how the use of functions has been enhanced in Object Oriented programming.

5.2 Functions with Default Arguments

In C++, it is possible to call a function without specifying all its arguments. Of course, this will not work on just any function. The function declaration must provide default values for those arguments that are not specified.

In the function declaration, the default values are given. Whenever a call is made to a function without specifying an argument, the program will automatically assign values to the parameters from the default declaration. Let us look at an example. Consider the function declaration

```
void func(int = 1, int = 3, char = '*'); //prototype declaration
```

The default argument follows an equal sign that is placed directly after the type name. You can also use variable names, as shown:

```
void func(int num1,int num2 = 3,char ch = '*');
```

In the function declaration, once an argument is given a default value in the list of formal arguments, all the remaining must have default values also. It is not possible to leave out any of the values in the middle of the list.

```
void func(int num1=2,int num2, char ch='+');     //error
```

When the function is called, missing arguments are assumed to be the last arguments. The following calls to the function func declared, are all considered valid except for the last one.

```
func(2,13,'+');
func(1);          //default values for second and third arguments
func(2,25);       //default value for third argument
func();           // default values for all three arguments
func(2,,'+');     //invalid
```

The missing arguments for the function call must be those at the end of the argument list. If you leave out any arguments in the middle the compiler would not know what you are referring to and will indicate an error.

The default values can be given in either the prototype or the function definition header, but not in both. As a matter of style, it is highly recommended that the default values be given in the prototype declaration rather than in the function definition.

➔ **Advantages**

- Default arguments are useful if you want to use arguments, which will have the same value in a function.
- They are also useful when the programmer decides to increase the capability of a function by adding an argument. In such a case, the existing function calls can continue to use the old number of arguments, while new function calls can use more.

5.3 Friend Functions

The basic object-oriented concepts of data hiding and encapsulation are implemented by restricting non-member functions from accessing an object's private data. The basic policy is, if you are not a member then you cannot get in. Private data values cannot be read or written to by non-member functions. Sometimes this rigid rule for isolating data from function can be inconvenient. Imagine that you want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments, and operate on their private data. Of course this is not possible since a function cannot be a member of two classes. What we need is a means to allow a function access to the private members of a class without requiring membership. A non-member function that is allowed access to the private variable of a class is called a **friend** of the class. Figure 5.1 depicts this.

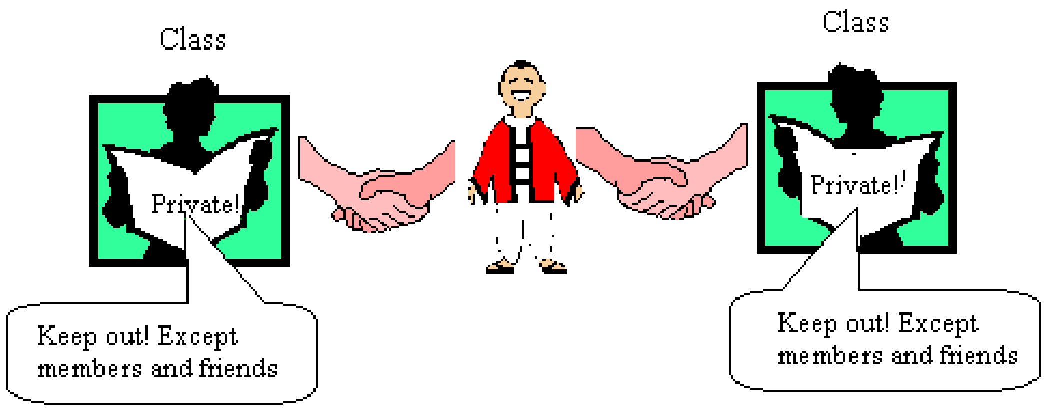

Class
Private!
Keep out! Except members and friends
Class
Private!
Keep out! Except members and friends

Figure 5.1: Friend Function

A function is made a friend of a class by a friend declaration in that class. For example:

Example 1:

```
class person{
      .
      .
      .
      public:
            void getdata();
            friend void display(person abc);
};
void display(person abc) //friend function without :: operator
{//… some code…}
```

Note: The keyword friend is not repeated in the function definition.

The friend function in the example could perhaps have done the same operation if it were a member function of the class. However, if the same function needed to access objects from different classes it would be most useful to make it a friend of the different classes. Let us look at an example.

Example 2:

```
class Teacher;                          //forward declaration
class Student{
     private:
     int st_data;
     public:
     void getstuddata();
 friend void display(Student abc, Teacher xyz);
 };
class Teacher{
     private:
 int th_data;
     public:
     void getteachdata();
 friend void display(Student abc, Teacher xyz);
 };
 void display(Student abc, Teacher xyz)
{//… some code…}
```

In the example a friend function `display()` accesses members of two classes. The friend function is declared in the class specifier of both the classes although it does not belong to either. To access the private members of the classes the name of the data member has to be prefixed with the name of the object along with the dot operator. For example, in the definition of the friend function given, to access the data member of the object `abc` the syntax would be:

```
abc.st_data
```

In Example 2, the class `Teacher` has been declared before the class `Student`. This is because a class cannot be referred to until it has been declared. The class `Teacher` is referred to in the declaration of the friend function in class `Student`. Therefore, it has been declared at the beginning of the program. This is called a forward declaration.

There are some features of a friend function that are worth remembering.

➔ A friend function can access the private members of a class.

➔ Friend functions need not have a '`this`' pointer, as it does not belong to any object.

➔ The friend declaration is unaffected by its location in the class.

➔ The definition of a friend function does not require the class name with the scope resolution operator prefixed to it, as it is required for a class member function.

Only when a function accesses private members of two or more classes directly, it has to be declared as a friend function. Otherwise public members of a class can be accessed directly by any function. There is some controversy regarding the use of friend functions.

On one hand, friend functions increase flexibility in programming, on the other they are against the principles of object-oriented programming.

A friend function has to be declared in the class whose data it will access. This cannot be done if the source code is not available to a programmer. If the source code is available, then existing classes should not be modified as far as possible. If you intend to use friend functions the class should be designed as such, right from the beginning. Even then, friend functions are not a tidy concept to work with because it involves a lot of intermingling with different classes. Sometimes they are unavoidable, but excessive use of friend functions over many classes also suggests a poorly designed program structure.

➔ **Advantages:**

Friend functions provide a degree of freedom in the interface design options. Member functions and `friend` functions are equally privileged. The major difference is that a friend `function` is called like `func(xobject)`, while a member function is invoked using `xobject.func()`. The ability to choose between member functions (`xobject.func()`) and `friend` functions (`func(xobject)`) allows a designer to select the syntax that is considered most readable, which lowers maintenance costs.

Advantages of friend functions are as follows:

- When two or more classes contain members that are interrelated with the other parts of your program then friend functions prove to be useful.
- Friend functions may be useful in operator overloading which we will discuss later.
- Friend functions may facilitate the creation of some type of I/O functions.

5.3.1 Friend Classes

We can declare a single member function or a few member functions or a whole class as a friend of another class. Now, you may wonder when we would want to declare a whole class as a friend. When all or most of the functions of a particular class have to gain access to your class, you can consider allowing the whole class the friend privileges. Instead of declaring each of the member functions of the class as friends, you can save time by declaring the whole class as a friend.

Let us look at an example where the member function of one class is a friend of another.

Example 3:

```
#include <iostream.h>
class beta;                          //forward declaration
class alpha{
 private:
      int a_data;
 public:
      alpha(){a_data = 10;}
void display(beta);
};
class beta{
 private:
int b_data;
public:
beta(){b_data = 20;}
friend void alpha::display(beta bb);
};
void alpha::display(beta bb)
{
      cout<<"\n data of beta ="<<bb.b_data;
      cout<<"\n data of alpha ="<<a_data;
}
void main()
{
      alpha a1;
      beta b1;
a1.display(b1);
}
```

In this example the function `display()` which is a member function of class `alpha` is made a friend function of class `beta`. In the class specifier of class `beta` the function is declared as a friend. The class name of `alpha` is given with the scope resolution operator to identify the function. The friend function can access the private data member of class beta with the dot access operator.

The member functions of a class can all be made friends at the same time when you make the entire class a friend. Consider Example 4.

Example 4:

```
#include<iostream.h>
class beta;
class alpha{
 private:
      int data;
 public:
      friend class beta;          //beta is a friend class
};
class beta{
      .
      .
 public:
      void display(alpha d)       //Access alpha.data
{cout<<d.data;} //
void get_data(alpha d)
{ int x = d.data;}
};
```

By using the declaration,

`friend class beta;`

we make the class `beta's` entire member functions friends of `alpha`. Now all the member functions of class `beta` can access the private data members of `alpha`. However, the member functions of the class `alpha` cannot access the private members of the class `beta`. It is also worth noting that friends of class `beta` are not automatically made friends of the class `alpha`. However, more than one class can be made a friend of class `alpha`.

How are the classes instantiated when they are declared as friend class? In which order are they instantiated?

5.4 Function Overloading

Other than classes and objects in object-oriented programming, overloading of functions is an important feature. Function overloading is used to define a set of functions that are given the same name and perform the same operations, but use different argument lists.

Function overloading can be considered as a type of polymorphism called functional polymorphism.

Polymorphism is essentially one thing having many forms.

Therefore, functional polymorphism means one function having several forms.

It may seem mysterious how an overloaded function knows what to do. It performs one operation on one kind of data but another operation on a different kind.

For example,

```
void display(); // Display functions

void display(const char*);

void display(int one, int two);

void display(float number);
```

➔ Advantages

The main advantages of using function overloading are:

- Eliminates the use of different function names for the same operation
- Helps to understand and debug code easily
- Maintaining code is easier

Function overloading is an exciting feature but it should not be overused. Only those functions that basically do the same task, on different sets of data, should be overloaded. In function overloading, more than one function has to be actually defined and each of these occupy memory. In some cases, instead of function overloading, using default arguments may make more sense and create fewer overheads.

```
int square(int);

float square(float);

double square(double);
```

You will notice that one method uses a single integer and another uses a single float type variable, but the system is able to select the correct one. You can use as many overloadings as desired provided all of the parameter patterns are unique.

5.4.1 Overloading with Different Number of Arguments

In addition to being overloaded for different data types, functions can also be overloaded for the number of arguments in the function call. Consider the following example:

```
int square(int);            //function declarations

int square(int,int,int);

int asq = square(a)         //function calls

int bsq = square(x,y,z)
```

When a function, `square`, is called, the compiler compares the types of the actual arguments with the types of the formal arguments of all functions called `square`. The idea is to invoke the function that is the best match on the arguments or else the compiler gives an error if no function produces the best match.

Note that the way the compiler resolves the overloading is independent of the order in which the functions are declared. In addition, the return types of the functions are not considered.

We have used constructors, which can be given different number of arguments or with different data types. This is an example of function overloading.

5.4.2 Scope Rules for Function Overloading

We know now that function overloading is the process of defining two or more functions with the same name, which differ only by the type of arguments and number of arguments. The overloading mechanism is acceptable only within the same scope of the function declaration. Sometimes, one can declare the same function name for different scopes of the classes or with global and local declaration, but it does not come under the technique of function overloading.

Example 5 shows how a function cannot be considered overloaded when it is declared for a different scope.

Example 5:

```
class first{
      .
.
      public:
            void display();
};
class second{
      .
.
```

```
      public:
            void display();
};
void main()
{
first object1;
      second object2;
      object1.display();    //no function overloading takes place
object2.display();
}
```

The same function `display()` is available in both functions. The scope is strictly confined to the classes in which they are declared.

5.5 Operator Overloading

Operator overloading is one of the interesting and useful features of object-oriented programming. In traditional programming languages, expressions involving operators like `+`, `-`, `>`, `+=`, `==`, etc. can be used only on basic data types like `int` and `float` and not on derived or user-defined data types like objects. For example,

```
x = y + z;
```

or

```
if(x>y) {. . .}
```

works only with basic types like `int` or `float`. If you try to perform the same operation on objects of a class the compiler would give an error. However, the concept of operator overloading allows statements like,

```
if (obj1>obj2){ . . .}
```

where `obj1` and `obj2` are objects of a class. The operation of comparing the objects can be defined in a member function and associated with the comparison operator.

Definition

The ability to associate an existing operator with a member operator function and use it with objects of its class as its operands is called overloading.

Essentially, operator overloading gives you the ability to redefine the language by allowing you to change the way operators work. We have seen with overloaded functions that the compiler knows which function to use based on the data type of the arguments. In the same way, the compiler can distinguish between overloaded operators by examining the data type of its operators.

Operator overloading is one form of polymorphism. Polymorphism allows the creation of multiple definitions for operators and functions. Operator overloading can be termed as operational polymorphism.

In C++, programs can overload existing operators with some other operations. If the operator is not used in the context as defined by the language then the overloaded operation, if defined, will be carried out. For example, in the statement

```
x = y-z;
```

If, `x`, `y` and z are integer variables, then the compiler knows the operation to be performed. If `x`, `y` and `z` are objects of a particular class, then the compiler will carry out the instructions that have been associated with the '-' operator.

Since the operator is being overloaded it is also possible to write instructions that would make the '-' operator function perform operations other than subtraction. However, that would not be contrary to the original specification for which the operator was intended.

There are certain points about overloading that are worth noting:

➔ Overloading cannot alter the basic function of an operator, nor change its place in the order of precedence that is already defined in the language. For example, ++ (increment) and --(decrement) can be used only as unary operators.

➔ Overloading an operator should never change its natural meaning. For example, an overloaded + operator can be used to multiply two objects but this would make your code unreadable.

➔ Only operators that already exist in the language can be overloaded. You cannot use a new symbol. Barring a few exceptions, all the operators in C++ can be overloaded. The list of operators that cannot be overloaded is shown at the end of the session.

➔ Overloading of operators is only available for classes. You cannot redefine the operators for the predefined basic types.

The main advantage of using operator overloading is that it makes programs easier to read and debug. It is easier to understand that two objects are being added and the result assigned to a third object, if you use the syntax

```
obj3 = obj1 + obj2;
```

instead of the following statement,

```
obj3.addobjects(obj1,obj2);
```

➔ The operator keyword

The actual instructions to overload an operator are written in a special member function defined with the keyword `operator`. The operator that has to be overloaded follows the keyword. Such a member function is called an *operator function*.

The general format of this function is

```
return_type operator op(argument list);
```

where `op` is the symbol for the operator that is being overloaded. For example in a class `Sample` the declaration,

```
void operator --();
```

tells the compiler to call this member function whenever the decrement operator is found in the program, provided the variable that is operated on by `--` is of the type `Sample`. For example,

```
Sample s1;

s1--;
```

calls the operator -- member function of the class Sample.

5.5.1 Overloading Unary Operators

Unary operators have only one operand. Examples of unary operators are the increment operator ++, the decrement operator --, and the unary minus operator.

The increment and decrement operators can be used as either prefix or postfix operations. Look at Example 6 that illustrates overloading the operator ++.

Example 6:

```
 #include<iostream.h>
class Rectangle
{
 private:
      int width;
      int length;
 public:
      Rectangle()
      {
      width=5;
length=10;
}
      void Area()
      {
       cout<< "The area = "<<width*length;
 }
      void operator++()
      {
width++;
length++;
}
};
void main()
{
 Rectangle obj1;
 obj1.Area();
 obj1++;   // increases height and width by one
 obj1.Area();
}
```

The output for the example 6 is:

The area = 50 The area = 66

In `main()` the increment operator is applied to a specific object. The member function, operator++(), does not take any arguments. It increments the value of all the data members, `length and width`, of the object. A similar function to decrement the object can also be included in the class as:

```
void operator --()
{
width--;
      length--;
}
```

This can be invoked with the statement

```
--obj1;
```

or

```
obj--;
```

In the example, the compiler checks to see if the operator is overloaded. If an operator function is found in the class specifier, the statement to increment the object

```
obj1++;
```

gets converted, by the compiler to the following:

```
obj1.operator++();
```

This is just like a normal function call qualified by the object's name. The compiler treats it like any other member function of the class.

However, the operator function in the example has a problem. On overloading, it does not work exactly as it does for basic data types. With the overloaded increment and decrement operators, the operator function is executed first, regardless of whether the operator is postfix or prefix.

Whether the operator is prefix or postfix will not matter in statements like `obj1++ or ++obj1` but, there is a problem when the following statement is used:

```
obj1 = obj2++;
```

If `obj1` and `obj2` were basic data types the compiler would not give any error. However, it will give an error if `obj1` and `obj2` are objects. The ++ operator function in the class is declared with a `void` return type, whereas we want the operator function to return a variable of type `Sample` so that it can be assigned to the object on the left-hand side. We will have to revise the function so that it is able to return an incremented object. To do that we can use a temporary object in the operator function as shown:

Example 7

```
. . .
Rectangle Rectangle:: void operator++()
{
 Rectangle temp;            //create a temporary object
 temp.width = ++width;      //assign incremented value
 temp.length= ++length;     //assign incremented value
 return temp;               //return incremented object
}
. . .
```

In this example, the operator function creates an object `temp` of class `Sample`, assigns the incremented value of `counter` to the data member of `temp` and returns the new object. Now the object that is returned from the function can be assigned to another object in main. This is just one way of returning the object. Another way is to create a nameless temporary object and return it.

Example 8:

```
class Rectangle{
 private:
      . . . // data members width and length
 public:
      Rectangle()                            //constructor with no argument
      {
         . . . // default values
       }
      Rectangle(int w, int l)
      {
      width = w;
      length = l;
      }
};
Rectangle Rectangle::void operator++()
{
++width;
++length;
return Rectangle(width,length);
}
```

One change in the class definition is a constructor with one argument. No new temporary object is explicitly created in the operator function. However, in the return statement, an unnamed temporary object of class `Rectangle` is created by the constructor that takes two arguments.

Yet another way of returning an object from the member function is by using the `this` pointer. In an earlier session we have seen that the `this` pointer is a special pointer that points to the object that invoked the member function. This is how we can use the `this` pointer to return an object.

Example 9:

```
. . .
Rectangle Rectangle::void operator++()
{
++width; ++length;
return(*this);
}
. . .
```

The one argument constructor is not needed in this approach.

When `++` and `--` are overloaded, there is no distinction between the prefix and postfix operation. The expressions,

```
obj2 = obj1++;
```

and

```
obj2 = ++obj1;
```

produce the same effect. In both the cases `obj1` is incremented before it is assigned to `obj2`. This is different for basic data types where the first expression would cause assignment before incrementing. In this aspect the use of the `++` operator and `--` operator with objects is not totally equivalent to its usage with the basic data types.

5.5.2 Overloading Binary Operators

Binary operators are operators with two operands. They can also be overloaded just like unary operators. We will look at three different categories of binary operators

- Arithmetic operators
- Compound assignment operators
- Comparison operators

Binary operators can be overloaded in two ways:

- as member functions they take one formal argument, which is the value to the right of the operator. For example, when the addition `obj1+obj2` has to be done the overloaded operator function for + is declared as,

  ```
  operator(operator symbol)(Parameter)
  example: operator+(Rectangle obj2)
  ```

- as friend functions they take two arguments. For example,

  ```
  operator(operator symbol)(Parameter1, Parameter2)
  example: operator+(Rectangle obj1,Rectangle obj2)
  ```

a. Binary Arithmetic Operators

Arithmetic operators are binary operators and therefore, require two operands to perform the operation.

Using the `Sample` class from Example 10 we can define an overloaded function for the operator + as shown in Example 10.

Example 10:

```
Rectangle Rectangle::operator+(Rectangle a)
{
 int counter = 10;
 Rectangle temp;                      //temporary object
 temp.counter = counter + a.counter;  //addition
 return temp;                         //return temp object
}
```

Now we are able to perform addition of objects with a statement,

```
obj3 = obj1 + obj2;                         //objects of class Sample
```

The operator + can access two objects. In the statement, the object on the left side of the operator, `obj1`, is the one that will invoke the function and the object on the right hand side, `obj2`, is taken as the argument to the function call. In the operator function, the left operand (object `obj1`) is accessed directly since this is the object invoking the function. The right hand operand is accessed as the function's argument as `a.counter`.

Using the overloaded + operator it is also possible to perform multiple additions such as

```
obj4 = obj3 + obj2 + obj1;
```

where all the variables are objects of the class. This kind of multiple addition is possible only because the return type of the + operator function is an object of type `Sample`.

Languages like BASIC have an in-built facility to concatenate two character strings using the + operator. This facility is not available in C++ but we can overload a + operator to achieve the same

effect.

For a class `String` that contains a data member, `str`, which is a character array of, say, 100 characters, we can define an operator function as shown in Example 11.

Example 11:

```
. . .
String String::operator+(String ss)
{
 String temp;                   //make a temporary string

 strcpy(temp.str,str);          //copy this string to temp
 strcat(temp.str,ss.str);       //add the argument string
 return temp;
}
. . .
```

The operator function concatenates the strings of the two objects using the standard string handling functions into the string of the temporary object and returns it. You can use the function to add two strings as shown in the statements:

```
String s1 = "Welcome";
String s2 = "to C++";
String s3;

s3 = s1 + s2;
```

The statement would produce the string `Welcome to C++`.

Similarly other binary arithmetic operators can also be overloaded so that you can subtract, multiply, and divide objects of the class.

b. Compound Assignment Operators

Compound assignment operators can also be overloaded like binary arithmetic operators. Operators such as the += operator combine assignment and addition in one step. Let us look at Example 12 that illustrates an overloaded operator function using the same class `Sample`.

Example 12:

```
. . .
void Rectangle::operator+=(Sample a)
{
 width += a.width;    //addition
 length += a.length;  //addition
}
. . .
```

In this example there is no need for a temporary object. The function also needs no return value because the result of the assignment operator is not assigned to anything. The operator is used in expressions like

```
obj1 += obj2;
```

If you want to use this operator in more complex expressions such as

```
obj3 = obj1 += obj2;
```

then the function would need to have a return value. Then the member function declaration would be:

```
Sample Sample::operator+=(Sample a);
```

The return statement can be written as:

```
return Sample();
```

where a nameless object is initialized to the same values as this object. Of course, you will have to define a constructor that takes one argument so as to be able to create the nameless object.

c. Comparison Operators

Comparison and logical operators are binary operators that need two objects to be compared. The comparison operators that can be overloaded include <, <=, >, >=, ==, and !=. If we use the following comparison with an overloaded > operator,

```
if (s2>s2)
```

the overloaded operator accepts the object `s2`, on its right, as the argument to the function and the object `s1`, to its left, as the invoking object. Let us look at a typical example, Example 13, using the class `String` that will help us to overload a comparison operator.

Example 13:

```
. . .
int string::operator>(String ss)
{
 return(strcmp(str,ss.str) > 0);
}
. . .
```

In the `return()` statement of Example 13,

➔ The first argument, **str**, is the data member of invoking object

➔ The second argement,**str**,is the data member of object ss that is passed as argument

The return value of the comparison is an integer. The > operator is used to indicate whether the first string comes before or after the second in list that is alphabetically ordered. The same function can be adapted for use with the other comparison operators. You could also define overloaded comparison operators to compare string lengths if that is what is needed in your program.

5.5.3 Overloading the Assignment Operator

We have probably used the assignment operator = many times without going into the details of how it operates with objects. When we use a statement such as,

```
obj2 = obj1;
```

the default assignment operator simply copies the source object to the destination object byte by byte. Let us consider what would happen in a class where the data members contain pointers and have been allocated memory using the `new` operator. Refer to Example 14.

Example 14:

```
#include<iostream.h>
#include<string.h>
class Oap{
 private:
      char *str;
public:
      Oap(char *s = "")                //constructor
      {
      int length = strlen(s);
      str = new char[length+1];
      strcpy(str,s);
      }
      ~ Oap(){delete str;}        //destructor
      void display(){cout<<str;}
};
void main()
{
 Oap s1="Welcome to my world \n";
 Oap s2;
      s2 = s1;
      s1.display();
      s2.display();
}
```

In Example 14, two objects `s1` and `s2` has been created. The constructor function allocates memory for a Oap and copies the contents of its formal argument in it. The assignment in main() assigns the object s1 to s2. The output of this program will be:

```
Welcome to my world

Welcome to my world

Null pointer assignment
```

We shall see why the program generates a null pointer assignment message.

After the assignment the data member str of both the objects points to the same location of memory. That is because; the assignment operator copies only the pointer and not the actual contents of the string. As a result there are two pointers to the same memory location. After the program is executed, the destructor function is called automatically. This releases the memory allocated by new to the data member str. The delete operator called for one-object releases the memory and for the second call it attempts to release the same memory location all over again. This results in the error message.

Figure 5.2 depicts assignment and deletion.

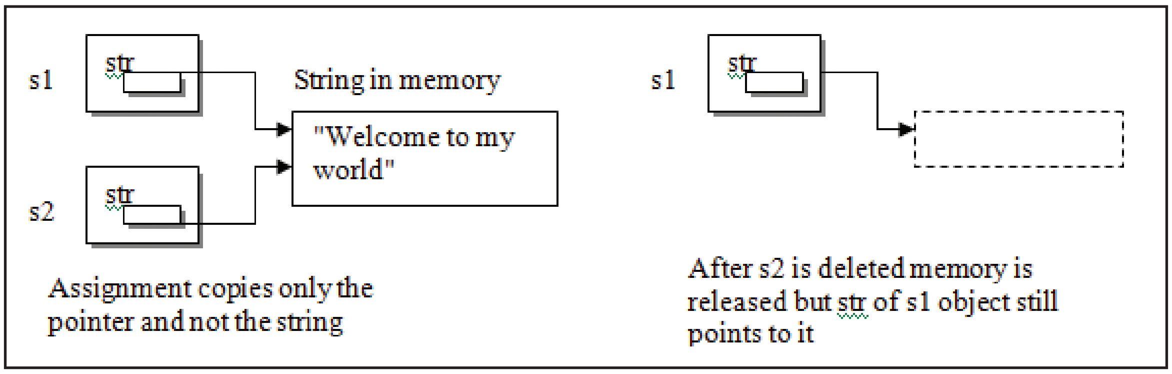

s1
str
String in memory
"Welcome to my world"
s2
str
Assignment copies only the pointer and not the string
s1
str
After s2 is deleted memory is released but str of s1 object still points to it

Figure 5.2: Assignment and Deletion

The solution to this is to define an operator function for the assignment operator, i.e., unary operator implemented as global operator functions have a single parameter that is the operand. As a non-member function, the operator function for the unary operator **Oper** could be declared as:

```
Class_Name& operator Oper(Class_Name&)
```

This can be implemented as given:

```
Oap& Oap::operator=(Oap& s)
{
      delete str;
      int length = strlen(s.str);
      str = new char[length+1];
      strcpy(str,s.str);
      return(*this);
}
```

By including this function in the Oap class of example14 the error message will not appear. When we use the assignment operator an existing object is being copied to another object. Since it is a member-by member-copy the pointer that is copied will be pointing to the old object. To avoid this, the previous Oap object in example 14 pointing to str is first deleted. A new space is allocated in memory for the string. The pointer str of the new object is made to point to this new space. The value of the old Oap object is copied into the new Oap object. Thus the problem of two pointers to the same location of memory is avoided.

Let us see an example of a similar function used for a data member that is an integer, as in class Sample used in Example 15.

Example 15:

```
. . .
Oap Oap::operator=(Oap& a)
{
 int counter;
      counter = a.counter;
      return Oap(counter);
}
. . .
```

When you overload the assignment operator you must make sure that all the data members are copied from one object to the other in the operator function. In the Oap class there is only one data member. Where more data members are involved each has to be copied to the target object.

Here the argument in the operator function is passed by reference. We have seen earlier that an argument passed by value creates a copy of itself in the function to which it is passed. This holds true for the operator=() function. If objects are large, each copy operation could take up a lot of memory.

The operator function returns the value by creating a temporary Oap object and initializing it using a constructor that takes one argument. The value returned is a copy of the object of the same class that the member function belongs to. Thus, it is possible to use a chain of = operators, such as:

```
obj3 = obj2 = obj1;
```

Returning by value has the same problems of memory as passing an argument by value. Returning by reference returns only the address of the variable being returned. For local variables the address points to data within the function.

5.6 Copy Constructors

The problems that we have seen while using the assignment operator can also occur whenever we try to initialize an object with another object of the same class. Although initialization and assignment both assign values, initialization occurs only once when an object is created, whereas assignment can occur whenever it is wanted in the program. Let us take a look at two statements:

```
Oap s1("This is a string.");

Oap s2(s1);
```

The first statement declares an object `s1` and passes a string as an argument to its constructor. The second statement declared another object `s2` and contains an object as its argument. Since we have still not defined a constructor with an object as its formal argument, the compiler itself initializes the data members of `s2` with those of `s1`. Since the data member of class Oap is a pointer, the null pointer assignment problem arises just as in the assignment operator. The problem is solved by defining a constructor function that takes an object as its argument. This constructor is called the *copy constructor*. The general format for the copy constructor is:

```
X::X(X &ptr)
```

where `X` is the user defined class name and `ptr` is an object of class `X` that is passed by reference.

The copy constructor may also be used in the following format using the `const` keyword:

```
X::X(const X &ptr)
```

By using the const keyword we make sure that the copy process does not inadvertently make any changes to the object that is being copied.

We have seen that an object is copied when one object is used to initialize another object. When an argument is passed to a function, a variable that has not been initialized so far, as the formal argument, is initialized. This invokes the copy constructor. Thus we can see that a copy constructor may be invoked when an object is defined. It is also invoked when arguments are passed by value to functions and when values are returned from functions.

The compiler generates the copy constructor automatically if we do not define one. It is used to copy the contents of an object to a new object during construction of that new object. For simple classes with no pointers, that is usually sufficient, but when, we have a pointer as a data member, a byte by byte copy would copy the pointer from one to the other and they would both be pointing to the same allocated member. A copy constructor for the `Oap` class is shown in Example 16.

Example 16:

```
. . .
Oap::Oap(Oap& ss)
{
      int size = strlen(ss.str);
str = new char[size+1];
      strcpy(str,ss.str);
}
```

This constructor contains an object as its argument. It allocates a new memory space for a Oap and `str` is made to point to this space. It then copies the contents of the Oap to this new location.

Assignment and initialization are different operations. When a class X has a data member of pointer type, the class should have a constructor, assignment operator function, copy constructor and destructor. A typical class has been illustrated in Example 17.

Example 17:

```
class X{
.
.
X(some_value);             //constructor
X(const X&);               //copy constructor
X& operator=(const X&);    //assignment
~X();                      //destructor
};
```

5.7 Conversion Functions

Conversion functions are member functions used to convert objects to or from basic data types and for conversions between objects of different classes. However the compiler knows nothing about converting user defined types such as objects. The program has to define the conversion functions. Let us look at the program in Example 18, which we will use for the different conversions.

Example 18:

```
#include<iostream.h>
class Converter
{
 private:
      int feet;
      float inches;
 public:
      Converter()                  //default constructor
{feet = 0; inches =0.0;}
      Converter(float metres)      //constructor with one arg
{
      float f;
      f= 3.28 * metres;
      feet = int(f);
 cout<<feet<<"\n";
      inches = 12 *(f-feet);
 cout<<inches;
}
};
void main()
{
Converter d1 = 1.55;        //uses second constructor
Converter d2;               //uses first constructor
d2 = 2.0;                   //uses second constructor
}
```

The class contains two data members, `feet` and `inches`. It stores a value, given in terms of metres, as feet and inches. The first constructor with no arguments initializes the data members to zero. The second constructor takes the argument `metres`, which is a `float` and converts it to feet and inches.

5.7.1 From Basic Types to User-Defined Types

In Example 18, the declaration of the object d1 uses the second constructor and assigns the value 1.55.

It shows the conversion of a float constant to an object of class Converter. The statement

```
d2 = 2.0;
```

also uses the constructor function for assignment of a float to object d2. The compiler first checks for an operator function for the assignment operator. If the assignment operator is not overloaded, then it uses the constructor to do the conversion. If the constructor also had not been defined the compiler would have given an error. We can see that the constructor function does not distinguish between initialization at the time of declaration and assignment later on.

5.7.2 From User-Defined Types to Basic Types

Conversion of one basic data type to another is automatically done by the compiler using its own built-in routines or with the use of typecasting. Since the compiler knows nothing about a user-defined type such as an object, it has to be explicitly instructed if an object has to converted to a basic data type.

These instructions are coded in a conversion function and defined as a member of that class. In Example 18 earlier, we will have to add a member function, as given, to convert an object of the class Converter to a float variable.

Example 19:

```
. . .
Oap::Oap(Oap& ss)
{
      int size = strlen(ss.str);
str = new char[size+1];
      strcpy(str,ss.str);
}
. . .
```

This conversion function can be used to convert an object of class Converter to a float variable implicitly using the statement,

```
m = d2;
```

or explicitly using statement,

```
m = float(d1);
```

This conversion function is nothing but overloading of the type cast operator. The conversion function contains the operator keyword and instated of an operator symbol it contains the data type. The compiler first checks for an operator function for the assignment operator and if that is not found it uses the conversion function. The conversion function must not define a return type nor should it have any arguments.

5.7.3 Conversion between Objects of Different Classes

Conversion of an object of one class to an object of another class can be done using the assignment operator, but since the compiler is not aware anything about user-defined types, conversion functions have to be specified. This function can be a member function of the source class (i.e. the right hand side of the assignment operator) or it can be a member function of the destination class (i.e., the left-hand side of the assignment operator). In the following statement,

```
objectA = objectB;
```

`objectA` is considered an object of the destination class and `objectB` is an object of the source class.

Conversion of objects of two different classes can be achieved either with a one-argument constructor or a conversion function. Conversion functions are typically defined in the source class and one-argument constructors are typically defined in the destination class.

Let us look at two classes that can store a given length. The first class `LFeet` stores the length in terms of feet and inches and has a constructor to receive lengths in terms of feet and inches. The second class `LMetres` stores the length in metres and has a constructor to receive the length in terms of metres. Let us look at the class definitions first and then look at the conversion functions needed. Example 20 illustrates this.

Example 20:

```
class LFeet
{
 private:
      int feet;
      float inches;
 public:
      LFeet(){feet = 0; inches =0.0;}   //Constructor 1
      LFeet(int ft,float in)        //Constructor 2
      {
      feet = ft;
      inches = in;
}
};
class LMetres
{
```

```
private:
      float metres;
 public:
      LMetres(){metres = 0.0;}                    //Constructor 1
      LMetres(float m)                            //Constructor 2
      {
      metres = m;
}

};
void main()
{
 LMetres dm1 = 1.0;
 LFeet df1;
 df1 = dm1;
}
```

In `main()`, to be able to use the statement,

```
df1 = dm1;
```

that is, to convert from one class to another we will have to define either a conversion function in the source class or a constructor function in the destination class. We will look at example of both.

➔ A Conversion Function in the Source Class

The conversion function in the source class to convert the length from the source class `LMetres` to the destination class `LFeet` would be as follows:

Example 21:

```
. . .
. . .
class Lfeet {
. . .
};
. . .
```

```
operator LFeet()                    //conversion function
{
 float ffeet, inc;
 int ifeet;
      ffeet = 3.28*metres;
ifeet = int(ffeet);
inc = 12*(ffeet - ifeet);
return LFeet(ifeet,inc);
}
. . .
```

This conversion function will be defined in the source class `LMetres`. The statement to convert one object to another,

`df1 = dm1;`

calls the conversion function implicitly. It could also have been called explicitly as,

`df1 = LFeet(dm1);`

This statement shows that the conversion function is nothing but a member function for the overloaded cast operator. The use of this overloaded cast operator is similar to the one in section 5.4.2, where we converted an object to float.

➔ A Constructor Function in the Destination Class

The constructor function in the destination class should be as follows:

Example 22:

```
Class Lfeet {
. . .
};
. . .
LFeet::LFeet(LMetres dm)            //constructor function
{
 float ffeet;
ffeet = 3.28*dm.GetMetres();
feet = int(ffeet);
inches = 12*(ffeet - feet);
}
. . .
```

This is defined in the class `LFeet`. The constructor function can also be used at the time of initialization of an object. In addition, we will also have to define a member function called `GetMetres()` in the source class `LMetres` as shown in Example 23.

Example 23:

```
. . .
float LMetres::GetMetres()
{
 return metres;
}
. . .
```

This function returns the data member metres of the invoking object. This function is required because the constructor is defined in the `LFeet` class and since `metres` is a private data member of the `LMetres` class, it cannot be accessed directly in the constructor function.

The use of a conversion function in the source class or a constructor function in the destination class is strictly a matter of choice. Table 5.1 summarizes the conversion approaches we have seen.

Type of Conversion	Function in Destination Class	Function in Source Class
Basic to Class	Constructor	N/A
Class to Basic	N/A	Conversion Function
Class to Class	Constructor	Conversion Function

Table 5.1: Table for Type Conversions

5.8 Operators That Cannot be Overloaded

Not all operators can be overloaded. Here is a list of operators that cannot be overloaded:

➔ The sizeof() operator

➔ The dot operator (.)

➔ The scope resolution operator (::)

➔ The conditional operator (?:)

➔ The pointer-to-member operator (.*)

5.9 Check Your Progress

1. A default argument in a function has a value that
    a. is a variable value
    b. is a constant value
    c. is supplied by the function
    d. increments each time the function is called

2. A friend function is used to avoid arguments between classes. (True / False)

3. The keyword friend appears in
    a. the main() program
    b. the private or public section of a class
    c. the class allowing access to another class
    d. the class desiring access to another class

4. A friend class cannot be referred to unless it is first declared. (True / False)

5. _____perform one operation on one kind of data but another operation on a different kind of data.

6. When an alias of a variable is passed to a function it is said to be a ___________________.

7. ________________ functions are best reserved for small, frequently used functions.

8. Overloaded functions
    a. are a group of functions with the same name
    b. have the same number and types of arguments
    c. must have constant values for arguments
    d. save memory space

9. The _______________ of an operator cannot be changed by overloading the operator.

10. The keyword _______________ introduces an overloaded function definition.

11. Operator overloading allows us to create symbols for new operators (True / False)

12. In a class X with three objects a1, a2, and a3, for the statement a3 = a1*a2; to work correctly, the overloaded * operator must
    a. take two arguments
    b. take no arguments
    c. use the object of which it is a member as an operand

d. create a temporary object

13. In the definition of an overloaded unary operator we require ___________ arguments.

14. When you overload a compound assignment operator, the result must be returned

    a. is assigned to the object on the right of the operator

    b. is assigned to the object on the left of the operator

    c. is assigned to the object of which the operator is a member

15. The operation of the assignment operator and the copy constructor are similar except that the copy constructor creates a new object. (True / False)

16. A ____________________ is invoked when a function returns a value.

17. The compiler does not automatically know how to convert between user-defined types and basic types. (True / False)

18. If there are two objects objectA and objectB which belong to different classes, the statement objectA = objectB will give an error. (True / False)

19. To convert from a user-defined type to a basic type, you would use a __________that is a member of the class.

5.10 Do It Yourself

1. Write a declaration for a function called default_test() that takes two arguments and returns the type float. The first argument is type `int` and the second argument is type float with a default value of 2.217.

2. Write a function called small_test() that is passed two arguments of type `int` as reference. The function should find the smaller of the two numbers and change it to -1.

3. Write declarations for two overloaded functions named overboard(). The first function should take one argument of type char and the second should take two arguments of type char. Both functions return the type `int`.

4. Write the following:

   a. Declaration of a friend function called Barney() that returns type `int` and takes an object of class Fred as an argument.

   b. A class Fred with a declaration to make all members of the class Barney friends of the class.

5. Write a function, fn(), to increment the values of two variables of type `int` that are passed by reference.

6. Design a class date that can be used in many applications. You can make functions to store a given date, compare dates, print the date in different formats and add dates. Write a program to test this class.

7. Write the complete definition for an overloaded ++ operator for the class `LFeet` from Example 12. It should add 1 to the feet member data so that you are able to execute the statement

   ```
   df++;
   ```

   where `df` is an object of the class `LFeet`.

8. Rewrite the operator function in Question 1 so that it is possible to execute statements such as `df2 = df1++;`

   where `df1` and `df2` are objects of class `LFeet`.

9. Write a program using a class called Alpha that has one data member of type `int`. The class should have constructors, an overloaded assignment operator and copy constructor. The program should display the result whenever an initialization or assignment takes place. For example, display the result of executing the following statements:

   ```
   Alpha obj1(50);

   Alpha obj2;

   obj2 = obj1;

   Alpha obj3(obj1);
   ```

```
Alpha obj4 = obj2;
```

10. Rewrite the program in Example 14 with overloaded >, == and < operators that work with the Oap class. The overloaded operators should allow you to check if a s1 is longer than, shorter than or of the same length as s2.

11. Write a program with a class, named Student. The class should have data members for the name and marks of a student. The program should check if a student's marks are, above, below, or equal to a pass mark of 35. If the student's marks are below 35, the student should be given an extra 5 marks. Use overloaded operator functions wherever possible. If the marks are still below 35 the student fails. Display the name and marks of the student and whether the student has passed or failed.

Summary

- Friend functions can access the private data of a class, even though they are not member functions of the class.
- An overloaded function is a group of functions with the same name. The function, which is executed when the function is called for depends on the type of arguments and the number of arguments supplied in the call.
- The ability to associate an existing operator with a member function and use it with objects of its class as its operands is called operator overloading. Operator overloading gives you the ability to redefine the language by allowing you to change the way operators work.
- Only operators that already exist in the language can be overloaded. You cannot use a new symbol.
- Overloading of operators is only available for classes. You cannot redefine the operators for the predefined basic types.
- Unary operators have only one operand. Examples of unary operators are the increment operator ++, the decrement operator --, and the unary minus operator.
- Binary operators that are overloaded, take one formal argument, which is the value to the right of the operator. When binary operators are overloaded by means of friend functions they take two arguments.
- Comparison and logical operators are binary operators that need two objects to be compared. The comparison operators that can be overloaded include <, <=, >, >=, ==, and !=.
- A copy constructor is called in three contexts:
  - when an object of a class is initialized to another of the same class
  - when an object is passed as an argument to a function
  - when a function returns an object.

Session - 6

Inheritance

Welcome to the Session, **Inheritance**.

This session introduces concepts of Inheritance. Further session describes about diffrent types of inheritance exhibited in C++. This session also describes about constructors and destructors in C++ and also describes concept of container classes.

At the end of this session, you will be able to:

- ➔ Describe single inheritance
- ➔ Describe Base classes and derived class
- ➔ Describe how to access base class members and use pointers in classes
- ➔ Describe types of inheritance
- ➔ Describe constructors and destructors under inheritance
- ➔ Describe how to call member functions of the base class and derived class
- ➔ Describe Container classes

6.1 Single Inheritance

Leading from the concept of classes in object-oriented programming, is another powerful feature called inheritance. In order to maintain and reuse class easily, we should to be able to relate classes of similar nature with another. For example let us consider people employed in an organization. There are employees who perform different duties, for example the director, a manager, a secretary, and a clerk. Now each of these employees have some common features that link all of them. For example, all employees must have:

- ➔ name
- ➔ age
- ➔ employee id
- ➔ salary structure
- ➔ department

These are the common properties they share. In addition to this, a director has certain number of people working under him, added responsibilities, and probably a different set of perks also. If we were to group the employees in different levels they could be categorized as level 1 for the director, level 2 for a manager, and so on. If we needed to put all this information together to form a class in an object-oriented program we would first consider a common class called Employee. This class can then be subdivided into classes such as Director, Manager, Secretary, and Clerk as shown in figure 6.1.

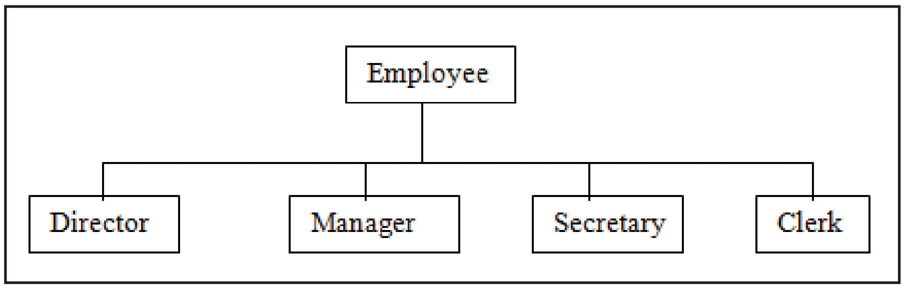


Figure 6.1: Class Employee and its Subclasses

Definition - Inheritance is the process of creating new classes from an existing base class.

The basic idea behind inheritance is that in a class hierarchy, the derived classes inherit the methods and variables of the base class. In addition they can have properties and methods of their own. Figure 6.2 depicts this.

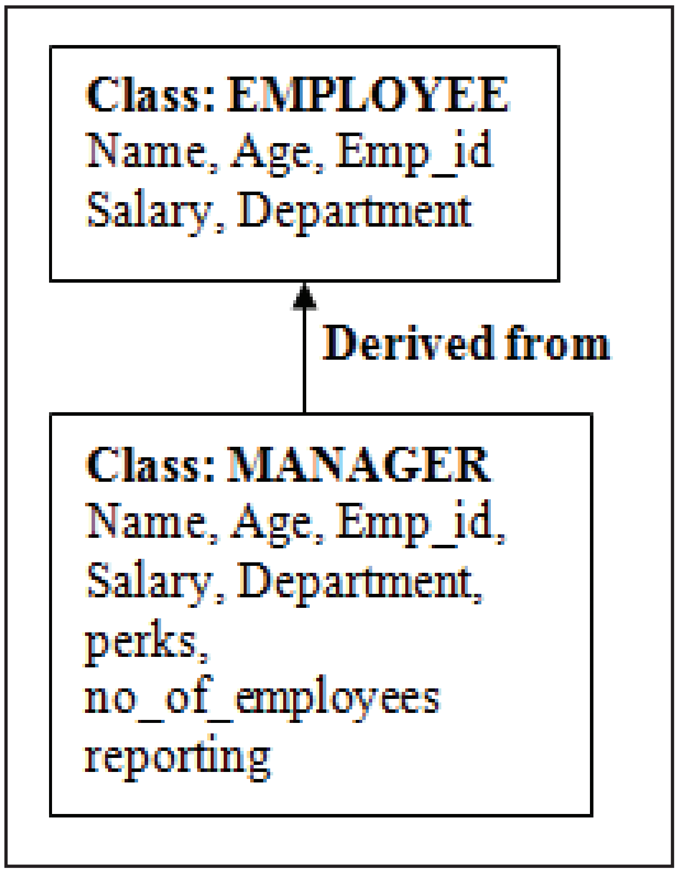


Figure 6.2: Derived Class Has Properties of Its Own

The class **Manager** that is derived from the class **Employee** inherits the common properties of name, age, employee id, salary and department. In addition, it has the data members perks and number of employees reporting, which are specific to the class Manager.

The most important advantage of inheritance is the reusability of code. Once a base class has been created it need not be changed but it can be adapted to work in different situations. One result of reusability of code is the development of class libraries.

Many vendors offer class libraries. A class library consists of data and methods encapsulated in a class. The source code of these class libraries need not be available to modify a class to suit one's needs. Modifying a class library does not require recompilation.

Deriving a class from an existing one allows redefining a member function of the base class and also adding new members to the derived class. This is possible without having the source code of the class definition. All that is required is the class library, which does not require recompilation. The base class remains unchanged in the process.

6.2 Base Class and Derived Class

Derivation can be represented graphically with an arrow from the derived class to the base class as shown in figure 6.3.

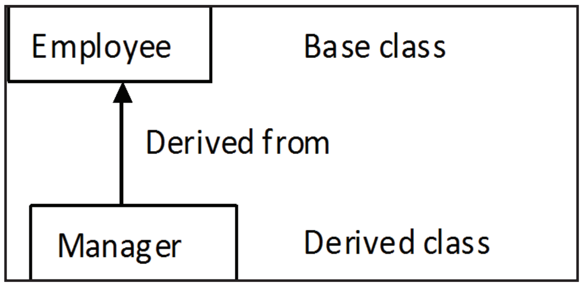


Figure 6.3: Manager Derived From Employee

The arrow in the diagram is meant to show that class **Manager** is derived from the base class **Employee**. The arrow pointing towards the base class signifies that the derived class refers to the functions and data in the base class, while the base class has no access to the derived class. These concepts will become clearer as we see them from the programmers point of view.

The declaration of a derived class is similar to that of any ordinary class except that we also have to give the name of the base class. For example,

```
class Manager : public Employee
```

Any class can be used as a base class. This means that a derived class can in turn be the base for another class. Therefore, a base class can be classified into two types:

- direct base
- indirect base

A base class is called direct if it is mentioned in the base list. For example:

```
class A
{ };
class B : public A
{ };
```

where class A is a direct class. An indirect class can be written as:

```
class A
{ };
class B : public A
{ };
class C : public B
{ };
```

Here B is derived from A, and C from B. The class A is the indirect base class for C. This can be extended to an arbitrary number of levels.

We will see how the base class and derived class interact with each other.

6.3 Accessing Base Class Members

An important aspect of inheritance is knowing when a member function or data member of a base class can be used by objects of the derived class. This is called **accessibility**. Let us first look at the base class members, which are defined with private and public access specifiers.

Class members can always be accessed by member functions within their own class, whether the members are private or public. Objects defined outside the class can access class members only if the members are public. For example, if `emp1` is an instance of class `Employee`, and `display()` is a member function of `Employee`, then in `main()` the statement

```
emp1.display();
```

is valid if `display()` is public.

With inheritance, the member functions of the derived class can access members of the base class if its members are public. The derived class members cannot access the private members of the base class.

There is one more section called the protected section. The protected section is like the private section in terms of scope and access that is, like private members, protected only members of that class can access members. Objects or functions from outside the class cannot access protected members within the class. This property of protected members is also similar to that of private members. However, the difference between them appears only in derived classes.

Private members of a class cannot be derived. Only public and protected members of a class can be derived. In other words, members of the derived class can access public and protected members; they cannot access the private members of the base class. This restriction is in conformance with the object-oriented concept of information hiding. The designer of a class may not want anyone to have access to some of the class members and those members can be put in the private section. On the other hand, the designer can allow controlled access by providing some protected members.

Inheritance does not work in reverse. The base class and its objects will not know anything about any classes derived from it. Table 6.1 summarizes the access for the different sections of the base class:

Access specifier	Accessible from own class	Accessible from derived class	Accessible from objects outside the class
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No

Table 6.1: Access Rules For Base Class Members

Using the rules in the table, let us look at an example.

Example 1:

```
class Employee{                       //base class
private:
      int privA;
protected:
      int protA;
public:
      int pubA;
};
class Manager: public Employee{       //derived class
public:
      void fn()
      {
      int a;
      a = privA;                  //error:not accessible
      a = protA;                  //valid
      a = pubA;                   //valid
      }
};
void main()
{
Employee emp;
                    //Object of base class type
      emp.privA = 1;              //error:not accessible
      emp.protA = 1;              //error:not accessible
      emp.pubA = 1;
               //valid
Manager mgr;
               //object of derived class
      mgr.privA=1;                    //error:not accessible
      mgr.protA=1;                    //error:not accessible
      mgr.pubA=1;                  //valid

}
```

In the example, you can see that the function `fn()` can access the `public` and `protected` base class members from the derived class. However, in `main()` only the `public` base class member can be accessed.

While writing a class if you foresee it being used as a base class in the future, then any data or functions that the derived classes might need to access should be made `protected` rather than `private`.

➔ **Pointers in classes**

We need to understand how we can use a pointer, which has been declared to point to one class, to actually refer to another class. If we referred to an `Employee` we could be referring to a `Manager`, a `Secretary`, a `Clerk`, or any other kinds of `Employee`, because we are referring to a very general form of an object.

If however, we were to refer to a `Manager`, we are excluding `Secretaries`, `Clerks`, and all other kinds of `Employees`, because we are referring to a `Manager` specifically. The more general term of `Employee` can therefore, refer to many kinds of `Employees`, but the more specific term of `Manager` can only refer to a single kind of `Employee`, namely a `Manager`.

We can apply the same idea and say that if we have a pointer to an `Employee`, we can use that pointer to refer to any of the more specific objects. Similarly, if we have a pointer to a `Manager`, we cannot use that pointer to reference any of the other classes including the `Employee` class because the pointer to the `Manager` class is too specific and restricted to be used on any of the other classes.

The general rule is that if a derived class has a public base class, then a pointer to the derived class can be assigned to a variable of type pointer to the base. For example, because a `Manager` is an Employee, a `Manager*` can be used as an `Employee*`. However, an `Employee*` cannot be used as a `Manager*`.

For example, using the `Employee` and `Manager` classes of Example 1,

Example 2:

```
void main()
{
Manager mgr;
Employee* emp = &mgr; //valid:every Manager is an Employee
Employee eml;
Manager* man = &eml; //error: not every Employee is a Manager
}
```

Therefore, an object of a derived class can be treated as an object of its base class when manipulated through pointers. However, the opposite is not true. At a later stage we will see how this use of pointers will be a handy feature.

6.4 Inheritance and Access Keywords

C++ provides different ways to access class members. Member access control depends on the way the derived class is declared. A derived class can be declared with one of the specifiers i.e., public, private and protected.

The access specifier determines how elements of the base class are inherited by the derived class. When the access specifier for the inherited base class is public, all the public members of the base become public members of the derived class. If the access specifier is private, all public members of the base class become private members of the derived class. In either case, any private members of the base remain private to it and are in accessible by the derived class.

Example 3:

```
class A{                    //base class
private:
    int privA;
protected:
    int protA;
public:
    int pubA;
};
class B: public A       //publicly derived class
{
 public:
    void fn()    {
    int a;
    a = privA;          //error: not accessible
    a = protA;          //valid
    a = pubA;           //valid
    }};
class C: private A      //privately derived class
{
public:
    void fn(){
    int a;
    a = privA;          //error: not accessible
    a = protA;          //valid
    a = pubA;      //valid
    }
};
void main(){
int m;
B obj1;                 //object of publicly derived class
    m = obj1.privA;         //error: not accessible
    m = obj1.protA;         //error: not accessible
    m = obj1.pubA;          //valid: B is publicly derived from class A
C obj2;                 //object of privately derived class
    m = obj2.privA;     //error: not accessible
    m = obj2.protA;     //error: not accessible
    m = obj2.pubA;          //error: not accessible: C is privately
                        // derived from class A
}
```

The example shows a base class, A, with private, protected, and public data members. The class B is derived publicly from A and C is privately derived from A.

Note: If no access specifier is given while creating the class, private is assumed.

Functions in the derived classes can access protected and public members in the base class. You can observe this from the statements in the class declarations of class B and C. Objects of the derived classes, cannot access private or protected members of the base class from outside the class . This is what we have seen earlier.

Objects of class B, which is **publicly** derived from A **can** access public members of the base class. However, objects of the class C, which is **privately** derived from A, **cannot** access any of the members of the base class. You can see this from the statements in `main()`.

If the derived class is declared **protected**, the access control is similar to privately derived classes. In this case the functions in a derived class can access protected and public members of the base class. However, **objects** of the derived class (in main or outside the class) **cannot** access any of the members of the base class.

Table 6.2 summarizes the accessibility as it is called for the different access specifiers of the derived classes.

Base Class Members	Public Inheritance	Private Inheritance	Protected Inheritance
Public	Public	Private	Protected
Protected	Protected	Private	Protected
Private	Not inherited	Not inherited	Not inherited

Table 6.2: Accessibility For Derived Classes

There is an easy way to remember this table. First of all, derived classes have no access to private members of a base class. Secondly, inheriting the base class publicly does not change the access levels of the members inherited by the derived class from the base. The other two access levels of the base class cause all inherited members to be of the same access level, as the base class (private for private base, protected for protected base). The type of derivation we do, `public`, `private` or `protected`, will affect the access that the derived class functions have over the members of the base classes in a multi-level inheritance like the one shown in figure 6.4.

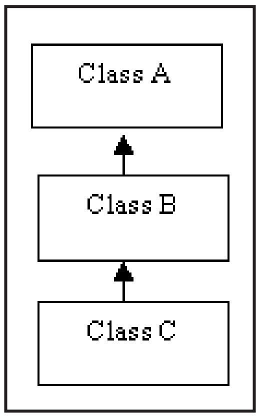


Figure 6.4: Type of Derivation in Multi-Level Inheritance

In the following code the class B derives privately from class A and class C in turn derives publicly from class B.

Example 4:

```
class A
{
public:
     int a;
};
class B : private A
{
public:
     int b;
     void func_b()
     { int x,y;
      x=a;                    // valid
y=b;                 // valid
     }
};
class C : public B
{
public:
     void func_c()
     { int x,y;
x=a;      // not valid
y=b;      // valid
     }
};
```

In the code, class B is privately derived from class A. So, the data member 'a' of class A can be accessed by the member functions of class B but becomes private after derivation. As a result the class C which even though being publicly derived from class B cannot access the data member 'a'. Thus, in class B the function **func**_b can access the data member 'a' while the func_c of class C cannot access.

6.5 Constructors and Destructors under Inheritance

The rules for creating an object of a derived class are not complicated. The constructor of the base part of an object is first called, then the appropriate constructor of the derived class is called. Example 5 shows how base and derived class constructors can be defined.

Example 5:

```
class Base
{
protected:
 int a;

 public:
      Base(){a=0;}                //default constructor
      Base(int c){a=c;}           //one-arg constructor
};
class Derived:public Base
{
public:
      Derived():Base(){}          //default constructor
      Derived(int c):Base(c){}    //constructor with one-arg
};
```

When you declare an object of the derived class, with the statement

`Derived obj;`

it will cause the constructor of the base class to be called first and then the constructor of the derived class. You can see that there is a difference in the declaration of the derived class constructor from constructors we have used so far. The base class constructor is given after the derived class constructor, separated by a colon, as in,

`Derived():Base(){}`

This applies not only to the default constructors of the classes, but you can also explicitly select which constructor of the base class should be called during a call to the constructor of a derived class. The following statement

`Derived obj1(20);`

uses the one-argument constructor in the class **Derived**. This constructor also calls the corresponding constructor in the base class.

```
Derived(int c):Base(c);        //argument c is passed to Base
```

The argument `c` is passed from `Derived` to `Base`, where the one-argument constructor is used to initialize the object. The derived class constructor is responsible for initializing the additional data members of the derived class.

Destructors are called in the reverse order to constructors. This means that the destructors will be called for the derived class first, then the destructor for the base class will be called. A destructor for the derived class is to be defined only if its constructor allocates any memory through dynamic memory management. If the constructor of the derived class does not do anything or no additional data members are added in the derived class, then the destructor for the derived class can be an empty function.

6.6 Calling Member Functions

Member functions in a derived class can have the same name as those in the base class. When the function is invoked with the object of the base class, the function from the base class is called. When you use the name of the derived class object, the function from the derived class is invoked.

If a member function from the derived class wants to call the base class function of the same name, it must use the scope resolution operator as shown in the example.

Example 6:

```
class Base
{
protected:
     int ss;
public:
     int func()
{return ss;}
};
class Derived: public Base
{
public:
     int func()
{return Base::func();}//calling base class func
};
```

```
void main()
{
     Base b1;             //base class object
     b1.func();                //calls base class func
Derived a1;           //derived class object
a1.func();                //calls derived class func
}
```

The function in a base class can be invoked using the objects of the base class as well as the derived class. If a function exists in the derived class and not in the base class, it can be invoked only with the objects of the derived class. The object of the base class does not know anything about the derived class and will always use the base class functions only.

6.7 *Container Classes*

In our example of the employee, the manager is an employee or the secretary is an employee. In contrast, a class X that has a member of another class Y is said to have a Y or X contains Y. This relationship is called a membership or a “has a” relationship.

```
class X{      //X contains Y
.
.
public:
     Y abc;
};
```

When a class contains an object of another class as its member it is called a container class. Consider the example of a jet plane. You might want to consider deriving a class for a jet plane from a class called engine. However a jet plane is not an engine but it has an engine. To be able to decide whether to use inheritance or membership you can ask whether a jet plane has more than one engine. If that is a possibility, it is most likely that the relationship will be “has a” rather than “is a”. Not all relationships will fall under clear-cut categories. Therefore, using a derived class or a container class is a matter to be considered at the design stage.

Let us look at an example of a container class to understand how constructors are defined.

Example 7:

```
class engine:
{
private:
     int num;
public:
engine(int s)
{num=s;}
};
class jet
{
private:
     int jt;
     engine eobj;          //declaring an object here
public:
     jet(int x, int y):eobj(y)
     {jt=x;}
};
```

As you can see, the class `jet` contains an object `eobj` in its private section. It also contains a constructor to which two variables `x` and `y` are passed. The variable `x` is used to initialize the private data member of the class `jet`.

In the constructor of class `jet` we would expect that after the colon a base class would be called. However, we are not dealing with inheritance and there is no base class. In this case, the name of the object of the class **engine** is written after the colon. It tells the compiler to initialize the `eobj` data member of class `jet` with the value in `y`. It is exactly like declaring an object of the class engine with the statement,

```
engine eobj(y);
```

Variables of any data type can be initialized like this.

6.8 Check Your Progress

1. If the class Alpha inherits from the class Beta, class Alpha is called the ______________ and class Beta is called the _____________ class.

2. Inheritance enables ____________ which saves time in development and encourages using class libraries.

3. An object of a derived class can be treated as an object of its corresponding public base class.

(A)	True	(B)	False

4. A derived class can be made the base for another class.

(A)	True	(B)	False

5. The three types of access specifiers are ____________, _______________ and _______________.

6. A derived class cannot access the public members of the base class.

(A)	True	(B)	False

7. When deriving from a base class with protected inheritance, public members of the base class become__________members of the derived class, and protected members of the base class become ____________ members of the derived class.

8. A "has a" relationship between classes represents ________________ and an "is a: relationship between classes represents _______________.

9. The private members of a class can be accessed by:

(A)	Member functions of the class
(B)	Non-member functions of the class
(C)	Member functions of a derived class if it is declared privately
(D)	Member functions of a derived class if it is declared publicly

10. Classify the following into "is a" and "has a" relationships.

(A)	country and capital
(B)	file and records
(C)	modem and input/output devices
(D)	car and door
(E)	bicycle and vehicle

6.9 Do It Yourself

1. Given the following classes:

```
class Alpha{
 public:
 int x;
};
class Beta: public Alpha
{
 public:
 int x;
 };
```

   a. Write statements in a main() program to initialize the data member of the base class and the derived class.

   b. If the data members of the base class and derived class were private, write statements to assign values to the data members in main().

   c. Using these classes with public data members, write default constructors and one-argument constructors for both the classes.

2. Design a class of Shapes that can be subdivided into rectangles, triangles, circles, and ellipses. Use inheritance to classify the shapes, look for common features such as width, height, alignment point, and methods such as draw, initialize, set_alignment that can be part of the base class, and see if the shapes can be divided further into subclasses.

3. Design the class Employee that is referred to in this session with private data members name, employee id, designation and department, public member functions to get the information from the user and display it. Design a manager class with private data members for perks (you can give a figure in terms of dollars) and number of employees reporting to the manager. Also design a secretary class with an additional data member giving the name of the boss that the secretary reports to. Give suitable member functions to display these details. Write a program to test the classes and display all the information given by the user.

4. Create class Point. It has the data members x-co-ordinate and y-co-ordinate. The class declaration for Point should also contain member functions to set the point, get the x-co-ordinate and y-co-ordinate. It should have a constructor with default arguments. Create another class called Circle, which contains a data member called radius and an object of class Point, in which we can store the centre point of the circle. Note that this is a container class. The class Circle has a constructor with default arguments. It should have the facility to set the radius, get the radius and calculate the area. Also include a member which prints the details of the circle (i.e. centre point, radius, area).

Summary

- Inheritance is the process of creating new classes from an existing class.
- Single inheritance is the process of creating new classes from an existing base class.
- The most important advantage of inheritance is the reusability of code.
- Private members of a class cannot be derived. Only public and protected members of a class can be derived. In other words, members of the derived class can access public and protected members only, they cannot access the private members of the base class.
- The keyword public in the class declaration of the derived class specifies that objects of the derived class are able to access public member functions of the base class.
- With the keyword private in the derived class declaration, objects of the derived class cannot access public member functions of the base class.
- If the base class is declared as protected in the derived class, the access control is similar to privately derived classes. Functions in a protected derived class can access protected and public members in the base class.
- The constructor of the base part of an object is first called, then the appropriate constructor of the derived class is called. You can explicitly select which constructor of the base class should be called during a call of the constructor of a derived class.
- Destructors are called in the reverse order to constructors. Destructors will be called for the derived class first, then for the base class.

Session - 7

Multiple Inheritance and Polymorphism

Welcome to the Session, **Multiple Inheritance and Polymorphism**.

This session introduces concept of multiple inheritance. The session describes concept of virtual base classes, pointers to derived classes, virtual functions and abstract classes.

At the end of this session, you will be able to:

- ➔ Describe multiple inheritance
    - Constructors under multiple inheritance
    - Ambiguity in multiple inheritance
    - Multiple inheritance with a common base
- ➔ Describe virtual base classes
- ➔ Use pointers to derived classes
- ➔ Describe virtual functions
- ➔ Describe polymorphism
- ➔ Describe dynamic binding
- ➔ Describe pure virtual functions
- ➔ Describe abstract classes
- ➔ Describe virtual destructors

7.1 Multiple Inheritance

Multiple Inheritance occurs when a class inherits from more than one parent. As an example of multiple inheritance let us consider a program which models an educational institution, say, a college. In this institution we have teachers and students. Let us consider that the college employs some junior teaching assistants or research associates. It very often happens that to qualify for further promotions the teaching assistant has to acquire higher educational qualifications. In this case the teaching assistant falls under the category of a student also. Thus, the teaching assistant can derive properties from the class teacher as well as the class student. There are two base classes, the teacher and the student and the derived class is the teaching assistant.

Definition:

Multiple inheritance is the process of creating a new class from more than one base class.

The basic idea behind inheritance is that in a class hierarchy, the derived classes inherit the methods and variables of the base class. In addition they can have properties and methods of their own as shown in figure 7.1

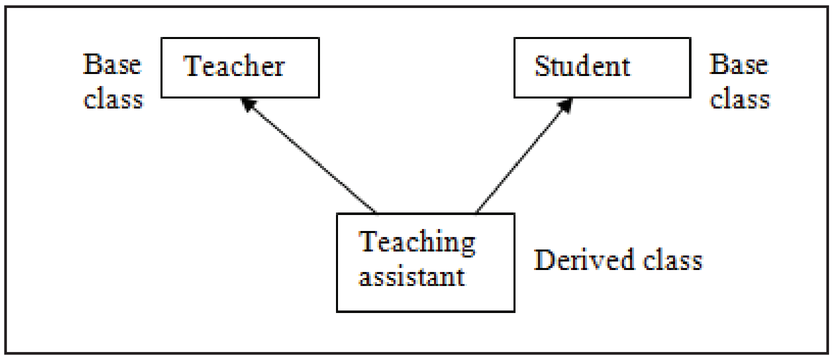


Figure 7.1: Multiple Inheritance

The derived class inherits the properties of two or more base classes. Multiple inheritance can combine the behavior of many base classes in a single class. A multiple inheritance hierarchy represents a combination of its base classes.

The syntax for multiple inheritance is as shown as follows:

```
class Teacher
{ };
class Student
{ };
class Teach_asst: public Teacher, public Student
```

The names of both the base classes are provided separated by a comma. The rules of inheritance and access for multiple inheritance are the same as for single inheritance.

7.1.1 Constructors under Multiple Inheritance

The following example illustrates how constructors are handled in multiple inheritance.

Example 1:

```
class Teacher
{
private:
     int x;
public:
Teacher(){x=0;}                     //constructors
     Teacher(int s){x=s;}
};
class Student
{
private:
     int y;
public:
Student(){y=0;}                  //constructor
Student(int a){y=a;}
};
class Teach_asst:public Teacher,public Student
{
private:
     int z;
public:
     Teach_asst():Teacher(),Student()        //constructor
     {z=0;}
     Teach_asst(int s, int a, int b):Teacher(s),Student(a)
     {z=b;}
};
```

Like with normal inheritance, constructors have to be defined to initialize the data members of all classes. The constructor in `Teach_asst` calls constructors for the base classes. The names of the base class constructor follow the colon and are separated by commas as in the statement,

```
Teach_asst():Teacher(),Student();           //constructor
```

If the constructors had arguments then the constructor in class `Teach_asst` would have to supply value for their arguments, besides arguments for its own constructor. We can see how this is done in the following statement.

```
Teach_asst(int s,                      arg for Teacher class
 int a,              arg for Student class
 int b):             arg for this class
 Teacher(s),         call Teacher constructor
 Student(a)                  call Student constructor
  {z = b;}                            set own data member
```

The first two arguments passed to the `Teach_asst` constructor are passed on to `Teacher()` and `Student()`. The last argument is used to initialize the data member in the `Teach_asst` class.

When constructors are used in the base class and derived classes, the base class is initialized before the derived class, using either the default constructor or a constructor with arguments depending on the code of the constructor of the derived class. The base classes are initialized first, in the order they appear in the list of base classes in the declaration of the derived class. If there are member objects in the class, they are initialized next, in the order they appear in the derived class declaration (Example 1 does not have any member objects in the derived class). Finally the code of the constructor is called. For example in an object of class `Teach_asst`, the sequence of calls will be:

1. Base classes as they appear in the list of base classes: `Teacher`, `Student`
2. The object itself (using the code in its constructor)

The call for the destructor of a derived class follows the same rules as constructors, but in the reverse order. The destructor of the class is called first, then those of member objects, and then the base classes.

7.1.2 Ambiguity in Multiple Inheritance

Some problems can surface in cases where two base classes have the same function or data member name. The compiler will not be able to understand which function to use. Let us look at an example.

Example 2:

```
class Alpha{
 public:
     void display();
};
class Beta{
 public:
     void display()
};
class Gamma : public Alpha, public Beta
{ };
void main()
{
 Gamma obj;
     obj.display();       //ambiguous: cannot be compiled
}
```

Since there is an ambiguity about which `display()` function is called by the object `obj` the compiler will issue an error message. To access the correct function or data member you will need to use the scope resolution operator. For example,

```
obj.Alpha::display();
obj.Beta::display();
```

It is upto the programmer to avoid such conflicts and ambiguities. It can be resolved by defining a new function `display()` in the derived class `Gamma`, such as:

```
void Gamma::display()
{
 Alpha::display();
 Beta::display();
}
```

This localises the information about the base classes. Since `Gamma::display()` overrides the `display()` functions from both its base classes, this ensures that `Gamma::display()` is called wherever `display()` is called for an object of type Gamma. The compiler detects the name clashes and resolves it.

7.1.3 Multiple Inheritance with a Common Base

We have seen that it is possible to use more than one base class to derive a class. There are many combinations which inheritance can be put to. There is the possibility of having a class as a base twice. For example, in the example of Teaching assistant, the classes Teacher and Student could have been derived from a class Person. Now Teaching assistant has a common ancestor - the Person class.

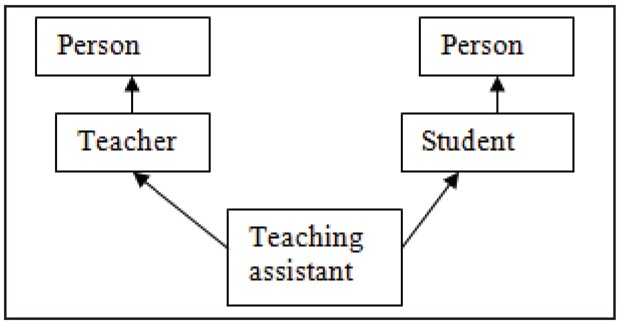


Figure 7.2: Multiple Inheritance with Common Base

Figure 7.2 is also referred as cyclic inheritance. One potential problem here is that both Teacher and Student contain a copy of the Person class members. Now when Teaching assistant is derived from both Teacher and Student it contains a copy of the data members of both classes. That means that it contains two copies of the Person. Figure 7.3 depicts this.

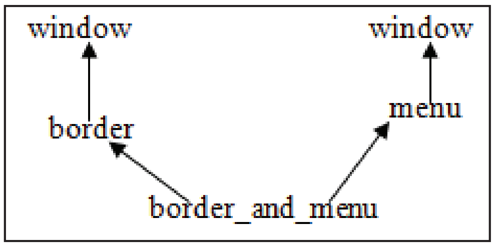


Figure 7.3: Object with a Common Base Class

class members - one from Teacher and one from Student. This gives rise to ambiguity between the base class data members. Another problem is that declaring an object of class Teaching assistant will invoke the Person class constructor twice. The solution to these problems is a virtual base class.

7.2 Virtual Base Classes

We have seen that multiple inheritance is the process of creating a new class from more than one base class. Multiple inheritance hierarchies can be complex, and may lead to a situation in which a derived class inherits multiple times from the same indirect class as we have seen earlier. For example, consider a class `window`, which has two directly derived classes `border` and `menu`. Now if we had another class `border_and_menu`, which is derived from the class `border` and `menu`, the class `window` would be a common base class.

Consider that the base class `window` has a data member `basedata`.

Example 3:

```
class window{
 protected:
 int basedata;
.
};
class border : public window
{ };
class menu : public window
{ };
class border_and_menu : public border, public menu
{
public:
int show()
{return basedata;}
};
```

A compiler error will occur when the `show()` function in the derived class `border_and_menu` tries to access `basedata` in the class `window`. When the classes `menu` and `border` are derived from `window`, each inherits a copy of `window`. This copy is called a subobject. Each of these subobjects contains its own copy of `basedata` from the class window. When the class `border_and_menu` refers to `basedata` the compiler does not know which copy is being accessed and hence, the error occurs. To avoid two copies of the base class we use a virtual base class. To make a base class virtual just include the keyword `virtual` when listing the base class in the derived class declaration.

```
class border: virtual public window
{
.
};
```

and

```
class menu: virtual public window
{
.
};
```

The use of the keyword virtual in these two classes `border` and `menu` causes them to share a single common object of their base class `window`.

Therefore, the class window is the `virtual base`. An object of `border_and_menu` can now be as shown in figure 7.4:

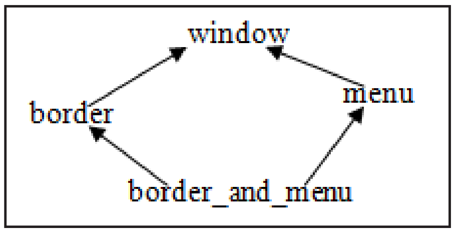


Figure 7.4: Object with a Virtual Base Class

This is different from multiple inheritance from a common base. Here, the virtual base class is represented by a single object of the class.

Virtual base classes are useful to avoid unnecessary repetitions of the same data member in a multiple inheritance hierarchy.

7.3 Pointers to Derived Classes

Pointers can point to objects as well as to simple data types. We have seen how to use pointers in classes in the previous session at some length; one special aspect was deferred until now because it related specifically to inheritance and virtual functions. The feature is this: a pointer declared as a pointer to a base can also be used to point to any class derived from that base. Consider two classes called **`exp1`** and **`exp2`**, where **`exp2`** inherits **`exp1`**. Now for this scenario, the following statements are true:

```
exp1 *p; // base class pointer

exp1 exp1_b; // object of type exp1

exp2 exp2_b; // object of type exp2

// p can, point to base objects

p=&exp1_b; // p points to exp1 object

// p can also point to derived objects without error

p=&exp2_b; // p points to exp2 object
```

As shown here, a base pointer can point to an object of any class derived from that base without generating a type mismatch error. Normally a pointer can only point to an object of the pointer's base type.

Although one can use a base pointer to point to a derived object, one can access only those members of the derived object that were inherited from the base. This is because the base pointer has knowledge only of the base class. It knows nothing about the member added by the derived class.

While it is permissible for a base pointer to point to a derived object, the reverse is not true. A pointer of the derived type cannot be used to access an object of the base class.

7.4 Virtual Functions

A virtual function is a member function that is declared within a base class and redefined by a derived class. Consider a program to draw different shapes like triangles, circles, squares, ellipses etc. Consider that each of these classes has a member function `draw()` by which the object is drawn. To be able to put all these objects together in one picture we would have to find a convenient way to call each object's `draw()` function. Let us look at the class declarations for a class Shapes and derived classes `circle` and `square`.

Example 4:

```
class Shapes
{
.
.
.
public:
     void draw()            //function in the base class
{cout<<"Draw Base\n";}
};
class circle: public Shapes
{
private:
     int radius;
public:
     circle(int r);
void draw()            //redefined in derived class
{cout<<"Draw circle";}
};
class square: public Shapes
{
private:
     int length;
public:
square(int l);
```

```
 void draw()            //redefined in derived class
 {
   cout<<"Draw square";}
};
void main()
{
 circle c;
 square s;
 Shapes* ptr;

 ptr = &c;
 ptr->draw();

 ptr = &s;
 ptr->draw();
}
```

When pointers are used to access classes the arrow operator (->) can be used to access members of the class. We have assigned the address of the derived class to a pointer of the base class as seen in the statement,

```
ptr = &c;       //c is an object of class circle
```

Now when we call the `draw()` function using,

```
ptr->draw();
```

we expect that the `draw()` function of the derived class would be called. However, the result of the program would be,

```
Draw Base
Draw Base
```

Instead, we want to be able to use the function call to draw a square or any other object depending on which object `ptr` pointed to. That means, different `draw()` functions should be executed by the same function call.

For us to achieve this, all the different classes of shapes must be derived from a single base class `Shapes`. The `draw()` function must be declared to be `virtual` in the base class. The `draw()` function can then be redefined in each derived class. The `draw()` function in the base class declaration would be changed as shown,

```
class Shapes
```

```
{
.
.
.
public:
      virtual void draw()              //virtual in the base class
{cout<<"Draw Base\n";}
};
```

The keyword `virtual` indicates that the function `draw()` can have different versions for different derived classes. A virtual function allows derived classes to replace the implementation provided by the base class. The compiler makes sure the replacement is always called whenever the object in question is actually of the derived class. With the virtual function, if we were to execute the program in Example 4 we would get,

```
Draw circle
Draw squarei am
```

It is the member functions of the derived classes that are executed and not that of the base class. The same function call,

```
ptr->draw();
```

executes different functions depending on the contents of `ptr`.

The virtual function should be declared in the base class and cannot be redeclared in a derived class. The return type of a member function must follow the keyword `virtual`. If a hierarchy of derived classes is used, the virtual function has to be declared in the top-most level. Here, the top-most class is `Shapes` and the virtual function is declared in it.

A virtual function must be defined for the class in which it was first declared (unless it is a pure virtual function as we shall see later). Then even if no class is derived from that class, the virtual function can be used by the class. The derived class that does not need a special version of a virtual function need not provide one.

To redefine a virtual function in the derived class you need to provide a function exactly matching the virtual function. It must have the same parameters (same number and data type), otherwise the compiler thinks you want to overload the virtual function. The return type does not have to match. For example, if the virtual function of a class returns a pointer to its class, you can redefine the function in a derived class and have it return a pointer to the derived class, instead of a pointer to the base class. This is possible only with virtual member functions.

The derived class can either fully replace (override) the base class member function, or the derived class can partially replace (augment) the base class member function. The latter is accomplished by having the derived class member function call the base class member function, if desired.

7.5 Polymorphism

The word 'poly' originates from a Greek word meaning 'many' and 'morphism' from the word meaning 'form'. Thus we have 'polymorphism' which means 'many forms'. In more practical terms, polymorphism allows you to refer to objects of different classes by means of the same program item like a base class, and to perform the same operation in different ways depending on which object is being referred to at that moment.

For example, when describing the class mammals, you might note that eating is a fundamental 'operation' for mammals to live. Every kind of mammal needs to perform the function EAT. However different mammals have different responses to the function EAT. A cow might ruminate in a field of grass, a lion might devour a deer, and a child might eat a chocolate fudge ice- cream. Polymorphism is the process of taking an object of type mammal and telling it to EAT. The object will perform the action that is most appropriate for it. Objects are polymorphic if they have some similarities but are still somewhat different.

Definition:

Polymorphism is the process of defining a number of objects of different classes in a group and using different function calls to carry out the operations of the objects.

In other words, polymorphism means, 'to carry out different processing steps by functions having the same messages'.

The facility to invoke an appropriate function from any of the given classes using the pointer of the base class is a form of polymorphism. This is what we have done by using virtual functions.

Note: We have earlier used the keyword virtual to define a base class. However that is in a different context and is not related to polymorphism.

7.6 Dynamic Binding

In earlier sessions we have seen polymorphism with reference to function overloading and operator overloading. Choosing a function in the normal way during compile time is called early binding or static binding. Non-virtual member functions are resolved statically. That is, the member function is selected statically (at compile-time) based on the type of the pointer (or reference) to the object. The compiler uses the static type of the pointer to determine whether the member function invocation is legal.

In contrast, virtual member functions are resolved dynamically (at run-time). That is, the member function is selected dynamically (at run-time) based on the type of the object, not the type of the pointer/reference to that object. This is called *dynamic binding or late binding*.

Definition:

Dynamic binding means that the address of the code in a member function invocation is determined at the last possible moment, based on the dynamic type of the object at run time.

It is called dynamic binding because the binding to the code that actually gets called is accomplished dynamically (at run time). Dynamic binding requires some overhead in processing but provides increased power and flexibility in programming. Dynamic binding is a result of virtual functions.

If the keyword `virtual` is used with the function declaration in the base class, the system will use dynamic binding, which is done at run time, but if the keyword is not included, static binding will be used. What these words actually mean is that with dynamic binding, the compiler does not know which method will actually respond to the message because the object that a pointer points is not known at compile time. With static binding, however, the compiler decides at compile time what method will respond to the message sent to the pointer to an object.

7.7 Pure Virtual Functions

Consider a base class **`Shape`**, from which one can derive classes defining specific shapes, such as Circle, **`Ellipse`**, **`Rectangle`**, **`Curve`** and so on. The Shape class might include a virtual function **`draw()`** that we would call to draw a particular shape, but the shape class itself is abstract: there is no meaningful implementation of the draw() function for the Shape class. This is a job for a pure virtual function. A pure virtual function is a type of function, which has only a function declaration. A pure virtual function is declared by assigning the value of zero to the function as in,

```
virtual void getdata() = 0;
```

Every derived class must include a function definition for each pure virtual function that is inherited from the base class if it has to be used to create an object. This assures that there will be a function available for each call.

No function call will ever need to be answered by the base class since it does not have an implementation for a pure virtual function. You cannot create an object of any class, which contains one or more pure virtual functions, because there is nothing to answer if a function call is sent to the pure virtual method.

7.8 Abstract Classes

A class containing one or more pure virtual functions cannot be used to define an object. The class is therefore, only useful as a base class to be inherited into a usable derived class. It is sometimes called an abstract class. No objects of an abstract class can be created. The abstract class can only be used as a base for another class. For example:

Example 5:

```
class Shapes{
.
.
public:
     virtual void draw() = 0;        //pure virtual function
virtual void rotate(int) = 0;    //pure virtual function
};
```

```
class circle: public Shapes{
private:
     int radius;
public:
circle(int r);
void draw();
void rotate(int){ }
};
```

If a class inherits an abstract class without providing a definition for the pure virtual function, then it too becomes an abstract class and cannot be used to define an object.

An important use of abstract classes is to provide an interface without exposing any implementation details. You will find abstract classes used in many commercially available libraries and in application frameworks.

7.9 Virtual Destructors

A destructor is invoked to free memory storage automatically. However the destructor of the derived class is not invoked to free the memory storage which was allocated by the constructor of the derived class. This is because destructors are non-virtual and the message will not reach the destructors under dynamic binding. It is better to have a virtual destructor function to free memory space effectively under dynamic binding. Let us look at an example.

Example 6:

```
class Alpha{
private:
     char* alpha_ptr;
public:
     Alpha()                //constructor cannot be virtual
{alpha_ptr = new char[5];}
     virtual void fn();         //virtual member function
     virtual ~Alpha()           //virtual destructor
{delete[] alpha_ptr;}
};
```

```
class Beta : public Alpha{
private:
 char* ptrderived;
public:
 Beta()
 {ptrderived = new char[100];}
 ~Beta()
 {delete[] ptrderived;}
};
void main()
{
 Alpha *ptr = new Beta;
 delete ptr;
}
```

When you `delete` a derived object via a base pointer a virtual destructor is required. For example, when we use,

`delete ptr;`

we are deleting a derived object using a pointer of the base class.

Virtual functions bind to the code associated with the class of the object, rather than with the class of the pointer/reference. When you say `delete ptr`, and the base class has a `virtual` destructor, the destructor that gets invoked is the one associated with the type of the object `*ptr`, rather than the one associated with the type of the pointer. As a derived class instance always contains a base class instance, it is necessary to invoke destructors of both the classes in order to ensure that all the space on the heap is released.

In general, it is a good idea to supply a `virtual` destructor in all classes that act as a base class. The idea behind this is, if you have any `virtual` functions at all, you are probably going to be doing some operations on derived objects via a base pointer, and some of the operations you may do may include invoking a destructor (normally done implicitly via `delete`).

A constructor cannot be virtual because it needs information about the exact type of the object it is creating to be able to construct it correctly.

7.10 Check Your Progress

1. C++ provides for _____________ which allows a derived class to inherit from many base classes even if these base classes are not related.

2. A pure virtual function is specified by placing __________ at the end of its prototype in the class definition.

3. If a class contains one or more pure virtual functions it is called an ____________.

4. Polymorphism is implemented via virtual base classes.

(A)	True	(B)	False

5. If there is a pointer `ptr` to objects of a base class, and it contains the address of an object of a derived class, and both classes contain a non-virtual member function `getdata()`, then the statement `ptr->getdata();` will call the function of the _________ class.

6. A function call resolved at run time is referred to as __________ binding and a function call resolved at compile time is referred to as ______________ binding.

7. There are many situations in which it is useful to define classes for which the programmer never needs to instantiate any objects. Such classes are:

(A)	Virtual base classes
(B)	Abstract classes
(C)	Base classes
(D)	Virtual abstract classes

8. A pointer to a base class can point to objects of a derived class.

(A)	True	(B)	False

9. When you delete a derived object via a base pointer you require.

(A)	Virtual constructor
(B)	Pure virtual function
(C)	Virtual destructor
(D)	Virtual destructor and virtual constructor

7.11 Do It Yourself

1. Create a class called `base1` which has one data member called `value` which is of type integer. Create another class called `base2`, which has one data member called index which is of type integer. Both these classes have a constructor with default argument and another member function to get the data and both have the same name which is `getdata()`. Derive a class from the two classes and name it as `derived`. It has one data member, which is called `real` and is of type `float`. The derived class has a constructor with default arguments and another function to get the value for `real`. Test this class with a program and see to it that a derived class object calls the `getdata()` function.

2. Create a class called `employee`. It stores the first name and the last name. This information is common to all employees including derived classes. From class employee now derive classes hourly worker, pieceworker, boss, commission worker. The hourly worker gets paid by the hour with 50% extra for overtime hours in excess of 40 hours per week. The pieceworker gets paid a fixed rate per item produced. Assume that this person makes only one type of item. The boss gets a fixed rate per week. The commission worker gets a small fixed weekly base salary plus a fixed percentage of that person's gross sales for the week. Every class should have a constructor, destructor and a function for displaying the data called `print()`.

   Test the classes using the following specifications

   a. Do not use pointers and see to it that an object of every class calls its corresponding print() function.

   b. Use pointers. Assign a derived class object to a base class pointer and a base class object to a derived class pointer. Now call the print function and observe what happens.

   c. Make employee an abstract base class.

Summary

➔ Multiple inheritance is the process of creating a new class from more than one base class. The derived class inherits the properties of two or more base classes. Multiple inheritance can combine the behavior of many base classes in a single class.

➔ If we need to access the member functions in the object pointed to by a pointer we need to use the arrow operator (->).

➔ A virtual function is a member function that is declared within a base class and redefined by a derived class. A virtual function allows derived classes to replace the implementation provided by the base class. The compiler makes sure the replacement is always called whenever the object in question is actually of the derived class.

➔ The virtual function is declared in the base class and cannot be redeclared in a derived class. The return type of a member function must follow the keyword virtual. If a hierarchy of derived classes is used, the virtual function has to be declared in the top-most level.

➔ Polymorphism is the process of defining a number of objects of different classes in a group and using different function calls to carry out the operations of the objects.

➔ Virtual member functions are resolved dynamically (at run-time). That is, the member function is selected dynamically (at run-time) based on the type of the object, not the type of the pointer/ reference to that object. This is called dynamic binding or late binding.

➔ Pure virtual functions are used in a base class that represents an abstract concept. A pure virtual function is a type of function, which has only a function declaration. Every derived class must include a function definition for each pure virtual function that is inherited from the base class if it has to be used to create an object.

➔ A class containing one or more pure virtual functions is called an abstract class. No objects of an abstract class can be created. The abstract class can only be used as a base for another class.

Session - 8

Data Structures Using C++ and Exception Handling

Welcome to the Session, **Data Structures Using C++ and Exception handling**.

This session introduces concept of data structures in C++ and exception handling mechanism. Further, the session describes about the terminate and unexpected functions.

At the end of this session, you will be able to:-

- ➔ Differentiate between pointers and references
- ➔ Operate with stacks
- ➔ Discuss linked lists
- ➔ Define queues
- ➔ Discuss the need of exception handling
- ➔ Write simple error handling routines
- ➔ Explain the function terminate()
- ➔ Explain the function unexpected()

8.1 Introduction to References

A **reference** is an alias for another variable. For example, in a family, a person is to be defined, we can define a reference to this as `person_name`. This means that the reference `person_name` can be used to point to that person in the family. References prove to be simple alternative to pointers while processing large structures. They are also useful while designing classes. Note that references are not pointers.

8.2 Defining References

In C++ the symbol '&' has an additional meaning. Apart from accessing the address of variables, it is used to define references. The following code fragment defines `add` as an alternative for the variable `sum`.

```
int sum;
int &add = sum ; // makes add as an alias for add
```

After defining a reference to `sum`, if somewhere in the program the variable `sum` is assigned a value of say 20, then the reference `add` automatically gets the value assigned to the variable `sum`.

Example 1 shows how the changes made to a variable are reflected in the reference and vice-versa.

Example 1:

```
// Use of reference
#include<iostream.h>
main (void)
{
int x=100;
int &y=x;
cout<<"x= "<<x<<endl;
cout<<"y= "<<y<<endl;
y++;                    //increment the value of reference
cout<<"x= "<<x<<endl; //incremented value of x
cout<<"y= "<<y<<endl; //incremented value of reference
}
```

Output:

```
x= 100
y= 100
x= 101
y= 101
```

As seen in the program, the symbol '`&`' is not used as an address operator. In fact, it is used as a part of the type identifier. Just like `int*` in a declaration means that a pointer to an integer variable is being defined, `int&` means that a reference to an integer is being defined. A reference declaration refers to the same value and memory location as that of the original variable.

Note: While using a reference parameter in a function the address (not the value) of an argument is automatically passed to the function. Operations on the reference parameter are automatically dereferenced.

If the original variable is deleted from memory then what happens to the reference?

8.3 Comparing Pointers and References

A reference is generally used to obtain access to a valid object. Therefore, the concept of "null reference" is meaningless. Whereas, in C++, the concept of "null pointers" is common. Programming with pointers in complex data structures often leads to programming errors. A reference is bound to a specific variable and it is initialized either as part of a declaration or in a constructor. The restrictions imposed on references are:

- ➔ A reference cannot be used for a reference variable
- ➔ An array of references cannot be defined
- ➔ A pointer to reference cannot be created

8.4 What is a Data Structure?

Programmers of high-level languages often take for granted the language they use. Features of a high-level language like loops and built-in data structures are appreciated only when a language does not have them. Many languages provide built-in data structures like a one or two-dimensional array and structures in C. What actually is a data structure? A data structure can be defined as a collection of data elements, whose organization is characterized by the functions accessing them. For instance, an array element is accessed using subscripts, the internal workings of which are transparent to the programmer. For a user-defined data structure, the programmer has to define the functions to access. Built-in data structures like arrays and structures have been covered in great detail. The next section describes using an array for implementing stacks.

8.5 Stacks

A stack is a data structure in which elements are added and removed from the same end. It is a structure, which works on the LIFO principle, that is, Last In First Out. At the logical level, a stack is an ordered set of elements. Since, removal and addition can take place only at the top of the stack one at a time, the last element added is always the first element to be removed, hence, LIFO. Its like filing papers in a box file. The first paper added to the file is always the last one in the file. The last one to be added is always on the top.

8.5.1 Operations on a Stack

Adding and removing elements to a stack is traditionally called as PUSH and POP, respectively. A push adds an item on the top of the stack and pop removes an item from the top of the stack. Two functions can be defined in C++, to do just the same. The push function receives the item to be pushed as an argument. The pop function removes an item and returns it back.

8.5.2 Implementing Stacks Using Arrays

A stack can be implemented using an array or a linked list. All the elements in an array are of the same data type and therefore, ideal for a stack. Elements of the stack are placed in the array as when they come in a sequential order. A subscript is used to access the top element. The first element in the stack goes in the 0th position of the array, the second element in position 1 of the array and so on. The subscript for the top element is kept floating from the position 0 to the number of the elements added, that is, it grows away from the position 0. When elements are removed, the subscript for the top element moves nearer to the position 0.

8.5.3 Defining a Class for Stacks

A stack can be defined using a class. The data members will be the array to hold the stack and a subscript to the top element of the stack. The methods will be functions to implement the push and pop operations and one more to initialize the data members. The class modeling a stack to hold integer values is given in Example 2.

Example 2:

```
const int MAX = 100;          // MAXIMUM ELEMENTS IN THE STACK.
class Stack
{
private:
int stack [MAX];          // ARRAY TO HOLD A STACK.
int top;                  // SUBSCRIPT FOR THE TOP ELEMENT.
public:
Stack(void)               // CONSTRUCTOR.
{
top = 0;
}
int StackPush (int item)      // PUSH OPERATION.
{
```

```
if(top<MAX)
{
stack[top++]=item;
return(0);              //RETURN 0 ON SUCCESS
}

else                 //ELSE -1 IF STACK FULL
return(-1);
}

int StackPop(int &item)     //POP OPERATION.
{

if(top>0)
{
item=stack[--top];
return(0);              //RETURN 0 ON SUCCESS
}
else
return(-1);             //ELSE -1 IF STACK EMPTY

}
};
```

The constructor initializes the subscript for the top element. The functions to implement the push and pop operations are fairly simple. `StackPush()` receives an element to be pushed as an argument, stores it in the position given by top and increments `top` for the next element. `Stack-Pop()` decrements `top` and return the element at `top`.

Given is a program, which uses this class to simulate the operations of a stack. Example 3 illustrates implementing stack programs.

Example 3:

```
#include<iostream.h>


void main(void)
{
Stack st; // Stack class from example 2
int num;
while(1) // ACCEPT NUMBERS TILL A ZERO.
{
cout << " enter a number (0 to quit): ";
cin >> num;
if(num)
{
if(st.StackPush(num)==-1)
{
cout <<"\nStack Full.\n";
break;
}
}
else
break;
}
while(1)                    // PRINT THE NUMBERS.
{
if(st.StackPop(num) == -1)
{
cout << "\n\nStack empty. ";
break;
}
else
cout << "\nNumber: " << num;
}
}
```

This is a primitive example, not overly exhaustive, but representative, of the usage of a stack. Also, this example does not use the push and pop operations intermittently, although they can be. A few numbers are accepted and pushed on the stack using a loop. Another loop, then, pops the element one at a time and prints them.

8.5.4 Applications of Stacks

A stack is an appropriate data structure on which information can be stored and then later retrieved in the reverse order. A very common use of stacks is in programs. Arguments are passed or returned to or from functions using a stack. The calling function pushes an argument on the stack and the called function pops it back from the stack. Thus the stack acts as a common area using which both the calling and the called functions communicate with each other. Not only the arguments, even the address of the calling function is pushed on the stack, because after the called function is over this is the address at which the execution has to continue. All this is transparent to the programmer of a high-level language.

8.6 Linked Lists

Elements in an array are arranged in a sequential order. That is, the physical arrangement of the elements is same as their logical ordering. All the elements of an array are stored in consecutive memory locations. Therefore, they can be accessed using subscripts. Whereas, in a linked representation, the logical order of the elements is not necessarily the same as that of its physical arrangement. A linked list always starts with a pointer to the first element and every element contains a pointer to the next element also called as the next pointer. The pointer to the first element is the head of the list, using which, the remaining elements in the list can be located. Its like an organized treasure hunt. The starting point gives the clue to the next point, on reaching which one gets the clue to the next and so on till the hunt ends in its last destination. Therefore, a linked list can be defined as a collection of elements, also called as nodes, each containing the data item to be stored and a pointer (or link) to the next node.

A node generally consists of two parts. First is the data part. This field can be a basic or a derived data type. This is the part, which contains the actual information to be stored in the list. The second part contains the link to the next node. This is a pointer variable. Figure 8.1 shows this.

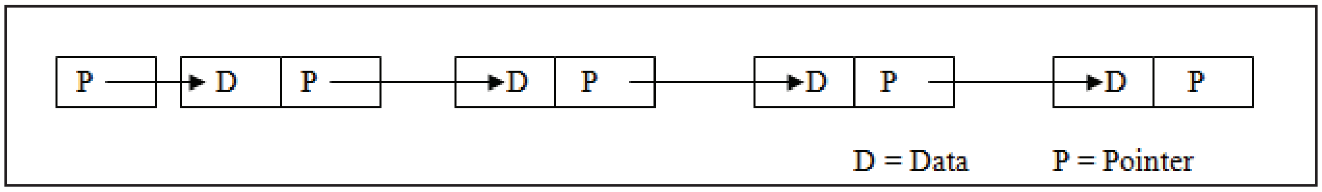


Figure 8.1: Node and Links

8.6.1 Operations on a Linked List

Built-in data structures like arrays are very easy to handle. Linked lists, on the other hand, have to be created, operated upon and disposed off by the program. Creating a linked list is done dynamically, that is, as and when some data is to be stored, a new node is created and added to the list. Whenever, data in a particular node is not required, the node can be removed. Once a linked list is ready, some manipulations on the data like traversing and retrieving the data from the proper node is done.

8.6.2 Defining a Class for the Linked List

A class can be used to define a linked list. A list consists of several nodes. So the data members can be a variable, of basic or user-defined data type, containing the information to be stored in the list and a pointer to the next node. These variables, do not form the data members of the list. Since, the node will be allocated dynamically, the class will have a pointer to a structure containing these variables rather than the structure variable itself. Using this pointer the nodes of the list can be accessed.

Given is a class which models a linked list. This class also contain function members for inserting a node in the list, deleting a node from the list, printing the list and a constructor function to initialize the pointer to the first node, which as explained earlier is the sole data member of the class.

Example 4:

```
struct Node                     // REPRESENTS A NODE.
{
char *string;                   // INFORMATION FIELD.
Node * next;               // POINTER TO NEXT NODE.
};
class List                      // MODELS A LINKED LIST.
{
private:
Node *head;                // POINTER TO THE FIRST NODE
public:

List ();                   // CONSTRUCTOR.
int ListInsert (char *str);
int ListDelete (char *str);
void ListPrint (void);
};
```

The structure defined earlier contains the data members of the class. Since memory is allocated dynamically, a variable of this structure type is not declared in the class. A pointer to this structure is declared as the sole data member of the class. The information field in the list is a character pointer.

8.6.3 Implementing the Linked List Methods

In the Linked list method, the class has only one member data item: the pointer to start of the list. When the list is first created, the constructor initializes the pointer to NULL. The NULL constant is defined in the `mem.h` header file, which is included in the `iostream.h` file. The constructor function has to just initialize the data member head to NULL, so that it can be determined if the list is initially empty or not. This is required at the time of creating the list.

```
List::List()                // CONSTRUCTOR
{
head=NULL;
}
```

➔ **Creating a Linked List**

Creating a node means, allocating memory for the node, copying the data in it and inserting the newly created node to the list. All this is done by the `ListInsert()` function. Some applications do not require the list to be maintained in any order. In which case, the new node can be appended to the end of the list, that is, the new node becomes the last node. Otherwise the new node can be put in the beginning of the list, which means the new node will become the first in the list and first becomes the second, the second becomes the third and so on. Given is an implementation of both these approaches separately.

Example 5:

```
//   NEW NODE PUT IN BEGINNING OF THE LIST.

List::List() // constructor
{
 head=NULL;
}

int List::ListInsert(char *newstr)
{
Node *n;
if ((n=new Node)==NULL)    // ALLOCATE A NEW NODE.
{
cout << "\nOut of memory";
return(-1);


}
strcpy(n->string, newstr); // STORE THE INFO.
n->next = head;              // NEW NODE COMES
head = n;                  // BEFORE head
return (0);
}
```

This implementation of the `Listinsert()` function puts the new node in the beginning of the list. First it allocates memory to the node and stores the information from `newstr` in the string data member of the node. The `next` data member is made to point to the first node in the list and the address of this new node is assigned to `head`. The same function is rewritten, but uses a different approach. It puts the new node after the end of the list. Refer to Example 6.

➔ Adding a Node

Example 6:

```
//   NEW NODE APPENDED TO THE LIST.


int List:: ListInsert (char *newstr)
{
Node *n;
static Node * store;
if((n = new Node) == NULL)         // ALLOCATE A NEW NODE.
{
cout << "\nOut of memory";
return(-1);
}
                           // MEMORY FOR INFO.


if((n->string = new char[strlen(newstr) + 1]) == NULL)
{
```

```
cout << "\nOut of memory";
return(-1);
}
strcpy(n->string, newstr);      // STORE THE INFO.
n->next = NULL;
if(head == NULL)
{
head = store = n;
}
else
{
store->next = n;
store = store->next;
}
return(0);
}
```

This function defines a static Node pointer, which stores the address of the last node allocated and stored in the list, over a number of calls. The next pointer of the last node allocated is made to point to the newly allocated node and becomes the last node allocated. Thus, it adds new nodes to the end of the list.

Some applications, however, require the linked list to be maintained in a sorted order. Initially, head points to the first node in the list. If the next information to be stored in less than the first in the list, the newly created node is placed at the beginning of the list, else, it is placed after the first node. Later, if information of a new node is less than the second, but greater than the first, the new node will be inserted in between the two. Rearranging the pointers of the first and the new node does this. The function `ListInsert()` given in Example 7 maintains a linked list in a sorted order.

Example 7:

```
//   MAINTAINS A LINKED LIST IN SORTED ORDER.
. . .
int List::ListInsert(char *newstr)
{
Node *n, *ptr;
if((n = new Node) == NULL)          // ALLOCATE NEW NODE.
```

```
{
cout << "\nOut of memory";
return(-1);
}
                            // MEMORY FOR INFO.

if((n->string = new char[strlen(newstr) +1]) == NULL)
{
cout < "\nOut of memory";
return(-1)'
}

strcpy(n->string, newstr);      // STORE THE INFO.

n->next = NULL;

if(head == NULL)                // IF EMPTY LIST.

{
head = n;
return (0);
}

                            // IF NEW STRING < FIRST STRING.

if(strcmp(n->string, head->string) <=0)
{
n->next = head;                 // INSERT IN BEGINNING.
head = n;
return (0);
}
ptr=head;                    // SCAN IF IT LIES IN
                        // BETWEEN TWO NODES OR
                        // IS IT THE LAST NODE ?
while(ptr->next!=NULL&&strcmp(ptr->next->string, n->string) <=0)

ptr=ptr->next;

ptr->next=n;

}
. . .
```

The function given here allocates a new node and inserts it in the linked list. There are four cases here.

- The list may be empty, in which case the new node becomes the first.
- The list is not empty, but the information in the new node is less than that of the first node, in which case the new node is put before the first one.
- The new node lies in between any two nodes. The while loop in the function attempts to find out these two nodes. This loop continues till the information in the new node is less than the information in any of the nodes searched sequentially. It stops when information in any of the existing nodes becomes greater than that of the new node is to be inserted.
- The last case is when the while loop terminates on reaching the end of the list. That is, the information in the new node is greater than that of all the nodes. In this case, the new node should be inserted in the list after the last node, which again is taken care of by the last two statements.

These are the different approaches for creating a linked list. The choice depends on the application itself. There are other operations that can be performed on a linked list. For instance, an application would like the information in the list to be printed. The function given illustrates the traversal of a linked list.

➔ **Printing a Linked List** – Example 8 illustrates how to print a linked list.

Example 8:

```
void List:: ListPrint(void)
{
Node *curr = head;
while(curr !=NULL)
{
cout << " [" << curr->string << "]\n";
curr = curr->next;
}
}
```

The function starts with head and uses a while loop for traversing the list. For each of the nodes, it prints the information and changes the `curr` pointer to print to the next node. This is an iterative method of traversing linked lists. Recursive data structures, like linked lists and trees, are more appropriate with recursive algorithms. The program, in Example 9, depicts a recursive algorithm to print the list.

Example 9:

```
//   PRINTING THE LIST RECURSIVELY.

void List:: ListPrint (void)
{
ListPrintRecur(head);
}

void List:: ListPrintRecur(Node *curr)
{
cout << curr->string << "\n";
if(curr->next != NULL)
ListPrintRecur(curr->next);
}
```

The pointer to the node whose information is to be printed is passed as an argument to the `ListPrintRecur()` function. Initially, it is head, subsequently it is next to the pointer passed to it. Hence, it travels till the end, printing the information from all the nodes on the way.

Therefore, `ListPrint()` is invoked without any parameters and this function invokes `ListPrintRecur()` `ListPrintRecur()` will be a private function invoked by `ListPrint()`.

➔ Deleting a Node from the Linked List

Yet another operation on a linked list is the deletion of a particular node. To delete a node at the end of the list, just traverse to the second last node in the list and put a null value in its next data member. If the node to be deleted is the first in the list, then head is made to point to the second node. After the node to be deleted is de-linked from the list, memory allocated to the node and its data members, if any, should be released. Some applications, also maintain another list of discarded nodes, so that the next addition to the list can be from this discarded list, rather than allocating a new.

The node prior to the required one is made to point to the node after it. Therefore, the node in the middle, which needs to be deleted, is de-linked from the linked list.

Example 10:

```
// DELETING A NODE FROM THE LIST.
int List::ListDelete(char *delstr)
{
Node *cur, *prev;
cur = prev = head;
while(cur)
{
if(strcmp(cur->string, delstr) > 0)      // STRING
return (-1);                  // NOT FOUND IN SORTED LIST.
                             // LIST

else if(strcmp(cur->string, delstr) == 0)
{                            // FOUND.
if(cur == head)               // IF IS FIRST NODE.
{
head = cur->next;             // RESET head.
delete cur->string;
delete cur;
return(0);
}
else
{
prev->next = cur->next;
delete cur->string;
delete cur;
return (0);
}
}
else                          // SEARCH FURTHER.
{
prev = cur;

cur = cur->next;
}
}
return (0);
}
```

The `ListDelete()` function deletes a node from the linked list, which is maintained in a sorted order. The function receives a character string `delstr`, as its argument. It first scans the list for a node with the matching string. If found it rearranges the pointer of its previous node to point to its next node and then it releases the memory that was occupied by the middle node.

➔ **Freeing the Linked List**

Some applications create a linked list only for a particular task and then it has to dispose off the entire list, so that memory occupied by it can be released. The function given released the memory occupied by the linked list.

It starts from head and iterates for all the nodes as shown in Example 11.

Example 11:

```
//   FREE THE LINK LIST.
void List::ListFree(void)
{
while(head!=NULL)
{
Node *temp=head;
head=temp->next;
delete temp->string;
delete temp;
}
```

➔ **Reversing a Linked List**

An interesting exercise on a linked list is reversing it in situ, that is without using another list or any other memory. Reversing the list means that if the list is in ascending order, then make it in the descending order. The first node becomes the last, the second node becomes second last and so on, till the last node becomes the first. The function to do the same is shown in Example 12. It uses an iterative algorithm to reverse the list.

Example 12:

```
//   REVERSING THE LIST IN SITU.
void List::ListReverse(void)
{
Node *last=NULL, *curr=head, *nxt;
while (curr!=NULL)
{
nxt=curr->next;
curr->next=last;
last=curr;
curr=nxt;
}
head=last;
}
```

8.6.4 An Example for the Linked List

An example to use the class and its member functions is given. It accepts several strings and stores them in the list in ascending order. There is a menu to select the operations on the list, which means that inserting a new node and deleting an existing node can be done intermittently. Insert the class list and member functions in `LINKLIST.CPP` before running.

Example 13:

```
//   LINKEDLIST.CPP: ILLUSTRATES THE USE OF A LINK LIST.

#include <string.h>
#include <conio.h> // FOR getch() AND clrscr().
#include <iostream.h>

class List          // MODELS A LINKED LIST.
{
private:
Node *head;         // POINTER TO THE FIRST NODE.

public:
List();             // CONSTRUCTOR.
int ListInsert(char *str);
int ListDelete(char *str);
void ListPrint(void);
void ListFree(void);
void ListReverse(void)'
};
// Include the definitions of the member functions of this
// class, as defined earlier. Also include the definition of // Node Structure, before executing the program.

void insert_element(List &Lobj)
{
char str[101];
clrscr();
cout <<”\n\t\tEnter a string: “;
cin.getline(str, 100);
Lobj.ListInsert(str);
}

void delete_element (List &Lobj)
{
char str[101];
clrscr();
cout << “\n\t\tEnter string to delete: “;
cin.getline(str, 100);

if(Lobj.ListDelete(str) == -1)
{
cout << “n\tString not found. Press any key …”;
if (getch() == 0)
getch();
}
}

void print_list(List &Lobj)
{
clrscr();
Lobj.ListPrint();
cout << “\n\n\t\tPress any key…”;
if(getch() == 0)
getch();
}
```

```
void main(void)
{
int option;
Lobj = List();

do
{
clrscr();
cout << "\n\n\n\n\n";
cout <<"\n\n\t\t\tF1  >>>>    INSERT";
cout <<"\n\n\t\t\tF2  >>>>    DELETE";
cout <<"\n\n\t\t\tF3  >>>>    PRINT";
cout <<"\n\n\t\t\tESC >>>>    QUIT";
if ((option = getch())==0)      // TRAP FUNCTION KEYS.
option = getch();

switch(option)
{
case 59: insert_element(Lobj);    // F1.
     break;

case 60: delete_element(Lobj);    // F2.
     break;

case 61: print_list(Lobj);        // F3.
break;
}
}
while(option!=27);            //ESCAPE.
}
```

8.6.5 Variations of Linked Lists

Many variations are possible in the implementation of linked lists. These variations, sometimes, are dependent on the applications, whereas, sometimes variations in a linked list are advantageous. The examples given in the previous section are about a singly linked list. A singly linked list has pointers in only one direction. As seen, the entire list can be traversed using the head data member of the class. This traversal can be in only one direction. Given a node, it can retrieve only the next node.

One variation of linked list is the doubly linked list, which contains two pointers in every node – one points to the next node, like in a singly linked list, and the other points to the previous node. The next pointer of the last node and the previous pointer of the first node contain a null value. In doubly linked lists, given a node, it is possible to move in either direction. Doubly linked lists, however, have a distinct overhead in terms of the space required for the previous pointers of each of the node.

Yet another variation of a linked list is a circularly linked list. In circularly linked lists the next pointer of the last node does not contain a null value, instead it points to the first node in the list. Thus, the list has no end. A circular list can be a singly linked list or it can be a doubly linked list, in which case the previous pointer of the first node points to the last node and the next pointer of the last node points to the first node.

8.6.6 Advantages and Disadvantages of Linked Lists

The foremost advantage of using linked lists is dynamic storage. Memory can be allocated, as and when required. The limit to the number of nodes in a linked list is the computer's memory itself. With linked list, it is not necessary to make decisions about memory allocation in advance.

Changes like insertions and deletions, can be made in the middle of a linked list more easily than it can be in a contiguous array. Inserting an element into an array means shifting all the items one position down. Whereas, inserting or deleting an item in a linked list requires only rearranging a few pointers.

One big overhead of a linked list is the memory occupied by the pointers. Every node in a single liked list has one pointer and in doubly linked list, has two pointers. If the list contains nodes in which the memory space occupied by the data part is equal to the memory space occupied by the pointers, then the list will occupy twice as much memory as a contiguous array would.

The biggest disadvantage of a linked list is that random access is not possible. With contiguous storage any element can be referred to immediately using a subscript. With a linked list, it may be required to traverse till the end of the list sequentially to retrieve a particular node.

8.6.7 Implementing Stacks using Linked Lists

Listed here is a drawback that explains why we implement stacks using linked list:

> If arrays are used to implement stacks then the stack memory becomes static and limited. Arrays cannot grow dynamically which may be required by a linked list. This is because when we define an array, memory locations are reserved for the elements of the array.

Every linked list has a head using which the entire list can be traversed and in a linked list implementation of a stack, this head can also be used as the top element. The algorithm given for creating a linked list, can be used as a push node function. A pop node function will be required for getting the top element. An example for implementation of stacks using linked lists is not covered here and is left as an exercise.

8.7 Queues

For computer applications, we define a queue to be a list in which all additions are made at one end and all deletions are made from the other. Queues are First In, First Out (FIFO) form of lists, unlike stacks, which recall are Last in, First Out (LIFO) form of lists. Figure 8.2 shows the diagrammatic representation of a queue.

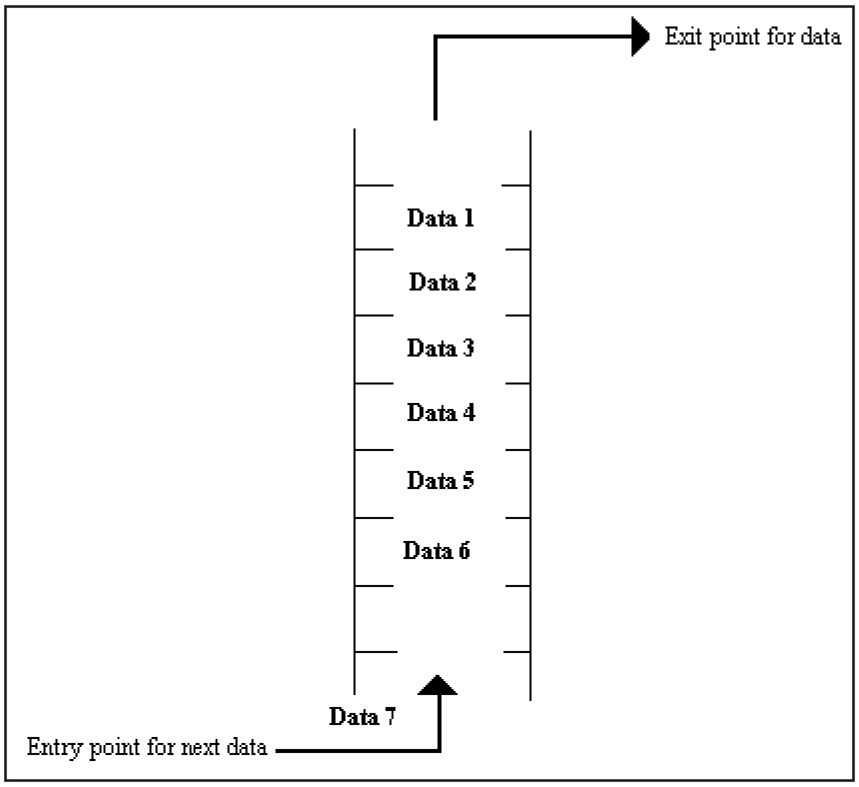


Figure 8.2: Queue (First In First Out)

8.7.1 Operations On a Queue

Besides operations like creating and initializing a queue, the two other main operations performed on a queue are adding an element to the rear end of the queue and removing an element from the front end of the queue. An add function will receive an item as its argument and add it to rear end of the queue, much like the push operation of the stack. A delete function will remove an item from front end of the queue and return it back, much like the pop operation of the stack. The first item on the queue is the first one to be removed and is also called as the head. The last item of the queue is after which a new element is added and is also called as the tail.

8.7.2 Implementing a Queue

➔ **Implementing A Queue Using Arrays**

Listed are some points, which are to be considered when, implementing a queue using arrays?

- Two subscripts are required to be maintained, one for the front end and other for the rear.
- One way is to keep the front of the queue always in the first physical location that is the `0th` position.
- An item can be added to the queue by increasing the rear end subscript.
- The subscript for the rear end will grow away from the `0th` position.
- Deleting an item will be from the front end, that is the `0th` position.

For example consider a queue of people moving ahead in a queue for, say, purchasing tickets, but it makes a poor choice as far as implementing it in computers is concerned. One way to circumvent this problem is to use a floating front. That is, as and when the items are added to the queue the rear end grows away from the 0th position and when an item is deleted the front end also grows away from the 0th position. Here, the remaining elements in the queue are not moved up one position after an item is deleted. Its like moving the ticket counter nearer to the second person, after the first one in the queue leaves and so on.

However, there is a problem in this approach. As the queue moves down the array, the space in the beginning of the array is discarded and never used again. Soon, the rear end and the front end of the queue will come to the physical end of the array, which is limited and cannot be extended. In same situations, however, if the queue becomes empty at any given point of time, that is the front and rear are the same, then they can be set to the 0th position once again. Such a situation is, however, circumstantial.

The space can be used efficiently, only if the queue itself can be made to float in the array. Once it reaches the end, it should wrap around to the beginning of the array, thus making it a circular array. In this way, as items are added and deleted from the queue, the front end will continuously chase the rear end. When the front and the rear ends come to the physical end of the array, they should wrap around to the physical start of the array. Thus, at different times the queue will occupy different parts of the array.

However, no matter how efficiently space is used, if the queue is implemented using array, it will have a limited capacity. Decision about such a limit is often made at the time of programming the queue; there is an upper limit on the number of elements that can be added. Implementation of queues using linked lists can solve this memory limit very efficiently.

➔ **Implementing A Queue Using Linked Lists**

Queues can also be implemented using linked lists. Linked queues are just as easy to handle as linked stacks. A linked implementation of queues is programmed just like explained in the section on linked lists. Two pointers are required – one for the front end and one for the rear end. A dynamically allocated queue can grow as far as the computer's memory limit. When a node is removed from the front end, memory occupied by it is released.

8.7.3 Defining a Class for a Queue

Defining a class for queues is same as defining a class for linked list. In a queue, where add and delete operations are done at two different ends, the class will contain two pointers. The two methods in this class will be one to add a node and another to delete a node. The implementation of a queue using a class is given in Example 14.

Example 14:

```
struct Node        // REPRESENTS A NODE.
{
char *message;          // INFO FIELD.
Node *next;        // POINTER TO NEXT NODE.
};
class Queue    // IMPLEMENTS A QUEUE.
{
private:

Node *front, *rear;     // TWO POINTERS FOR TWO ENDS.

public:
Queue(void);       // CONSTRUCTOR.
int QueuePut(char *msg);
int QueueGet(char *msg);
};
```

The class contains two pointers to the two ends of the queue. The constructor function initializes these two pointers to a null value. The `QueuePut()` function adds an element to the rear end of the queue. It receives the information, in this case, a character string, as its argument. The `QueueGet()` function removes a node from the front end of the queue. It receives a pointer to a buffer, where the function copies the information and releases the memory occupied by it. The buffer should be large enough to hold the information. Both these functions return `0` on success or `-1` on error, which can be an empty queue, in the `QueueGet()` function or no free memory space in the `QueuePut()` function.

8.7.4 Implementing the Queue Methods

➔ **The Constructor Function**

The constructor function for this class initializes the pointers for the two ends of the queue.

Example 15:

```
Queue::Queue(void)
{
front = rear = NULL;
};
```

➔ **Adding An Element To The Queue**

The `QueuePut()` function adds an element to the rear end of the queue. It receives the information, in this case, a character string, as its argument. This function has to allocate a new node for it and put at the end of the queue. An implementation of this function is given in Example 16.

Example 16:

```
int queue::QueuePut(char *msg)
{
Node *tmp;
tmp = new Node;      // ALLOCATE A NEW NODE.
if(!tmp)                 // NO MEMORY.
return(-1);
// MEMORY FOR INFO.
tmp->message = new char[strlen(msg) + 1];
strcpy(tmp->message, msg); // STORE THE INFO.
tmp->next = NULL;
if(front == NULL)        // IF QUEUE EMPTY.
front = rear = tmp;
else
{
rear->next = tmp;       // PUT IT AFTER rear
rear = tmp;
}
return (0);             // RETURN 0 FOR SUCCESS.
}
```

➔ **Deleting an Element From the Queue**

The `QueueGet()` function removes a node from the front end of the queue. It receives a pointer to a buffer, where the function copies the information and releases the memory occupied by it. The buffer should be large enough to hold the information. This implementation does not check if the buffer is large enough, nor there is any way for the program to find out how much memory is required. One way out for this is not to free the memory occupied by the character string, but return back the character pointer from the node and bank upon the user of this class to free the memory after its use is over. This is just a matter of personal preference or programming convenience. Given is an implementation for the `QueueGet()` function.

Example 17:

```
int Queue::QueueGet(char *msg)
{
Node *temp;
if (front == NULL)      // IF QUEUE EMPTY.
return (-1);      // SAVE THE INFO.
strcpy(msg, front->message);
temp = front;
front = front->next;        // DELINK THE NODE FROM QUEUE
delete temp->message;       // RELEASE MEMORY OF STRING
delete temp;            // AND OF THE NODE.
if (front == NULL)      // IF QUEUE BECOMES EMPTY.
rear = NULL;
return(0);                  // RETURN 0 ON SUCCESS.
}
```

8.7.5 Using the Queue

Given is a sample program, which simulates the use of a queue.

Example 18:

```
//   QUEUE.CPP: SIMULATES A MESSAGE QUEUE.

#include <string .h>
#include <iostream.h>

// Include the Class Queue definition and
// member function definitions in the
// program. Also include definition of
// structure Node.

void main (void)
{
Queue q;
char option;
char m[101];
do
{
cout << "\n\nAdd/Remove/Quit (A/R/Q) ? ";
cin >> option;
cin.get ();

switch (option)
{
case `A':
case `a': cout << "\nEnter message: ";
          cin.getline (m, 100);
          if (q.QueuePut (m) == -1)
          cout << "\nNo room on Queue";
          break;

case `R':
case `r': if(q.QueueGet(m) != -1)
          cout << m << "Removed From Queue";

          else

          cout << "\nQueue is empty. ";
          break;

case `Q':
case `q': return;      // LOOP ENDS.
}
}while (1);
}
```

8.8 Introduction to Exceptions

C++ provides a built in mechanism for handling errors known as exception handling. Run-time errors can be easily managed and handled using exception handling feature. An exception can be called as an error or an unexpected event. Exceptions can occur due to the following reasons.

Synchronous events (errors): These are the errors that occur in the natural flow of a program, like a divide by zero error. Errors that occur due the execution of the programs fall into this category.

Asynchronous events: These occur randomly like a mouse click, or a disk error. Errors that do not occur due to execution of programs fall into this category.

8.9 The Need for Exception Handling

Code written to handle exception is known as exception handler. The need for an exception handler arises because exceptions might cause unpredicted program termination. They may also result in abnormal behavior of the program. For example, consider a situation wherein data is fed to a program and the program performs a division operation on the data. If the user enters some data, which is accidentally divided by zero within the program, the program terminates abnormally. This might result in the loss of data previously entered. Then the process of reentering the data should begin again. At times, some critical errors may result in hardware problems.

Although simple C++ statements can be used to check, errors but these statements become redundant throughout the program. This may consume space and time. Such problems can be avoided by the use of exception handling.

8.10 Handling Exceptions

The approach, which must be followed, in order to take care of exceptions is as follows:

- ➔ Monitor the code, which is likely to generate exceptions.
- ➔ On detection of error, pass the information to take corrective measures.
- ➔ Take the remedial actions.

C++ provides three keywords, which are used in exception handling. These keywords are listed.

- ➔ **try**
- ➔ **throw**
- ➔ **catch**

The keyword `try` is used to monitor for exceptions. The keyword `throw` is used pass the error information in the form of parameters. The keyword `catch` encloses the code, which performs remedial action.

Pictorially this sequence can be represented as shown in figure 8.3.

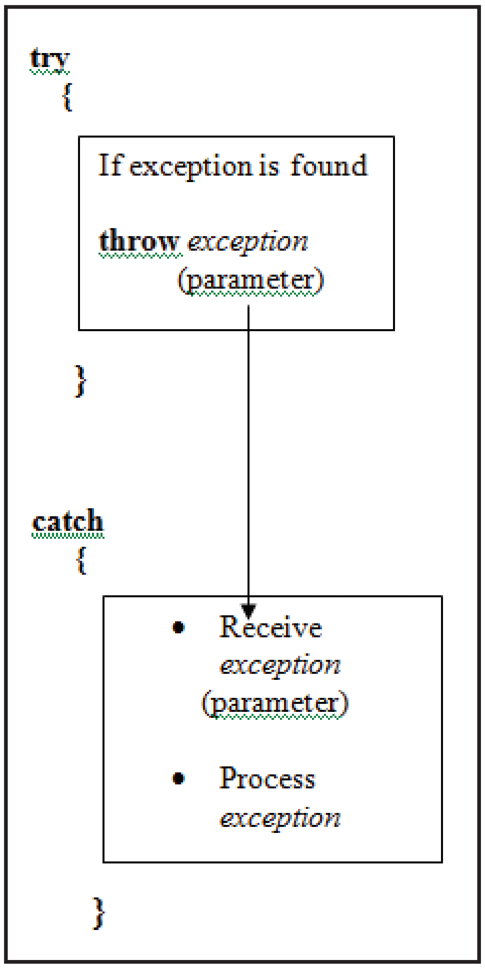


Figure 8.3: Exception Handling Syntax

A statement that can throw an exception must have been executed in the **try** block. A function, which is called from a **try** block, can also **throw** an exception. An exception, which is thrown, must be caught by a catch statement. A single throw can have multiple catch statements depending on the parameter passed by the throw. An outline of **try**, **throw**, and **catch** is shown.

```
try{


 //Try block statements
statement 1;
statement 2;
statement 3;
          .
          .
statement n;
throw parameter;
statement m;
statement ….
}
catch (parameter)
{
// Statements in the catch block
statement 1;
statement 2;
      .
      .
statement n;
}
```

As a simple example look at the program as given in Example 19. This program demonstrates a simple exception catching example.

Example 19:

```
#include<iostream.h>
main()
{
cout << "Demonstration start\n";

try
{
// The start of try block
cout<<"Program control within a try block\n";
throw 50; // Throwing the error
cout<< " This statement will not be executed";
cout<<"...because program control out of try\n";
}
catch (int n)
{
// The start of catch block
cout<<" Program control now in a catch block\n";
cout<< " Exception is caught the value of the parameter is";
cout<<n<<"\n";
}

cout<<" End of demonstration";

return 0;
}
```

Output:

```
Demonstration start
Program control within a try block
Program control now in catch block
Exception is caught the value of the parameter is 50
End of demonstration
```

Looking at the example, you can see that after the `throw` statement the rest of the statements in the `try` block are ignored and the program control goes into the `catch` block. The pending function in the main, is executed only after the `catch` block is processed. Sometimes an exception may indicate a catastrophic error and the program has to be terminated. Therefore, while programming a typical catch block may consist of the function `abort()`. The execution of the function `abort()` results in immediate termination of the program execution.

8.11 Exception Matching

The data type of exception thrown from a try block should match the data type, which is used to receive the exception parameter. If there is a data type mismatch then the program terminates abnormally. For example, if the program given in Example 19 is rewritten as shown in Example 20, then it terminates abnormally. In this example, the data type thrown does not match the data type specified in the `catch` block.

Example 20:

```
#include<iostream.h>
main()
{
cout << " Demonstration start\n";

try
{
// The start of try block
cout<<"Program control within a try block\n";
throw 50; // Throwing the error
cout<< " This statement will not be executed";
cout<<"...because program control out of try\n";
}
catch (double n) // Data type mismatch
{
// The start of catch block
cout<<" Program control now in a catch block\n";
cout<<" Exception is caught the value of the parameter is";
cout<<n<<"\n";
}

cout<<" End of demonstration";

return 0;
}
```

Output:

```
Demonstration start
Program control within a try block
Abnormal program termination
```

Exceptions can also be thrown from a statement, which is outside the `try` block. Example 21 demonstrates this feature.

Example 21:

```
#include<iostream.h>
#include<conio.h>

void outfunc(int i)
{
cout<<" Program control inside the function";
if(i==0)
{
cout<<" Test condition valid";
throw i;
}
}
main()
{
cout<<" Start of program execution - in main()";
try
{    // Start of the try block
cout<<" Program execution in the try block\n";
outfunc(0);
outfunc(1);
}

catch (int j)
{    // Start of catch block
cout<<" Exception detected and the value is: ";
cout<<j<<"\n";
}
cout<< " End of example";
return 0;
}
```

Output:

```
Start of program execution – in mail ()

Program execution in the try block

Program control inside the function

Test condition valid
```

Exception detected and the value is: 1

It is not necessary that a try block is always localized to the function `main()`. It can be used to monitor statements within a function. Along with the `try` block the `catch` block can also be localized to the function. The following example consists of a function which compromises of both the `try` and `catch` blocks.

Example 22:

```
#include<iostream.h>

#include<conio.h>

void test(int k)

{

cout<<"Program control inside the function";

cout<<"Try and catch inside the function"';

try

{

if(k)

throw k;

}

catch(int k)

{

cout<<"Program control inside catch";

cout<<"Exception caught and the value is"<<k;

}

}

main()

{
```

```
cout<<"Start of program execution - in main()";
test(1);
cout<<"\n";
test(2);
cout<<"\n";
cout<< " End of example";
return 0;
}
```

Output:

```
Start of program execution - in main()
Program control inside the function
Try and catch inside the function
Program control inside catch
Exception caught and the value is 1

Program control inside the function
Try and catch inside the function
Program control inside catch
Exception caught and the value is 2
End of example
```

8.12 Options in Catch Blocks

It is not necessary that a `catch` block in an exception handling mechanism should exhibit a one to one correspondence with a `try` block. Various other options of `catch` blocks are discussed in the sub sections.

8.12.1 Using Multiple Catch Blocks

A `try` block can be associated with multiple `catch` blocks. The program control passes to the appropriate `catch` block depending on the parameter passed by the `throw` statement in the `try` block. In such a case, care should be taken that all the `catch` blocks should be written to handle different types of exceptions. The program in Example 23 uses multiple `catch` blocks for one `try` block.

Example 23:

```
#include<iostream.h>
// Multiple CATCH blocks for one TRY block
void handler ( int value)
{
try{
if (value)
{
cout<<" Integer throw to proceed";
throw value;
}
else
{
cout<<"String throw to proceed";
throw " Value passed to handler function is string";
}
}
catch (char *strin)          //Catch block for zero value
{
cout<<" A string exception caught: ";
cout<<strin<<"\n";
}
catch (int d)        //Catch block for non zero value
{
cout<<" An integer exception caught\n";
cout<<" The value is ";
cout<<d<<"\n";
}
main ()
{
```

```
cout<<" Start of program execution in main";
handler(1);
handler(0);
cout<<"\n Program control back in main ------ now it ends";
return 0;
}
```

Output:

```
Start of program execution in main
Integer throw to proceed
An integer exception caught
The value is 1
String throw to proceed
A string exception caught: Value passed to handler function is string
Program control back in main ------ now it ends
```

8.12.2 One Catch and All Exceptions

A `catch` block can be written so that it is ready to handle all types of exceptions thrown to it. The simple method is to use the `catch(…)` form of definition for the catch. This is illustrated in Example 24.

Example 24:

```
#include<iostream.h>
// One CATCH block for multiple types of throw
void handler( int value)
{
try{
if(value)
{
cout<<" Integer throw to proceed";
throw value;
}
else
{
```

```
cout<<"String throw to proceed";
throw " Value passed to handler function is string";
}
}
catch(...)        //Catch block for zero value
{
cout<<" An exception caught: ";
cout<<"\n";
}
main()
{
cout<<" Start of program execution in main";
handler(1);
handler(0);
cout<<"\n Program control back in main ------ now it ends";
return 0;
}
```

Output:

```
Start of program execution in main
Integer throw to proceed
An exception caught
String throw to proceed
An exception caught
Program control back in main ------ now it ends
```

The `catch(…)` form of the exception handling can be used as the default `catch` block for exceptions which are not taken into account. For example, if a program is written to handle exceptions of the type integer and string and an exception of the type double is generated then the program may terminate abnormally. To avoid this, a `catch` block having the form `catch(…)` can be written so as to handle exceptions for which no `catch` block is defined. Thus, it will act as the default catch block.

8.13 Restricting Exceptions from Functions

As we have seen functions called from the try block can raise exceptions. As per the type of exception raised, restrictions are imposed on the functions. In addition, they can be defined as not raising any exception at all. The restriction can be imposed while defining the function. The general format while defining such functions would be as shown

```
ret-type function-name(argument list) throw(type-list)
```

As it can be seen the function definition is followed by the keyword `throw`. The type of exceptions that the function can give rise to is specified by the type-list. If the function tries to throw an exception, which is not, there in the type-list then abnormal program termination occurs. That is the function `unexpected()` will be called and consequently the function `abort()` is called leading to abnormal termination of the program. The program given is an example imposing restrictions on the type of exceptions that can be thrown by the function.

Example 25:

```
#include<iostream.h>

// This function should raise only exceptions of the type integer // and double

void rest-func( int num ) throw(int,double)
{
if (num==0) throw num;  // throwing integer
if(num==1) throw 999.99 // throwing double
}
main()
{
cout<<" Start of program\n";
try
{
rest-func(0);
}

catch(int p)
{
cout<<" Found an integer exception\n";
}
catch(double d)
{
cout<<" Found an exception of the type double\n";
}

cout<<"Ending main()";
return 0;
}
```

Output:

```
Start of program
Found an integer exception
Ending main()
```

If the `type-list` is kept empty then the function is not allowed to throw any exception.

8.14 Special Functions in Exception Processing

There are some special functions, which are called when an exception handling fails. Two of these have been discussed in the following sub sections.

8.14.1 The terminate() function

If a function raises an exception, which does not have a corresponding catch block, the function `terminate()` is invoked. This function by default calls the function `abort()`. The function `abort()` terminates the program immediately. It acts like an emergency stop.

8.14.2 The unexpected() function

Once an exception is thrown it should be caught by an appropriate catch block. This means that that the data type of the thrown parameter and the parameter in the catch block should match. If there is a data type mismatch then the function `unexpected()` is called which by default calls the function `abort()`. This results in immediate termination of program.

8.15 Check Your Progress

1. References are pointers.

(A)	True	(B)	False

2. The symbol '$' is used to define references.

(A)	True	(B)	False

3. A pointer to a reference cannot be created.

(A)	True	(B)	False

4. A change in the reference value is not reflected is the actual argument.

(A)	True	(B)	False

5. A stack works on the principle of_________________________.
6. The operations performed on a stack are PUSH and ___________.
7. In a __________________the elements are tied together using pointers.
8. In a linked list each element is known as____________.
9. Dynamic memory allocation is possible in arrays.

(A)	True	(B)	False

10. Null Pointers are used in the_______________ node in a linked list.
11. Nodes can be accessed in random order in a linked list.

(A)	True	(B)	False

12. A______________ works on the principle of FIFO.
13. The built in mechanism to handle errors in C++ is known as exception handling.

(A)	True	(B)	False

14. The throw statement is embedded in the ________ block.
15. The statements following the throw statement in a try block are executed.

(A)	True	(B)	False

16. There cannot be more than one catch block for a try block.

(A)	True	(B)	False

17. The function _______________ results in immediate termination of the program.

18. A try block has to be localized to the function `main()`.

(A)	True	(B)	False

19. The __________ form of catch block can be used to handle all type of exceptions.

20. A function called from the try block cannot raise exceptions.

(A)	True	(B)	False

21. Restrictions can be imposed on the type of exception raised by a function.

(A)	True	(B)	False

22. When a corresponding catch block is not found __________ function is invoked.

23. The function terminate() by default calls the function ___________.

8.16 Do It Yourself

1. Write a program wherein give the values 10 and 20 to variables x and y respectively. Swap the values of two variables using references.

2. Write a program to implement a stack using linked list.

3. Write a program to implement a circular linked list.

   (Hint: A circular linked does not have a pointer pointing to the first node. Instead the pointer in the last node points to the first node)

4. Write a C++ program in which a function called from the try block is restricted to raise exceptions of the type integer.

5. Write a C++ program incorporating a exception handling mechanism to segregate the type of data entered by the user. The program must also segregate all alphanumeric and special characters into one category.

6. Write a C++ program, which should take care of the exception, generated while reading a file, which does not exist.

Summary

- Alternative name for a previously defined variable is known as reference. A reference is not a pointer.
- A null reference cannot be created.
- Stacks work on the principle of LIFO (Last In First Out).
- A linked list always starts with a pointer to the first element and every element contains a pointer to the next element.
- A queue is a data structure that works on the principle of FIFO (First In First Out).
- C++ provides a built in mechanism for handling errors known as exception handling.
- C++ provides three keywords, which are used in exception handling. These keywords are try `throw` and `catch`.
- The keyword `try` is used to monitor for exceptions. The keyword `throw` is used pass the error information in the form of parameters. The keyword `catch` encloses the code, which performs remedial action.
- A statement that can throw an exception must have been executed in the try block.
- The data type of exception thrown from a try block should match the data type, which is used to receive the exception parameter.
- Exceptions can also be thrown from a function called from the try block.
- A try block can be associated with multiple catch blocks.
- A catch block can be written in a manner such that it is ready to handle all types of exceptions thrown to it.