PDCA - 03

VARDHAMAN MAHAVEER OPEN UNIVERSITY, KOTA

Post Graduate Diploma in Computer Application (PGDCA)

OOPS Programming with C++ and JAVA

Course Development Committee

Chaiman

Prof. (Dr.) Naresh Dadhich
Vice-Chancellor
Vardhaman Mahaveer Open University, Kota

Convener / Coordinator

Prof. (Dr.) D.S. Chauhan
Department of Mathematics
University of Rajasthan, Jaipur

Member Secretary / Coordinator

Sh. Rakesh Sharma
Assistant Professor (Computer Application)
V.M. Open University, Kota

Members

1. Prof. (Dr.) S.C. Jain
   Engineering College, Ajmer
2. Prof. (Dr.) M.C. Govil
   M.N.I.T., Jaipur
3. Dr. (Mrs.) Madhavi Sinha
   A.I.M. & A.C.T., Jaipur

Editing and Course Writing

Editor

Dr. (Mrs.) Madhavi Sinha
Reader (Computer Science Department)
A.I.M. & A.C.T. Jaipur

Writers

1. Prof. (Dr.) S.C. Jain
   Department of Computer Engineering
   Engineering College, Ajmer
2. Sh. N.K. Joshi
   HOD Computer Application Department
   MIMT Kota
3. Sh. Manoj Singh
   Lecturer (Computer Engineering)
   Modi Engineering College, Kota
4. Sh. D.K. Gupta
   Modi Engineering College,
   Kota
5. Sh. Arvind Sharma
   Lecturer (Computer Sc.) and
   Software Developer
   DAV Sr. Secondary School,
   Kota

Course Supervision and Production

Director (Academic)

Prof. (Dr.) Anam Jaitly
Vardhaman Mahaveer Open University,
Kota

Director (Material Production & Distribution)

Prof. (Dr.) P.K. Sharma
Vardhaman Mahaveer Open University,
Kota

Production July 2007

Printed and published on behalf of V.M. Open University, Kota by Director (Academic).

INDEX

UNIT - 1

Object Oriented Programming Concepts

Structure of the Unit

1.0 Objective

After studying this unit, you will learn following:

- Concept of Object Oriented Programming
- Object & Class
- Advantages of Object Oriented Programming

1.1 Programming Methodologies

Since the invention of the computer, many programming approaches have been tried. These include techniques such as modular programming, top-down programming, bottom-up programming and structured programming. The primary motivation in each has been the concern to handle the increasing complexity of programs that are reliable and maintainable. These techniques have become popular among programmers over the last two decades.

With the advent of languages such as C, structured programming became very popular and was the main technique of the 1980s. Structured programming was a powerful tool that enabled programmers to write moderately complex programs fairly easily. However, as the programs grew larger, even the structured approach failed to show the desired results in terms of bug-free, easy-to-maintain, and reusable programs.

Before you get into Object Oriented Programming (OOP) , take a look at conventional procedure-oriented programming in a language such as C. Using the procedure-oriented approach, you view a problem as a sequence of things to do. You organize the related data items into C structs and write the necessary functions (procedures) to manipulate the data and, in the process, complete the sequence of tasks that solve your problem. Although the data may be organized into structures, the primary focus is on the functions. Each C function transforms data in some way. For example, you may have a function that calculates the average value of a set of numbers, another that computes the square root, and one that prints a string. You do not have to look far to find examples of this kind of programming–C function libraries are implemented this way. Each function in a library performs a well-defined operation on its input arguments and returns the transformed data as a return value. Arguments may be pointers to data that the function directly alters or the function may have the effect of displaying graphics on a video monitor.

Object-Oriented-Programming (OOP) is an approach to program organization and development that attempts to eliminate some of the pitfalls of conventional programming methods by incorporating the best of structured programming features with several powerful new concepts. It is a new way of organizing and developing programs and has nothing to do with any particular language. However, not all languages are suitable to implement the OOP concepts easily.

In the procedure-oriented programming system, procedures are dissociated from data and are not a part of it. Instead, they receive structure variables or their addresses and work upon them. The code design is centered around procedures. While this may sound obvious, this programming pattern has its drawbacks.

The drawback with this programming pattern is that the data is not secure. It can be manipulated by any procedure. Associated functions that were designed by the library programmer do not have the exclusive rights to work upon the data. They are not a part of the structure definition itself. Let us see why this is a problem.

1.2 Evolution of OOPs

Suppose the library programmer has defined a structure and its associated functions as described above. Further, in order to perfect his/his creation, he/she has rigorously tested the associated functions by calling them from small test applications. Despite his/her best efforts, he/she cannot be sure that an application that uses the structure will be bug free. The application program might modify the structure variables, not by the associated function he/she has created, but by some code inadvertently written in the application program itself. Compilers that implement the procedure-oriented programming system do not prevent unauthorized functions from accessing/manipulating structure variables.

Let us look at the situation from the application programmer's point of view. Consider an application of around 25,000 lines (quite common in the real programming world), in which variables of this structure have been used quite extensively. During testing, it is found that the date being represented by one of these variables has become 29th February 1999! The faulty piece of code that is causing this bug can be anywhere in the program. Therefore, debugging will involve a visual inspection of the entire code (of 25000 lines!) And will not be limited to the associated functions only.

The situation becomes especially grave if the execution of the code that is likely to corrupt the data is conditional. For example,

```
if (<some condition>)
        d.m++;          // d is a variable of date structure... d.m may
                        // become 13!
```

The condition under which the bug-infested code executes may not arise during testing. While distributing his/her application, the application programmer cannot be sure that it would run successfully. Moreover, every new piece of code that accesses structure variables will have to be visually inspected and tested again to ensure that it does not corrupt the members of the structure. After all, compilers that implement procedure-oriented programming systems do not prevent unauthorized functions from accessing/manipulating structure variables.

Let us think of a compiler that enables the library programmer to assign exclusive rights to the associated functions for accessing the data members of the corresponding structure. If this happens, then our problem is solved. If a function which is not one of the intended associated functions accesses the data members of a structure variable, a compile-time error will result. To ensure a successful compile of his/her application code, the application programmer will be forced to remove those statements that access data members of structure variables. Thus, the application that arises out of a successful compile will be the outcome of a piece of code that is free of any unauthorized access to the data members of the structure variables used therein. Consequently, if a run-time error arises, attention can be focussed on the associated library functions.

It is the lack of data security of procedure-oriented programming systems that led to object-oriented programming systems (OOPS). This new system of programming is the subject of our next discussion.

1.3 Definition of Object Oriented Programming

1.3.1 Object-Oriented Programming Systems

In OOPS, we try to model real-world objects. But, where are real-world objects? Most real-

world objects have internal parts and interfaces that enables us to operates them. These interfaces perfectly manipulate the internal parts of the objects. They also have the exclusive rights to do so.

Let us understand this concept with the help of an example. Take the case of a simple LCD projector (a real-world object). It has a fan and a lamp. There are two switches– one to operate the fan and the other to operate the lamp. However, the operation of these switches is necessarily governed by rules. If the lamp is switched on, the fan should automatically switch itself on. Otherwise, the LCD projector will get damaged. For the same reason, the lamp should automatically get switched off if the fan is switched off. In order to cater to these conditions, the switches are suitable linked with each other. The interface to the LCD projector is perfect. Further, this interface has the exclusive rights to operate the lamp and fan.

This, in fact, is a common characteristic of all real-world objects. If a perfect interface is required to work on an object, it will also have exclusive rights to do so.

Coming back to C++ programming, we notice a resemblance between the observed behavior of the LCD projector and the desired behavior of the structure's variables. In OOPS, with the help of a new programming construct and new keywords, associated functions of the date structure can be given exclusive rights to work upon its variables. In other words, all other pieces of code can be prevented from accessing the data members of the variables of this structure.

Compilers that implement OOPS enable data security by diligently enforcing this prohibition. They do this by throwing compile-time errors pieces of code that violate the prohibition. This prohibition, if enforced, will make structure variables behave like real-world objects. Associated functions that are defined to perfectly manipulate structure variables can be given exclusive rights to do so.

There is still another characteristic of real-world objects–a guaranteed initialization of data. After all, when you connect the LCD projector to the mains, it does not start up in an invalid state (fan off and lamp on). By default, either both the lamp and the fan are off or both are on. Users of the LCD projector need not do this explicitly. The same characteristic is found in all real-world objects.

Programming languages that implement OOPS enable library programmers to incorporate this characteristic of real-world objects into structure variables. Library programmers can ensure a guaranteed initialization of data members of structure variables to the desired values. For this, application programmers do not need to write code explicitly.

Two more features are incidental too OOPS. They are:

- Inheritance
- Polymorphism

Inheritance allows one structure to inherit the characteristics of an existing structure.

As we know from our knowledge of structures, a variable of the new structure will contain data members mentioned in the new structure's definition. However, because of inheritance, it will also contain data members mentioned in the existing structure's definition from which the new structure has inherited.

Further, associated functions of the new structure can work upon a variable of the new structure. For this, the address/name of a variable of the new structure is passed to the associated functions of the new structure. Again, as a result of inheritance, associated functions of the existing struc-

ture from which the new structure has inherited will also be able to work upon a variable of the new structure. For this, the address/name of a variable of the new structure is passed to the associated functions of the existing structure.

In inheritance, data and interface may both be inherited. This is expected as data and interface complement each other. The parent structure can be given the general common characteristics while its child structures can be given the more specific characteristics. This allows code reusability by keeping the common code in a common place–the base structure. Otherwise, the code would have to be replicated in all of the child structures, which will lead to maintenance nightmares. Inheritance also enables code extensibility by allowing the creation of new structures that are better suited to our requirements as compared to the existing structures.

Polymorphism, as the name suggests, is the phenomena by virtue of which the same entity can exist in two or more forms. In OOPS, functions can be made to exhibit polymorphic behavior. Functions with different set of formal arguments can have the same name. Polymorphism is of two types: static and dynamic. We will understand how this feature enables C++ programmers to reuse and extend existing code in the subsequent chapters.

1.3.2 What is Object-Oriented Programming?

The term *object-oriented programming (OOP)* is widely used, but experts cannot seem to agree on its exact definition. However, most experts agree that OOP involves defining abstract data types *(ADT)* representing complex real-world or abstract objects and organizing your program around the collection of ADTs with an eye toward exploiting their common features. The term *data abstraction* refers to the process of defining ADTs; *inheritance* and *polymorphism* refer to the mechanisms that enable you to take advantage of the common characteristics of the ADTs–the *objects* in OOP. This chapter further explores these terms later.

The term *abstract data type,* or *ADT* for short, refers to a programmer-defined data type together with a set of operations that can be performed on that data. It is called abstract to distinguish it from the fundamental built-in C data types such as int, char, and double. In C, you can define an ADT using typedef and struct and implementing the operations with a set of functions. As you will learn soon, C++ has much better facilities for defining and using ADTs.

Before you jump into OOP, take note of two points. First, OOP is only a method of designing and implementing software. Use of object-oriented techniques does not impart anything to a finished software product that the user can see. However, as a programmer implementing the software, you can gain significant advantages by using object-oriented methods, especially in large software projects. Because OOP enables you to remain close to the conceptual, higher-level model of the real-world problem you are trying to solve, you can manage the complexity better than with approaches that force you to map the problem to fit the features of the language. You can take advantage of the modularity of objects and implement the program in relatively independent units that are easier to maintain and extend. You can also share code among objects through inheritance.

The second point is that OOP has nothing to do with any programming language, although a programming language that supports OOP makes it easier to implement the object-oriented techniques. As you will see shortly, with some discipline. You can use objects in C programs.

1.3.3 OOP Terminology

As mentioned earlier, there are three basic concepts underlying OOP:

- Data Abstraction
- Inheritance
- Polymorphism

Individually, these concepts have been known and used before, but their use as the foundation of OOP is new.

Data Abstraction

To understand data abstraction, consider the file I/O routines in the C run-time library. These routines enable you to view the file as a stream of bytes and to perform various operations on this stream by calling the file I/O routines. For example, you can call fopen to open a file, fclose to close it, fgetc to read a character from it, and fputc to write a character to it. This abstract model of a file is implemented by defining a data type named FILE to hold all relevant information about a file. The C constructs struct and typedef are used to define FILE. You will find the definition of FILE in the header file stdio.h. You can think of this definition of FILE, together with the functions that operate on it, as a new data type just like C's int or char.

To use the FILE data type, yo do not have to know the C data structure that defines it. In fact, the underlying data structure of FILE can vary from one system to another. Yet, the C file I/O routines work in the same manner on all systems. This is possible because you never access the members of the FILE data structure directly. Instead, you rely on a functions and macros that essentially hide the inner details of FILE. This is known as *data biding.*

Data abstraction is the process of defining a data type, often called an *abstract data type (ADT)*, together with the principle of data hiding. The definition of an ADT involves specifying the internal representation of the ADT's data as well as the functions to be used by others to manipulate the ADT. Data hiding ensures that the internal structure of the ADT can be altered without any fear of breaking the programs that call the functions provided for operations on that ADT. Thus, C's FILE data type is an example of an ADT (see Figure 1.1).

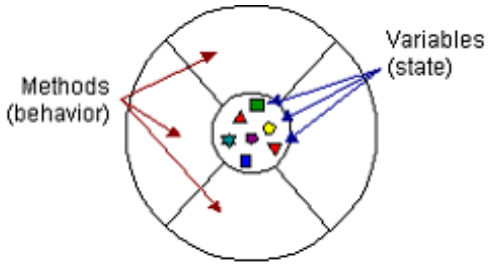
1.4 Objects, Classes and Methods

In OOP, you create an object from an ADT. Essentially, an ADT is a collection of variables together with the functions necessary to operate on those variables. The variables represent the information contained in the object, whereas the functions define the operations that can be performed on that object. You can think of the ADT as a template from which specific instances of objects can be created as needed. The term class is often used for this template. Consequently, class is synonymous with an ADT.

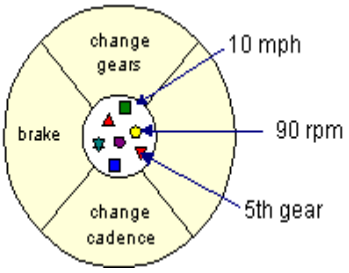
The functions that operate on an object are known as *methods*. This term comes from the object-oriented language Smalltalk. The methods define the behavior of an object. In C++, you do this by calling the appropriate member function of the object. For objects implemented in C, you can send a message by calling a function that accepts a pointer to a data structure representing the ADT's internal structure. Of course, the function must be capable of handling the operation you want. For instance, C's file I/O routines accept a pointer to the FILE structure as an argument. The file I/O routines use that pointer to identify the file with which the I/O operation is to be performed.

Object

An Object is a software bundle of related variables and methods. Software objects are often used to model real world object you find in everyday life.



Visual representation of software object



A bicycle modeled as a software object

Class

A class is a blueprint or prototype that defines the variables and the methods common to all objects of a certain kind.

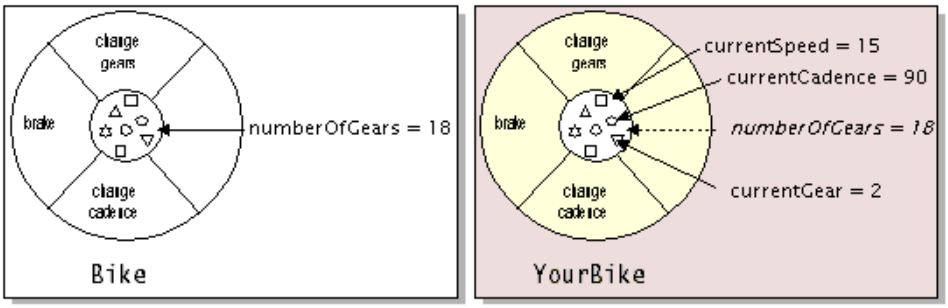


Figure 1.1 : Object and Class

1.5 Inheritance & Polymorphism

Encapsulation

Encapsulation is the grouping together of data and functionality. C++ implements encapsulation by allowing all members of a class to be declared as either public, private, or protected. A public member of the class will be accessible to any function. A private member will only be accessible to functions that are members of that class and to functions and classes explicitly granted access permission by the class ("friends"). A protected member will be accessible to members of classes that inherit from the class in addition to the class itself and any friends.

The Object Oriented (OO) principle is that all and only the functions that can access the internal representation of a type should be encapsulated within the type definition. C++ supports this (via member functions and friend functions), but does not enforce it: the programmer can declare parts or all of the representation of a type to be public, and is also allowed to make public entities that are not part of the representation of the type. Because of this, C++ supports not just OO programming but other weaker decomposition paradigms, like modular programming.

It is generally considered good practice to make all data private or protected, and to make public only those functions that are part of a minimal interface for users of the class, that hides implementation details.

Inheritance

Inheritance allows one data type to acquire properties of other data types. Inheritance from a base class may be declared as public, protected, or private. This access specifier determines whether unrelated and derived classes can access the inherited public and protected members of the base class. Only public inheritance corresponds to what is usually meant by "inheritance". The other two forms are much less frequently used. If the access specifier is omitted, inheritance is assumed to be private for a class base and public for a struct base. Base classes may be declared as virtual; this is called virtual inheritance. Virtual inheritance ensures that only one instance of a base class exists in the inheritance graph, avoiding some of the ambiguity problems of multiple inheritance.

Data abstraction does not cover an important characteristic of objects. Real-world objects do not exist in isolation. Each object is related to one or more other objects. In fact, you can often describe a new kind of object by pointing out how the new object's characteristics and behavior differ from that of a class of objects that already exists. This is what you do when you describe an object with a sentence such as: B is just like A, except that B has…, and B does… Here you are defining objects of type B in terms of those of type A.

This notion of defining a new object in terms of an old one is an integral part of OOP. The term inheritance is used for this concept, because you can think of one class of objects inheriting the data and behavior from another class. Inheritance imposes a hierarchical relationship among classes in which a child class inherits from its parent. In C++ terminology, the parent class is known as the base class; the child is the derived class.

Multiple Inheritance

A real-world object often exhibits characteristics that it inherits from more than one type of object. For instance, on the basis of eating habits, an animal may be classified as a carnivore; other ways of classification place it in a specific family, such as the bear family. When modeling a corporation, you may want to describe a technical manager as someone who is an engineer as well as a manager. An example from the programming world is a full-screen text editor. It displays a block of text on the screen and also stores the text in an internal buffer so that you can perform operations such as insert a character and delete a character. Thus, you may want to say that a text editor inherits its behavior from two classes: a text buffer class and a text display class that, for instance, manages an 80-character by 25 line text display area.

These examples illustrate multiple inheritance–the idea that a class can be derived from more than one base class. Many object-oriented programming languages do not support multiple inheritance, but C++ does.

Polymorphism

In a literal sense, polymorphism means the quality of having more than one form. In the context of OOP, polymorphism refers to the fact that a single operation can have different behavior in different objects. In other words, different objects react differently to the same message. For example, consider the operation of addition. For two numbers, addition should generate the sum. In a programming language that supports OOP, you should be able to express the operation of addition by a single operator, say, +. When this is possible, you can use the expression x+y to denote the sum of x and y, for many different types of x and y: integers, floating-point numbers, and complex numbers, to name a few. You can even define the + operation for two strings to mean the concatenation of the string.

Similarly, suppose a number of geometrical shapes all respond to the message, draw. Each object reacts to this message by displaying its shape on a display screen. Obviously, the actual

mechanism for displaying the object differs from one shape to another, but all shapes, perform this task in response to the same message.

1.6 Dynamic Binding

Dynamic binding occurs when the type of variable changes at run-time. A common way for a variable to change its type is via assignment.

Bike:= MoutainBike is safe

MountainBike:=Bike is not safe

MountainBike is declared to have all the features of Bike so the assignment does no harm.

A variable that starts life of the type Bike may be attached to any object that is a kind of Bike, including MountainBike,RacingBike,…

A variable that starts life of the type MountainBike can only be attached to MountainBike objects but does not include RacingBike or general Bike.

Self Learning Exercises

1. What is an Object?

2. What are key features of Object Oriented Programming?

1.7 Summary

Object-oriented programming (OOP) relies on three basic concepts: data abstraction, inheritance, and polymorphism.

Data abstraction refers to the ability to define abstract data types or ADT's (essentially, user-defined data types) that encapsulate some data together with a set of well-defined operations. Such user-defined datatypes can represent objects in software. The term class refers to the template from which specific instances of objects are created. Objects perform specific actions in response to messages (which may be implemented by function calls).

Inheritance is the mechanism that enables one object to behave just like another, except for some modifications. Inheritance implies a hierarchy of classes with derived classes inheriting behavior from base classes. In the context of software, inheritance promotes the sharing of code and data among classes. Polymorphism refers to the fact that different objects react differently to the same message. In particular, OOP refers to a new way of organizing your program using a collection of objects whose classes are organized in a predefined hierarchy with a view to sharing code and data through inheritance.

A comparison of two implementations of an example–one using procedural approach and the other using OOP-shows that the object-based organization enhances the modularity of the program by placing related data and functions in the same module. This makes it easier to accommodate changes in an object-based program than in a procedural one.

C++ is very powerful programming language to provide security of data & program. It's features of inheritance is used for reusability of code, that enhance the programmers efficiency. Object & Class concept help in working according to real world.

1.8 Glossary

OOP: Object Oriented Programming

Inheritance :A relationship among classes, wherein one class shares the structure or behavior defined in one (single inheritance) or more (multiple inheritance) other classes.

1.9 Further Readings

1. Introduction To Object Oriented Programming And C++ ,Kanetkar , BPB,**ISBN:**
2. A complete Reference to C/C++,Herbert Shildt
3. C++: An introduction to programming by Jense Liberty Tim Keogh: BPB Publications,New Delhi
4. OO Programming in C++ by Robert Lafore: , Galgotia Publications Pvt. Ltd., Daryaganj,New Delhi
5. Object Oriented Programming Using C++, Sanjeev Sofat, Cyber Tech. Publication,
6. Object Oriented Programming in C++ by E. Balaguruswamy, TMH Publishing Co.New Delhi

1.10 Answers to Self Learning Exercises

1. Object is an instance of a class
2. Key features of OOPs are Polymorphism, Inheritance and Data Abstraction.

1.11 Unit End Questions

1. What is procedural programming?
2. What is OOPS
3. What is class? How it is different from objects
4. Write the advantage of OOPS?
5. Explain Inheritance?

UNIT - II

Simple Programming in C++

Structure of the Unit

2.0 Objective

After studying this unit, you will learn:

- Data types of C++
- How to write simple C++ program

2.1 Introduction

A major difference between the data types of C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name, ANSI C++ places no limit on its length and, therefore, all the characters in a name are significant. Care should be exercised while naming a variable which is being shared by more than one file containing C and C++ programs. Some operating systems impose a restrictions the length of such a variable name. Constants refer to fixed values that do not change during the execution of a program. Like C, C++ also supports several kinds of literal constants. They include integers, character floating point numbers and strings. Literal constant do not have memory locations. Example:

```
123             // decimal integer
12.34           // floating point integer
037             // octal integer
0x2             // hexadecimal integer
"C++"           // string constant.
'A'             // character constant
L'ab"           // wide-character constant.
```

The wchar-t type is a wide-character literal introduced by ANSI C++ and is intended character sets that cannot fit a character into a single byte. Wide-character literals be with the letter L.

C++ also recognizes all the backslash character constants available in C.

2.2 Basic Data Types

Both C and C++ compilers support all the built-in (also known as basic or fundamental data types. With the exception of **void**, the basic data types may have several modifiers preceding them to serve the needs of various situations. The modifiers **signed**, **unsigned**, **long**, and **short** may be applied to character and integer basic data types. However, the modifier long may also be applied to **double**. Data types representation is machine specific in C++.

2.2.1 Variables

What Is a Variable?

In C++ a variable is a place to store information. A variable is a location in your computer's memory in which you can store a value and from which you can later retrieve that value.

Your computer's memory can be viewed as a series of cubbyholes. Each cubbyhole is one of many, many such holes all lined up. Each cubbyhole—or memory location—is numbered sequentially. These numbers are known as memory addresses. A variable reserves one or more cubbyholes in which you may store a value.

Let us think that I ask you to retain the number 5 in your mental memory, and then I ask you to memorize also the number 2 at the same time. You have just stored two different values in your memory. Now, if I ask you to add 1 to the first number I said, you should be retaining the numbers 6 (that is 5+1) and 2 in your memory. Values that we could now for example subtract and obtain 4 as result.

The whole process that you have just done with your mental memory is a simile of what a computer can do with two variables. The same process can be expressed in C++ with the following instruction set:

```
a = 5;
b = 2;
a = a + 1;
result = a - b;
```

Obviously, this is a very simple example since we have only used two small integer values, but consider that your computer can store millions of numbers like these at the same time and conduct sophisticated mathematical operations with them.

Therefore, we can define a variable as a portion of memory to store a determined value.

Each variable needs an identifier that distinguishes it from the others, for example, in the previous code the variable identifiers were a, b and result, but we could have called the variables any names we wanted to invent, as long as they were valid identifiers.

2.1.2 Identifiers

A valid identifier is a sequence of one or more letters, digits or underline characters (_). Neither spaces nor punctuation marks or symbols can be part of an identifier. Only letters, digits and underline characters are valid. In addition, variable identifiers always have to begin with a letter. They can also begin with an underline character (_), but this is usually reserved for compiler specific keywords or external identifiers. In no case they can begin with a digit.

Another rule that you have to consider when inventing your own identifiers is that they cannot match any keyword of the C++ language or your compiler's specific ones since they could be confused with these.

The standard keywords are:

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t, while

Additionally, alternative representations for some operators cannot be used as identifiers since they are **reserved words** under some circumstances:

and, and_eq, bitand, bitor, compl, not, not_eq, or, or_eq, xor, xor_eq

Your compiler may also include some additional specific reserved keywords.

Very important: The C++ language is a "case sensitive" language. That means that an identifier written in capital letters is not equivalent to another one with the same name but written in small letters. Thus, for example, the **RESULT** variable is not the same as the **result** variable or the **Result** variable. These are three different variable identifiers.

Fundamental data types

When programming, we store the variables in our computer's memory, but the computer has to know what we want to store in them, since it is not going to occupy the same amount of memory to store a simple number than to store a single letter or a large number, and they are not going to be interpreted the same way.

The memory in our computers is organized in bytes. A byte is the minimum amount of memory that we can manage in C++. A byte can store a relatively small amount of data: one single character or a small integer (generally an integer between 0 and 255). In addition, the computer can manipulate more complex data types that come from grouping several bytes, such as long numbers or non-integer numbers.

Next you have a summary of the basic fundamental data types in C++, as well as the range of values that can be represented with each one:

Name	Description	Size*	Range*
char	Character or small integer.	1byte	signed: -128 to 127 unsigned: 0 to 255
short int (short)	Short Integer.	2bytes	signed: -32768 to 32767 unsigned: 0 to 65535
int	Integer.	4bytes	signed: -2147483648 to 2147483647 unsigned:0 to 4294967295
long int (long)	Long integer.	4bytes	signed: -2147483648 to 2147483647 unsigned:0 to 4294967295
bool	Boolean value. It can take one of two values: true or false.	1byte	true or false
float	Floating point number.	4bytes	3.4e +/- 38 (7 digits)
double	Double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
long double	Long double precision floating point number.	8bytes	1.7e +/- 308 (15 digits)
wchar_t	Wide character.	2bytes	1 wide character

*The values of the columns Size and Range depend on the architecture of the system where the program is compiled and executed. The values shown above are those found on most 32bit systems. But for other systems, the general specification is that int has the natural size suggested by the system architecture (one *word*) and the four integer types char, short, int and long must each one be at least as large as the one preceding it. The same applies to the floating point types float, double and long double, where each one must provide at least as much precision as the preceding one.

Declaration of variables

In order to use a variable in C++, we must first declare it specifying which data type we want it

to be. The syntax to declare a new variable is to write the specifier of the desired data type (like int, bool, float...) followed by a valid variable identifier. For example:

```
int a;
float mynumber;
```

These are two valid declarations of variables. The first one declares a variable of type int with the identifier a. The second one declares a variable of type float with the identifier mynumber. Once declared, the variables a and mynumber can be used within the rest of their scope in the program.

If you are going to declare more than one variable of the same type, you can declare all of them in a single statement by separating their identifiers with commas. For example:

```
int a, b, c;
```

This declares three variables (a, b and c), all of them of type int, and has exactly the same meaning as:

```
int a;
int b;
int c;
```

The integer data types char, short, long and int can be either signed or unsigned depending on the range of numbers needed to be represented. Signed types can represent both positive and negative values, whereas unsigned types can only represent positive values (and zero). This can be specified by using either the specifier signed or the specifier unsigned before the type name. For example:

```
unsigned short int NumberOfSisters;
signed int MyAccountBalance;
```

By default, if we do not specify either signed or unsigned most compiler settings will assume the type to be signed, therefore instead of the second declaration above we could have written:

```
int MyAccountBalance;
```

with exactly the same meaning (with or without the keyword signed)

An exception to this general rule is the char type, which exists by itself and is considered a different fundamental data type from signed char and unsigned char, thought to store characters. You should use either signed or unsigned if you intend to store numerical values in a char-sized variable.

short and long can be used alone as type specifiers. In this case, they refer to their respective integer fundamental types: short is equivalent to short int and long is equivalent to long int. The following two variable declarations are equivalent:

```
short Year;
short int Year;
```

Finally, signed and unsigned may also be used as standalone type specifiers, meaning the same as signed int and unsigned int respectively. The following two declarations are equivalent:

```
unsigned NextYear;
unsigned int NextYear;
```

To see what variable declarations look like in action within a program, we are going to see the C++ code of the example about your mental memory proposed at the beginning of this section:

```
// operating with variables
#include <iostream>
using namespace std;
int main ()
{
        // declaring variables:
        int a, b;
        int result;
        // process:
        a = 5;
        b = 2;
        a = a + 1;
        result = a - b;
        // print out the result:
        cout << result;
        // terminate the program:
        return 0;
}
```

Do not worry if something else than the variable declarations themselves looks a bit strange to you. You will see the rest in detail in coming sections.

Scope of variables

All the variables that we intend to use in a program must have been declared with its type specifier in an earlier point in the code, like we did in the previous code at the beginning of the body of the function main when we declared that a, b, and result were of type int.

A variable can be either of global or local scope. A global variable is a variable declared in the main body of the source code, outside all functions, while a local variable is one declared within the body of a function or a block.

Global variables can be referred from anywhere in the code, even inside functions, whenever it is after its declaration.

The scope of local variables is limited to the block enclosed in braces ({}) where they are declared. For example, if they are declared at the beginning of the body of a function (like in function main) their scope is between its declaration point and the end of that function. In the example above, this means that if another function existed in addition to main, the local variables declared in main could not be accessed from the other function and vice versa.

Table 2.1 Size and Range of C++ Basic Data Types

Type	Bytes	Range
char	1	-128 to 127
unsigned char	1	0 to 255

signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-31768 to 32767
short int	2	-31768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
signed long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E-38 to 3.4E+38
double	8	1.7E-308 to 1.7E+308
long double	10	3.4E-4932 to 1.1E+4932

2.3 User-Defined Datatypes

Structures and Classes

We have used user-defined data types such as **struct** and **union** in C. While these data types are legal in C++, some more features have been added to make them suitable for object-oriented programming. C++ also permits us to define another user-defined data type known as class which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus of object-oriented programming.

Enumerated Data Types

An enumerated data type is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code. The **enum** keyword (from C) automatically enumerates a list of words by assigning them values 0,1,2, and so on. This facility provides an alternative means for creating symbolic constants. The syntax of an **enum** statement is similar to that of the **struct** statement. Examples:

```
enum shape {circle, square, triangle};
enum colour{red, blue, green, yellow};
enum position {off, on};
```

The enumerated data types differ slightly in C++ when compared with those in ANSI C. In C++, the tag names shape, colour, and position become new type names. By using these tag names, we can declare new variables. Examples:

```
shape ellipse;                  // ellipse is of type shape
colour background;              // background is of type of colour
```

ANSI C defines the types of enums to be ints. In C++, each enumerated data type retains its own separate type. This means that C++ does not permit an int value to be automatically converted to an enum value. Examples:

```
colour background = blue;       // allowed
```

```
colour background = 7;                  // Error in C++
colour background = (colour) 7;         // OK
```

However, an enumerated value can be used in place of an int value.

```
int c = red;                            // valid, colour type promoted to int
```

By default, the enumerators are assigned integer values starting with 0 for the first enumerator, 1 for the second, and so on. We can over-ride the default by explicitly assigning integer values to the enumerators. For example:

```
enum colour{red, blue=4, green=8};
enum colour{red=5, blue, green};
```

are valid definitions. In the first case, red is 0 by default. In the second case, blue is 6 and green is 7. Note that the subsequent initialized enumerators are larger by one than their predesessors.

C++ also permits the creation of anonymous enums (i.e., enums without tag names). Example:

```
enum {off, on};
```

Here, off is 0 and on is 1. These constants may be referenced in the same manner as regular constants. Examples:

```
int switch_1 = off;
int switch_2 = on;
```

In practice, enumeration is used to define symbolic constants for a switch statement. Example:

```
enum shape
{
        circle,
        rectangle,
        triangle,
};
int main()
{
        cout << "Enter shape code:";
        int code;
        cin >> code;
        while(code >= circle && code <= triangle)
        {
                switch(code)
                {
                        case circle:
                        ……
                        break;
                        case rectangle:
                        ……
                        break;
                        case triangle:
                        ……
```

```
                    break;
                }
                cout << "Enter shape code:";
                cin >> code;
            }
            cout << "BYE \n";
            return 0;
        }
```

ANSI C permits an enum to be defined within a structure or a class, but the enum is globally visible. In C++, an enum defined within a class (or structure) is local to that class (or structure) only.

Self Learning Exercises

1. What are the basic data type of C++?

2. Is C++ Object Oriented Language?

2.4 Derived Data Types

Arrays

The applications of arrays in C++ is similar to that in C. The only exception is the way character arrays are initialized. When initializing a character array in ANSI C, the compiler will allow us to declare the array size as the exact length of the string constant. For instance,

```
char string [3] = "xyz";
```

is valid in ANSI C. It assumes that the programmer intends to leave out the null character \0 in the definition. But in C++, the size should be one larger than the number of characters in the string.

```
char string [4] = "xyz"; // O.K. for C++
```

Pointers

Pointers are declared and initialized as in C. Examples:

```
int *I;                          // int pointer
ip = &x;                         // address of x assigned to ip
*ip = 10;                        // 10 assigned to x through indirection
```

C++ adds the concept of constant pointer and pointer to a constant.

```
char * const ptr1 = "GOOD";      // constant pointer
```

We cannot modify the address that ptr1 is initialized to.

```
int const * ptr2 = &m;           // pointer to a constant
```

ptr2 is declared as pointer to a constant. It can point to any variable of correct type, but constant of what is points to cannot be changed.

We can also declare both the pointer and the variable as constants in the following way:

```
const char * const cp = "xyz";
```

This statement declares cp as a constant pointer to the string which has been declared constant. In this case, neither the address assigned to the pointer cp nor the constant it point to can be changed.

Pointer are extensively used in C++ for memory management and achieving polymorphism.

Type Compatibility

C++ is very strict with regard to type compatibility as compared to C. For instance, C++ defines int, short int, and long int as three different types. They must be cast when their values are assigned to one another. Similarly, unsigned char, char, and signed char are considered as different types, although each of these has a size of one byte. In C++, the types of values must be the same for complete compatibility, or else, a cast must be applied. These restrictions in C++ are necessary in order to support function overloading where two functions with the same name are distinguished using the type of function arguments.

Another notable difference is the way char constants are stored. In C, they are stored as ints, and therefore,

```
sizeof('x')
```

Scope Resolution Operator

Like C, C++ is also a block-structured language. Blocks and scopes can be used in constructing programs. We knows that the same variable name can be used to have different meanings in different blocks. The scope of the variable extends from the point of its declaration till the end of the block containing the declaration. A variable declared inside a block is said to local to that block. Consider the following segment of a program:

```
……
……
{
        int x = 10;
        ……
        ……
}
……
……
{
        int x = 1;
        ……
        ……
}
```

The two declarations of x refer to two different memory locations containing different values. Statements in the second block cannot refer to the variable x declared in the first block, and vice versa. Blocks in C++ are often nested. For example, the following style common.

```
……
……
{
        int x = 10;
        ……
        ……
        {
                int x = 1;
                ……
                ……
        }
        ……
```

```
}
```

Block2 is contained in block1. Note that a declaration in an inner block hides a declaration of the same variable in an outer block and, therefore, each declaration ofx causes it to refer to a different data object. Within the inner block, the variable x will refer to the data object declared therein.

In C, the global version of a variable cannot be accessed from within the inner block. C++ resolves this problem by introducing a new operator :: called the scope resolution operator. This can be used to uncover a hidden variable. It takes the following form:

```
:: variable-name
```

This operator allows access to the global version of a variable. For example, ::count means the global version of the variable count (and not the local variable count declared in that block).

2.5 Memory Management Operators

C uses malloc() and calloc() functions to allocate memory dynamically at run time. Similarly, it uses the function free() to free dynamically allocated memory. We use dynamic allocation techniques when it is not known in advance how much of memory space is needed. Although, C++ supports these functions, it also defines two unary operators new and delete that perform the task of allocating and freeing the memory in a better and easier way. Since these operators manipulate memory on the free store, they are also known as free store operators.

An object can be created by using new, and destroyed by using delete, as and when required. A data object created inside a block with new, will remain in existence until it is explicitly destroyed by using delete. Thus, the lifetime of an object is directly under our control and is unrelated to the block structure of the program.

The new operator can be used to create objects of any type. It takes the following general form:

```
pointer-variable = new data-type;
```

Here, pointer-variable is a pointer of type data-type. The new operator allocates sufficient memory to hold a data object of type data-type and returns the address of the object. The data-type may be any valid data type. The pointer variable holds the address of the memory space allocated. Example:

```
p = new int;
q = new float;
```

Where p is a pointer of type int and q is a pointer of type float. Here, p and q must have already been declared as pointers of appropriate types. Alternatively, we can combine the declaration of pointers and their assignments as follows:

```
int *p = new int;
float *q = new float;
```

Subsequently, the statements

```
*p = 25;
*q = 7.5;
```

assign 25 to the newly created int object and 7.5 to the float object.

We can also initialize the memory using the new operator. This is done as follows:

Pointer-variable = new data-type (value);

Here value specifies the initial value. Examples:

```
int *p = new int(25);
float *q = new float(7.5;
```

As mentioned earlier, new can be used to create a memory space for any data type including user-defined types such as arrays, structures and classes. The general form for a one-dimensional array is:

Pointer-variable = new data-type[size];

Here, size specifies the number of elements in the array. For example, the statement

```
int *p = new int[10];
```

creates a memory space for an array of 10 integers. P[10] will refer to the first element, p[1] to the second element and so on.

When creating multi-dimensional arrays with new, all the array sizes must be supplied.

```
array_ptr = new int[3] [5] [4];          // legal
array_ptr = new int[m] [5] [4];          // legal
array_ptr = new int[3] [5] [ ];          // illegal
array_ptr = new int[ ] [5] [4];          // illegal
```

The first dimension may be a variable whose value is supplied at runtime. All other must be constants.

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

Delete pointer -variable;

The pointer-variable is the pointer that points to a data object created with new. Examples:

```
delete p;
delete q;
```

If we want to free a dynamically allocated array, we must use the following form of delete:

delete [size] pointer-variable;

The size specifies the number of elements in the array to be freed. The problem with this form is that the programmer should remember the size of the array. Recent versions of C++ do not require the size to be specified. For example,

```
delete [ ]p;
```

will delete the entire array pointed to by p.

What happens if sufficient memory is not available for allocation? in such cases, like malloc(), new returns a null pointer. Therefore, it may be a good idea to check for the pointer produced by new before using it. It is done as follows:

```
……
……
p = new int;
if(!p)
```

```
{
        cout << "allocation failed \n";
}
……
……
```

The new operator offers the following advantages over the function malloc().

1. It automatically computes the size of the data object. We need not use the operator sizeof.
2. It automatically returns the correct pointer type, so that there is no need to use a type cast.
3. It is possible to initialize the object while creating the memory space.
4. Like any other operator, new and delete can be overloaded.

2.5.1 Type Cast Operator

C++ permits explicit type conversion of variables or expressions using the type cast operator.

Traditional C casts are augmented in C++ by a function-call notation as a syntactic alternative. The following two versions are equivalent:

```
(type-name) expression        // C notation
type-name (expression)        // C++ notation
```

Examples:

```
average = sum/(float)i;       // C notation
average = sum/float(i);       // C++ notation
```

A type-name behaves as if it is a function for converting values to a designated type. The function-call notation usually leads to simplest expressions. However, it can be used only if the type is an identifier. For example:

```
p = int * (q);
```

is illegal. In such cases, we must use C type notation.

```
p = (int *) q;
```

Alternatively, we can use typedef to create an identifier of the required type and use it in the functional notation.

```
typedef int * int_pt;
p = int_pt(q);
```

ANSI C++ adds the following new cast operators:

- const_cast
- static_cast
- dynamic_cast
- reinterpret_cast

Application of these operators is discussed in Chapter 16.

Implicit Conversions

We can mix data types in expressions. For example:

m = 5+2.75;

is a valid statement. Whereever data types are mixed in an expression. C++ performs the conversions automatically. This process is known as implicit or automatic conversation.

When the compiler encounters an expression, it divides the expressions into sub-expressions consisting of one operator and one or two operands. For a binary operator, if the operands type differ, the compiler converts one of them to match with the other, using the rule that the "smaller" type is converted to the "wider" type. For example, if one of the operand is an int and the other is a float, the int is converted into a float because a float is wider than an int. The "water-fall" model shown below illustrates this rule.

short à ß char
|
int
|
unsigned
|
long int
|
unsigned long int
|
float
|
double
|
long double

Whenever a char or short int appears in an expression, it is converted to an int. This is called integral widening conversion. The implicit conversion is applied only after completing all integral widening conversions.

2.6 Summary

C++ Has strong data types family. It has more features than C language.

Memory management functions are New & Delete, instead of malloc, calloc etc.

2.7 Glossary

enumerated data type : is another user-defined type which provides a way for attaching names to numbers, thereby increasing comprehensibility of the code.

variable : is a place to store information. A variable is a location in your computer's memory in which you can store a value and from which you can later retrieve that value.

2.8 Further Readings

1. Introduction To Object Oriented Programming And C++ ,Kanetkar , BPB,**ISBN:** 8176568635
2. A complete Reference to C/C++,Herbert Shildt
3. C++: An introduction to programming by Jense Liberty Tim Keogh: BPB Publications, New Delhi
4. OO Programming in C++ by Robert Lafore: , Galgotia Publications Pvt. Ltd.,

Daryaganj,New Delhi

5. Object Oriented Programming Using C++, Sanjeev Sofat, Cyber Tech. Publication, New Delhi

6. Object Oriented Programming in C++ by E. Balaguruswamy, TMH Publishing Co. Ltd.,New Delhi

2.9 Answers to Self Learning Exercises

1. Basic Data types of C++ are int, char and float.

2. Yes C++ is an Object Oriented Programming Language.

2.10 Unit End Questions

Q1. What is the difference between Basic & User define Data types.

Q2. Array is user define or derived data types , How?

Q3. Explain all derived data type?

Q4. Waht is type Cast operator ?

UNIT - III

Control Structure and Functions in C++

Structure of the Unit

3.0 Objective

After studying this unit, you will learn

- Various control structure of C++
- Function & it's uses in C++

3.1 Control Structures

A program is usually not limited to a linear sequence of instructions. During its process it may bifurcate, repeat code or take decisions. For that purpose, C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

With the introduction of control structures we are going to have to introduce a new concept: the *compound-statement* or *block*. A block is a group of statements which are separated by semicolons (;) like all C++ statements, but grouped together in a block enclosed in braces: { }:

```
{statement1; statement2; statement3;}
```

Most of the control structures that we will see in this section require a generic statement as part of its syntax. A statement can be either a simple statement (a simple instruction ending with a semicolon) or a compound statement (several instructions grouped in a block), like the one just described. In the case that we want the statement to be a simple statement, we do not need to enclose it in braces ({}). But in the case that we want the statement to be a compound statement it must be enclosed between braces ({}), forming a block.

3.1.1 Conditional structure

The if keyword is used to execute a statement or block only if a condition is fulfilled. Its form is:

```
if (condition) statement;
```

Where condition is the expression that is being evaluated. If this condition is true, statement is executed. If it is false, statement is ignored (not executed) and the program continues right after this conditional structure.

For example, the following code fragment prints x is 100 only if the value stored in the x variable is indeed 100:

```
if (x == 100)
cout << "x is 100";
```

If we want more than a single statement to be executed in case that the condition is true we can specify a block using braces { }:

```
if (x == 100)
{
cout << "x is ";
cout << x;
}
```

We can additionally specify what we want to happen if the condition is not fulfilled by using the keyword else. Its form used in conjunction with if is:

```
if (condition) statement1;else statement2;
```

For example:

```
if (x == 100)
cout << "x is 100";
else
cout << "x is not 100";
```

prints on the screen x is 100 if indeed x has a value of 100, but if it has not -and only if not- it prints out x is not 100.

The if + else structures can be concatenated with the intention of verifying a range of values. The following example shows its use telling if the value currently stored in x is positive, negative or none of them (i.e. zero):

```
if (x > 0)
cout << "x is positive";
else if (x < 0)
cout << "x is negative";
else
cout << "x is 0";
```

Remember that in case that we want more than a single statement to be executed, we must group them in a block by enclosing them in braces { }.

3.2 Loop Structure

Loops have as purpose to repeat a statement a certain number of times or while a condition is fulfilled.

The while loop

Its format is:

while (expression) statement and its functionality is simply to repeat statement while the condition set in expression is true.

For example, we are going to make a program to countdown using a while-loop:

```
// custom countdown using while
#include <iostream>
using namespace std;
int main ()
{
int n;
cout << "Enter the starting number > ";
cin >> n;
while (n>0)
{
cout << n << ", ";
—n;
```

```
    }
    cout << "FIRE!\n";
    return 0;
}
    Enter the starting number > 8
    8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

When the program starts the user is prompted to insert a starting number for the countdown. Then the while loop begins, if the value entered by the user fulfills the condition n>0 (that n is greater than zero) the block that follows the condition will be executed and repeated while the condition (n>0) remains being true.

The whole process of the previous program can be interpreted according to the following script (beginning in main):

1. User assigns a value to n
2. The while condition is checked (n>0). At this point there are two possibilities:
   * condition is true: statement is executed (to step 3)
   * condition is false: ignore statement and continue after it (to step 5)
3. Execute statement:
   cout << n << ", ";
   —n;
   (prints the value of n on the screen and decreases n by 1)
4. End of block. Return automatically to step 2
5. Continue the program right after the block: print FIRE! and end program.

When creating a while-loop, we must always consider that it has to end at some point, therefore we must provide within the block some method to force the condition to become false at some point, otherwise the loop will continue looping forever. In this case we have included —n; that decreases the value of the variable that is being evaluated in the condition (n) by one - this will eventually make the condition (n>0) to become false after a certain number of loop iterations: to be more specific, when n becomes 0, that is where our while-loop and our countdown end.

Of course this is such a simple action for our computer that the whole countdown is performed instantly without any practical delay between numbers.

The do-while loop

Its format is:

do statement while (condition);

Its functionality is exactly the same as the while loop, except that condition in the do-while loop is evaluated after the execution of statement instead of before, granting at least one execution of statement even if condition is never fulfilled. For example, the following example program echoes any number you enter until you enter 0.

```
// number echoer
#include <iostream>
```

```
using namespace std;
int main ()
{
        unsigned long n;
        do {
        cout << "Enter number (0 to end): ";
        cin >> n;
        cout << "You entered: " << n << "\n";
        } while (n != 0);
        return 0;
}
        Enter number (0 to end): 12345
        You entered: 12345
        Enter number (0 to end): 160277
        You entered: 160277
        Enter number (0 to end): 0
        You entered: 0
```

The do-while loop is usually used when the condition that has to determine the end of the loop is determined within the loop statement itself, like in the previous case, where the user input within the block is what is used to determine if the loop has to end. In fact if you never enter the value 0 in the previous example you can be prompted for more numbers forever.

The for loop

Its format is:

for (initialization; condition; increase) statement;

and its main function is to repeat statement while condition remains true, like the while loop. But in addition, the for loop provides specific locations to contain an initialization statement and an increase statement. So this loop is specially designed to perform a repetitive action with a counter which is initialized and increased on each iteration.

It works in the following way:

1. initialization is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. condition is checked. If it is true the loop continues, otherwise the loop ends and statement is skipped (not executed).
3. statement is executed. As usual, it can be either a single statement or a block enclosed in braces { }.
4. finally, whatever is specified in the increase field is executed and the loop gets back to step 2.

Here is an example of countdown using a for loop:

```
// countdown using a for loop
```

```
#include <iostream>
using namespace std;
int main ()
{
        for (int n=10; n>0; n—) {
        cout << n << ", ";
        }
        cout << "FIRE!\n";
        return 0;
}
        10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

The initialization and increase fields are optional. They can remain empty, but in all cases the semicolon signs between them must be written. For example we could write: for (;n<10;) if we wanted to specify no initialization and no increase; or for (;n<10;n++) if we wanted to include an increase field but no initialization (maybe because the variable was already initialized before).

Optionally, using the comma operator (,) we can specify more than one expression in any of the fields included in a for loop, like in initialization, for example. The comma operator (,) is an expression separator, it serves to separate more than one expression where only one is generally expected. For example, suppose that we wanted to initialize more than one variable in our loop:

```
        for ( n=0, i=100 ; n!=i ; n++, i— )
{
  // whatever here...
}
```

This loop will execute for 50 times if neither n or i are modified within the loop:

3.2.1 Jump statements

The break statement

Using break we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. For example, we are going to stop the count down before its natural end (maybe because of an engine check failure?):

```
// break loop example
#include <iostream>
using namespace std;
int main ()
{
        int n;
        for (n=10; n>0; n—)
```

```
        {
        cout << n << ", ";
        if (n==3)
                {
                cout << "countdown aborted!";
                break;
                }
        }
        return 0;
}
```

10, 9, 8, 7, 6, 5, 4, 3, countdown aborted!

The continue statement

The continue statement causes the program to skip the rest of the loop in the current iteration as if the end of the statement block had been reached, causing it to jump to the start of the following iteration. For example, we are going to skip the number 5 in our countdown:

```
// continue loop example
#include <iostream>
using namespace std;
int main ()
{
        for (int n=10; n>0; n—) {
        if (n==5) continue;
        cout << n << ", ";
        }
        cout << "FIRE!\n";
        return 0;
}
```

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

3.2.2 The goto statement

goto allows to make an absolute jump to another point in the program. You should use this feature with caution since its execution causes an unconditional jump ignoring any type of nesting limitations.

The destination point is identified by a label, which is then used as an argument for the goto statement. A label is made of a valid identifier followed by a colon (:).

Generally speaking, this instruction has no concrete use in structured or object oriented programming aside from those that low-level programming fans may find for it. For example, here is our countdown loop using goto:

```
// goto loop example
#include <iostream>
```

```
using namespace std;
int main ()
{
        int n=10;
        loop:
        cout << n << ", ";
        n—;
        if (n>0) goto loop;
        cout << "FIRE!\n";
        return 0;
}       10, 9, 8, 7, 6, 5, 4, 3, 2, 1, FIRE!
```

The exit function

exit is a function defined in the cstdlib library.

The purpose of exit is to terminate the current program with a specific exit code. Its prototype is:

```
void exit (int exitcode);
```

The exitcode is used by some operating systems and may be used by calling programs. By convention, an exit code of 0 means that the program finished normally and any other value means that some error or unexpected results happened.

3.3 The selective structure

The syntax of the switch statement is a bit peculiar. Its objective is to check several possible constant values for an expression. Something similar to what we did at the beginning of this section with the concatenation of several if and else if instructions. Its form is the following:

```
switch (expression)
{
        case constant1:
        group of statements 1;
        break;
        case constant2:
        group of statements 2;
        break;
        .
        .
        .
        default:
        default group of statements
}
```

It works in the following way: switch evaluates expression and checks if it is equivalent to constant1, if it is, it executes group of statements 1 until it finds the break statement. When it finds this break statement the program jumps to the end of the switch selective structure.

If expression was not equal to constant1 it will be checked against constant2. If it is equal to this, it will execute group of statements 2 until a break keyword is found, and then will jump to the end of the switch selective structure.

Finally, if the value of expression did not match any of the previously specified constants (you can include as many case labels as values you want to check), the program will execute the statements included after the default: label, if it exists (since it is optional).

Both of the following code fragments have the same behavior:

switch example	**if-else equivalent**
switch (x) {	
case 1:	if (x == 1) {
cout << "x is 1";	cout << "x is 1";
break;	}
case 2:	else if (x == 2) {
cout << "x is 2";	cout << "x is 2";
break;	}
default:	else {
cout << "value of x unknown";	cout << "value of x unknown";
}	}

The switch statement is a bit peculiar within the C++ language because it uses labels instead of blocks. This forces us to put break statements after the group of statements that we want to be executed for a specific condition. Otherwise the remainder statements -including those corresponding to other labels- will also be executed until the end of the switch selective block or a break statement is reached.

For example, if we did not include a break statement after the first group for case one, the program will not automatically jump to the end of the switch selective block and it would continue executing the rest of statements until it reaches either a break instruction or the end of the switch selective block. This makes unnecessary to include braces { } surrounding the statements for each of the cases, and it can also be useful to execute the same block of instructions for different possible values for the expression being evaluated. For example:

```
switch (x) {
case 1:
case 2:
case 3:
cout << "x is 1, 2 or 3";
break;
default:
cout << "x is not 1, 2 nor 3";
```

```
}
```

Notice that switch can only be used to compare an expression against constants. Therefore we cannot put variables as labels (for example case n: where n is a variable) or ranges (case (1..3):) because they are not valid C++ constants.

If you need to check ranges or values that are not constants, use a concatenation of if and else if statements.

When the function is called, control is transferred to the first statement in the function body. The other statements in the function body are then executed and control returns to the main program when the closing brace is encountered. C++ is no exception. Functions continue to be the building blocks of C++ programs. In fact, C++ has added many new features to functions to make them more reliable and flexible. Like C++ operators, a C++ function can be overloaded to make it perform different tasks depending on the arguments passed to it. Most of these modifications are aimed at meeting the requirements of object-oriented facilities.

3.4 main() FUNCTION

C does not specify any return type for the main() function which is the starting point for the execution of a program. The definition of main() would look like this:

```
main()
{
        // main program statements
}
```

This is perfectly valid because the main() in C does not return any value.

In C++, the main() returns a value of type int to the operating system. C++, therefore, explicitly defines main() as matching one of the following prototypes:

```
int main();
int main(int argc, char * argv[ ]);
```

The functions that have a return value should use the return statement for termination. The main() function in C++ is, therefore, defined as follows:

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

Since the return type of functions is int by default, the keyword int in the main() header is optional. Most C++ compilers will generate an error or warning if there is no return statement. Turbo C++ issues the warning

```
Function should return a value
```

And then proceeds to compile the program. It is good programming practice to actually return a

value from main().

Many operating systems test the return value (called exit value) to determine if there any problem. The normal convention is that an exit value of zero means the program run successfully, while a nonzero value means there was a problem. The explicit use of return() statement will indicate that the program was successfully executed.

3.5 Function Prototyping

Function prototyping is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type arguments and the type of return values. With function prototyping, a template is always use when declaring and defining a function. When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself. These checks and controls did not exists in the conventional C functions.

Remember, C also uses prototyping. But it was introduced first in C++ by Stroustrup and the success of this feature inspired the ANSI C committee to adopt it. However, there is a major difference in prototyping between C and C++. While C++ makes the prototyping essential, ANSI C makes it optional, perhaps, to preserve the compatibility with classic C.

Function prototype is a declaration statement in the calling program and is of the following form:

```
type function-name (argument-list);
```

The argument-list contains the types and names of arguments that must be passed to the function.

Example:

```
Float volume(int x, float y, float z);
```

Note that each argument variable must be declared independently inside the parentheses. That is, combined declaration like:

```
float volume(int x, float y, z);
```

is illegal.

In a function declaration, the names of the arguments are dummy variables and therefore they are optional. That is, the form

```
float volume(int, float, float);
```

is acceptable at the place of declaration. At this stage, the compiler only checks for the type arguments when the function is called.

In general, we can either include or exclude the variable names in the argument list of prototypes. The variable names in the prototype just act as placeholders and therefore, if names are used, they don't have to match the names used in the function call or function definition.

In the function definition, names are required because the arguments must be referenced inside the function. Example:

```
float volume(int a,float b,float, c)
{
        float v = a*b*c;
        ……
```

```
        ……
}
```

The function volume() can be invoked in a program as follows:

```
float cube1 = volume(b1,w2,h1);                    // Function call
```

The variable b1,w1, and h1 are known as the actual parameters which specify the dimensions of cube1. Their types (which have been declared earlier) should match with the types declared in the prototype. Remember, the calling statement should not include type names in the argument list.

We can also declare a function with an empty argument list, as in the following example:

```
void display( );
```

In C++ this means that the function does not pass any parameters. It is identical to the statement.

```
void display(void);
```

However, in C, an empty parentheses implies any number of arguments. That is, we have foregone prototyping. A C++ function can also have an 'open' parameter list by the use of ellipses in the prototype as shown below:

```
void do_something(…);
```

3.6 Parameter passing

3.6.1 Call By Reference

In Statement C, a function call passes arguments by value. The called function creates a new set of variables and copies the values of arguments into them. The function does not have access to the actual variables in the calling program and can only work on the copies of values. This mechanism is fine if the function does not need to alter the values of the original variables in the calling program. But, there may arise situations where we would like to change the values of variables in the calling program. For example, in bubble sort, we compare two adjacent elements in the list and interchange their values if the first element is greater than the second. If a function is used for bubble sort, then it should be able to alter the values of variables in the calling function, which is not possible if the call-by-value method is used.

Provision of the reference variables in C++ permits us to pass parameters to the functions by reference. When we pass arguments by reference, the 'formal' arguments in the called function become aliases to the 'actual' arguments in the calling function. This means that when the function is working with its own arguments, it is actually working on the original data. Consider the following function:

```
void swap(int a,int b)                    // a and b are reference variables
{
        int t = a;                        // Dynamic initialization
        a = b;
        b = t;
}
```

Now, if m and n are two integer variables, then the function call

```
swap(n, n);
```

will exchange the values of m and n using their aliases (reference variables) a and b. Reference variables have been discussed in detail in Chapter 3. In traditional C, this is accomplished using pointers and indirection as follows:

```
void swap1(int *a, int *b)          /* Function definition */
{
        int t;
        t = *a;                     /* assign the value at address a to t */
        *a = *b;                    /* put the value at b into a */
        *b = t;                     /* put the value at t into b */
```

This function can be called as follows:

```
swap1(x, y);                        /* call by passing */
                                    /* addresses of variables */
```

This approach is also acceptable in C++, Note that the call-by-reference method is neatering its approach.

Return by Reference

A function can also return a reference. Consider the following function:

```
int & max(int &x, int &y)
{
        if (x > y)
                return x;
        else
                return y;
}
```

Since the return type of max() is int &, the function returns reference to x or y (and not the values). Then a function call such as max(a, b) will yield a reference to either a or b depending in their values. This means that this function call can appear on the left-hand side of an assignment statement. That is, the statement

```
max(a, b) = -1;
```

is legal and assigns –1 to a if it is larger, otherwise –1 to b.

Inline Functions

One of the objectives of using functions in a program is to save same memory space, which becomes appreciable when a function is likely to be called many times. However, every time a function is called, it takes a lot of extra time in executing a series of instructions for tasks such as jumping to the function, saving registers, pushing arguments into the stock, and returning to the calling function. When a function is small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use macro definitions, popularly known as macros. Preprocessor macros are popular in C. The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.

C++ has a different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature, called inline function. An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code (something similar to macros expansion). The inline functions are defined as follows:

```
inline function-header
{
        function body
}
```

Example:

```
inline double cube(double = cube(double a)
{
        return (a* a* a).
}
```

The above inline function can be invoked by statements like

```
c = cube(3.0);
d = cube(2.5+1.5);
```

One the execution of these statements, the values of c and d will be 27 and 64 respectively, the arguments are expressions such as 2.5 + 1.5, the function passes the value of the expression 4 in this case. This makes the inline feature far superior to macros.

It is easy to make a function inline. All we need to do is to prefix the keyword inline to function definition. All inline functions must be defined before they are called.

We should exercise care before making a function inline. The speed benefits of inline functions diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of inline functions may be lost .In such cases, the use of normal functions will be more meaningful. Usually, the functions are made inline when they are small enough to be defined in one or two lines. Example:

```
inline double cube (double a)          {return(a* a* a);}
```

Remember that the inline keyword merely sends a request, not a command, to the compiler. The compiler may ignore this request if the function definition is too long or too complicated a compile the function as a normal function.

Some of the situations where inline expansion may not work are:

1. For functions returning values, if a loop, a switch, or a goto exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain static variables.
4. If inline functions are recursive.

Note Inline expansion makes a program run faster because the overhead of a function call and return is eliminated. However, it makes the program to take up memory because the statements that define the inline function are reproduced each point where the function is called. So, a trade-off becomes necessary:

INLINE FUNCTIONS

```
#include <iostream>
using namespace std;
        inline float mul(float x, float y)
        {
                return (x * y);
        }
        inline double div(double p, double q)
        {
                return (p/q);
        }
int main()
{
        float a = 12.345;
        float b = 9.82;
        cout << mul(a,b) << "\n";
        cout << div(a,b) << "\n";
        return 0;
}
```

The output of program 4.1 would be

```
121.228
1.25713
```

3.6.2 Default Arguments

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a default value to the parameter which does not have a matching argument in the function call. Default values are specified when the function is declared. The compiler looks at the prototype to see how many arguments a function uses and alerts the program for possible default values. Here is an example of a prototype (i.e. function declaration) with default values:

```
float amount (float principal, int period, float rate = 0.15);
```

The default value is specified in a manner syntactically similar to a variable initialization. The above prototype declares a default value of 0.15 to the argument rate. A subsequent function call like

```
value = amount (5000,7);                // one argument missing
```

passes the value of 5000 to principal and 7 to period and then lets the function use

default value of 0.15 for rate. The call

```
value = amount (5000,5,0.12);          // no missing argument
```

passes an explicit value of 0.12 to rate.

A default argument is checked for type at the time of declaration and evaluated at the time call. One important point to note is that only the trailing arguments can have default values is important to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

```
int mul (int i, int j = 5, int k = 10);        // legal
int mul (int i = 5, int j);                    // illegal
int mul (int i = 0, int j, int k = 10);        // illegal
int mul (int i = 2, int j = 5, int k = 10);    // legal
```

Default arguments are useful in situations where some arguments always have the same value. For instance, bank interest may remain the same for all customers for a particular period of deposit. It also provides a greater flexibility to the programmers. A function can be written with more parameters than are required for its most common application. Using default arguments, a programmer can use only those arguments that are meaningful to a particular situation. Program 4.2 illustrates the use of default arguments.

DEFAULT ARGUMENTS

```
#include <isotream>
using namespace std;
int main()
{
        float amount;
        float value  (float p, int n, float r = 0.15);          // prototype
        void printline (char ch = '*', int len = 40;            // prototype
        printline();                                // uses default values for arguments
        amount = value(5000.00,5);                  // default for 3rd argument
        cout << "\n              Final Value = " << amount << "\n\n";
        printline('=');                             // use default value for 2nd argument
        return 0;
}
/*_____
_____*/
float value (float p, int n, float r)
{
        int year = 1;
        float sum = p;
```

```
        while(year <= n)
        {
                sum = sum*(1+r);
                year = year+1;
        }
        return
}
void printline(char ch, int len)
{
        for(int i = 1;      i = len;  i++)      printf("%c" ,ch);
        printf("\n");
}
```

Advantages of providing the default arguments are:

1. We can use default arguments to add new parameters to the existing functions.
2. Default arguments can be used to combine similar functions into one.

3.6.3 Const Arguments

In C++, an argument to a function can be declared as const as shown below:

```
int strlen(const char *p);
int length(const string &s);
```

The qualifier const tells the compiler that the function should not modify the argument. The compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass arguments by reference or pointers.

Self Learning Exercises

1. What is main()?
2. What is difference between C and C++ parameter passing mechanism.

3.7 Summary

- It is possible to reduce the size of program by calling and using functions at different places in the program.
- In C++, the main() returns a value of type int to the operating system. Since the return type of functions is int by default, the keyword int in the main() header is optional. Most C++ compilers issue a warning, if there is no return statement.
- Function prototyping gives the compiler the details about the functions such as the number and types of arguments and the type of return values.
- Reference variables in C++ permit us to pass parameters to the functions by reference. A function can also return a reference to a variable.
- When a function is declared inline the compiler replaces the function call with the respective function code. Normally, a small size function is made as inline.
- The compiler may ignore the inline declaration if the function declaration is too long or

too complicated and hence compile the function as a normal function.

- C++ allows us to assign default values to the function parameters when the function is declared. In such a case we can call a function without specifying all its arguments. The defaults are always added from right to left.
- In C++, an argument to a function can be declared as const, indicating that the function should not modify the argument.
- C++ allows function overloading. That is, we can have more than one function with the same name in our program. The compiler matches the function call with the exact function code by checking the number and type of the arguments.
- C++ supports two new types of functions, namely friend functions and virtual functions.
- Many mathematical computations can be carried out using the library functions supported by the C++ standard library.

3.8 Glossary

function prototype : The prototype describes the function interface to the compiler by giving details such as the number and type arguments and the type of return values.

Default arguments : Default values assigned to function parameters if no value is passsed.

3.9 Further Readings

1. Introduction To Object Oriented Programming And C++ ,Kanetkar , BPB,**ISBN:** 8176568635
2. A complete Reference to C/C++,Herbert Shildt
3. C++: An introduction to programming by Jense Liberty Tim Keogh: BPB Publications, New Delhi
4. OO Programming in C++ by Robert Lafore: , Galgotia Publications Pvt. Ltd., Daryaganj,New Delhi
5. Object Oriented Programming Using C++, Sanjeev Sofat, Cyber Tech. Publication, New Delhi
6. Object Oriented Programming in C++ by E. Balaguruswamy, TMH Publishing Co. Ltd.,New Delhi

3.10 Answers to self learning exercises

1. main function is the starting point for the execution of a program.
2. C++ function call provides : Call by reference and default arguments.

3.11 Unit End Questions

1. State whether the following statements are TRUE or FALSE.
   (a) A function argument is a value returned by the function to the calling program.
   (b) When arguments are passed by value, the function works with the original arguments in the calling program.
   (c) When a function returns a value, the entire function call can be assigned to variable.

(d) A function can return a value by reference.

(e) When an argument is passed by reference, a temporary variable is created in the calling program to hold the argument value.

(f) It is not necessary to specify the variable name in the function prototype.

2. What are the advantages of function prototypes in C++?
3. Describe the different styles of writing prototypes.
4. Find errors, if any, in the following function prototypes:

   (a) float average(x,y);

   (b) int mul(int a,b);

   (c) int display(...);

   (d) void Vect(int? &V, int & size);

   (e) void print(floatdata [], size = 20);

5. What is the main advantage of passing arguments by reference?
6. When will you make a function inline? Why?
7. How does an inline function differ from a preprocessor macro?
8. When do we need to use default arguments in a function?
9. What is the significance of an empty parenthesis in a function declaration?

UNIT - IV

Classes and Objects

Structure of the Unit

4.0 OBJECTIVE

After completion of this unit, you should be able to write programs using

- Structures in C and C++
- Classes and Objects
- Access restrictions to object members

4.1 INTRODUCTION

The concept of object-oriented programming (OOP) was discussed in previous units. Programming in object oriented paradigm starts with identification of objects similar to real world objects. The language based on OOP concepts must support implementation of all features required to represent objects. Simple programming constructs of the language such as data types, operators, control structures and functions are discussed in previous units. Most of the above features are similar to its predecessor language C. This is the first unit that describes object oriented constructs of C++

As discussed in previous units, an object can be described by its state and behavior. States can be represented by data or variables and behavior can be by functions. An object can be represented by grouping of variables and functions. Grouping of variables is possible in C through structures. This unit starts with the properties of structures in C that combines one or more data types followed by structures of C++ that combines one or more data types and functions. The structures can then be extended to classes. The classes represent a blueprint of object properties. This unit covers definition of classes, declaration of objects, characteristics of class members and their access specifiers, object properties and use of constructor and destructors.

4.2 Introduction to structures

Combining one or more data types belonging to single entity is the basic use of the structures. By this way state of an object can be represented but behavior that manipulates the state can not be attached. In C++ structures, it is possible to attach the behavior i.e. function. The following subsection describes the structures in C and C++.

4.2.1 Structures In C

Though, it is possible to combine more than one variable using structure in C, the structure has following limitations when used in C language:

- Only variables of different data types can be declared in the structure as member, functions are not allowed as members.
- Direct access to data members is possible, because by default all the member variables are public. Hence, security to data or data hiding is not provided.
- The struct data type is not treated as built in type i.e., use of struct keyword is necessary to declare objects.
- The member variables cannot be initialized inside the structure.

The syntax and example of structure declaration is as follows:

Syntax:

```
Struct <struct name>
{
Variable 1;
```

Variable 2 ;

variable n;

} ;

Example:

```
struct item
{
int codeno;
float prize;
int qty;
};
```

In the above example, item is a structure name. The variables codeno, prize and qty are member variables. Thus, a custom data type is created by a combination of one or more variables.

Now the object of structure item can be declared as struct item a,*b ;

The object declaration is same as declaration of variables built in data types. The object a and pointer *b can access the member variables of struct item. Use of keyword struct is necessary.

The operators (.) dot and (->) arrow are used to access the member variables of struct. The dot operator is used when simple object is declared and arrow operator is used when object is pointer to structure. The access of members can be accomplished as per the syntax given below:

[Object name][Operator][Member variable name]

When an object is a simple variable, access to members is done as below:

```
a.codeno;
a.prize;
a.qty;
```

When an object is a pointer to structure then members are accessed as below:

```
a->codeno;
a->prize;
a->qty;
```

The following program illustrates the above discussion.

Program 4.1 :

//Write a program in C to declare structure, access and initialize its member using object of structure type.

```
# include <stdio.h>
# include <conio.h>
struct item                    //struct declaration
{
int codeno;
float prize;
```

```
int qty;  };
void main ( )
{
struct item a, *b;          //object declaration
clrscr();
a.codeno=123;               // direct access & initialization of member variables
a.prize=150.75l
a.qty=150;
cout<<"\n With simple variable ";
cout<< "\n Codeno  : %d"<<a.codeno;
cout<< "\n Prize  : %d"<<a.prize;
cout<< "\n Qty    : %d"<<a.qty;
b->codeno=124;  b->prize=200.75;  b->qty=75;
cout<<"\n With pointer to Structure ";
cout<< "\n Codeno  : %d"<<b->codeno;
cout<< "\n Prizeo  : %d"<<b->prize;
cout<< "\n Qty    : %d"<<b->qty;
}
```

Explanation: The above program is compiled with C compiler. The following discussion is in accord-ance with C compiler. In the above program, the structure item is declared with three member variables. The initialization of member variables inside the struct is not permitted. The declaration of member vari-ables is enclosed within the curly braces. The struct declaration is terminated by semicolon.

Item a, *b; // object declaration in c++

4.2.2 Structures In C++

First limitation of structures in C is that functions are not allowed within structures. But in C functions inside the structure are allowed. Following example illustrate the syntax and example of C++ structure.

Syntax

```
struct <struct-name>
{
variable1;
variable2;
.
variablen;
type function-name(arguments)
{
body;
```

```
}
};
```

Example

```
Struct item
{
int codeno;
float prize;
int qty;
int init (int code, float p, int q)
{
codeno = code;
prize =p;
qty = q;
}
};
```

While declaring an object the keyword struct is omitted in C++ which is compulsory in C. The structure item is a user-defined data type. C++ behaves structure data type as built in type and allows: variable declaration.

C++ introduces new keyword class, which is similar to structure. The other improvements are discussed with use of class in the following section.

4.3 Classes In C++

Classes and structure are the same with only a small difference that is explained later. The following discussion of class is applicable to struct too.

A class is nothing but grouping of variables of different data types with functions as was discussed in structures of C++. However, classes can also provide security of data and other features of object. Each variable of a class is called as member variable and functions are called as member functions or methods.

In C++, it is possible to restrict access to member variables directly by object that is not possible in C. It is always a programmer's choice to allow or disallow direct access to member variables. The mechanism of restricting access to variable outside the class is called as data hiding or encapsulation. In such a case, only member functions can access data. If class is used in place of struct, it restricts the access to member variables as shown in Figure 4.1 and 4.2.

```
struct demo d;
```

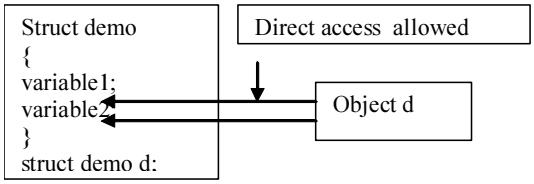


Figure 4.1 Structures in C

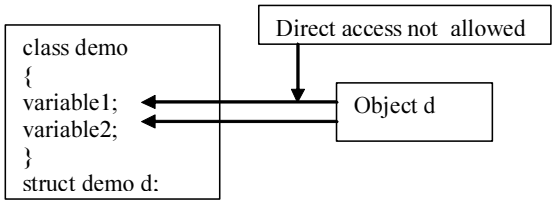


Figure 4.2 Class in C++

4.3.1 Declaring Classes

Whole declaration of class is given in Table 4.1. The class is used to pack variables and functions together. The class has a mechanism to prevent direct access to its members, which is the central idea of object-oriented programming. The class declaration is also known as formation of new abstract data type. The abstract data type can be used as basic data type such as int, float etc. The class consists of member variables that hold data and member functions that operate on the member variables of the same class.

Table 4.1: Syntax and an example of class

Syntax of class declaration	**Example of class**

Class <name of class>
{
private:
 	
declaration of variable ;
 	
prototype declaration of function
 	
public :
 	
declaration of variable;
 	
prototype declaration of function
 	
};
	Class item // class declaration
	{
	private :
	
	int codeno;
	int prize;
	int qty;
	void values;
	
	public :
	
	void show();
	
	};

The class and struct are keywords. The class declaration is same as struct declaration. The declaration of a class is enclosed with curly braces and terminated by a semi-colon. The member variables and functions are divided in two sections i.e., private and public. The private and public keywords are terminated by colon (:). The object cannot directly access the member variables and declared in private section but it can access the member variables and functions of public section. The private members of a class can only be accessed by member function of the same class.

4.3.2 Using Objects

A class declaration only builds the structure of object. The member variables and functions are combined in the class. The declaration of objects is same as declaration of variables of basic data types. Defining data type is known as class instantiation. When objects are created only during that moment, memory is allocated to them.

Consider following examples:

(a) int x,y,z //Declaration of integer variables

(b) char a,b,c //Declaration of character variables

(c) item a,b,*c // Declaration of object or class type variables

in example (a) three variables x, y and z of integer types are declared. In example (b) three variables a, b and c of char type are declared. In the same fashion the third example declares three object a, b and c of class item. The object *c is pointer to class item.

An object is an abstract unit having following properties:

(A) it is individual.

(B) It points to a thing, either physical or logical that is identifiable by the user.

(C) It holds data as well as operation method that handle data.

(D) Its scope is limited to the block in which it is defined

4.3.3 Accessing Members

The object can access the public member variables and functions of a class by using operator dot (.) and arrow (->). The syntax is as follows:

[Object Name][Operator] [Member name]

To have access to member of class item, the statement would be

a.show ()

Where a is an object and show () is a member function. The dot operator is used because a is a simple object.

In statement,

c->show () ;

*c is pointer to class item; therefore, the arrow operator is used to access the member.

Consider the given example:

```
class item          // class declaration
{
int codeno;
float prize;
int qty;
} ;
```

We replaced the struct keyword with class. If last few programs are executed with class, they won't work. For example,

```
void main ()
{
item a,*b;  // object declaration
a.codeno=123;  // Direct access is not allowed
a.prize=150.75;
a.qty= 150;
}
```

The above program will generate error messages, one of them being "item::codeno' is not accessible". This is because the object cannot directly access the member variables of class that is possible with structure. Hence, we can say that the difference between class and struct is that the member variables of struct can be accessed directly by the object, whereas the member variables of class cannot be accessed directly by the object.

4.4 Access Specifiers

Access specifiers can restrict access to members of a class. This enables classes to hide information from outside world. There are 3 access specifiers provided in C++ namely, public, private and protected. Following subsections describe each specifier with example.

4.4.1 THE public KEYWORD

In Section 4.3, we have noticed that the object directly accesses the member variables of structure whereas the same is not possible with class members. The keyword public can be used to allow object to access the member variables of class directly like structure. The public keyword is written inside the class. It is terminated by colon (:). The member variables and functions declared followed by the keyword public can be accessed directly by the object. The declaration can be done as below:

```
class item          // class declaration
{
public:       // public section begins
int codeno;
float prize;
int qty;
} ;
```

The following program illustrates the use of public keyword with class.

Program 4.3:

```
//Write a program to declare an members of a class as public. Access the elements using object.
# include <iostream.h>
# include <constream.h>
class item
{
public:
int codeno;
float prize;
int qty;
}
int main ( )
{
item one;
one.codeno=123;
```

```
one.prize=123.45;
one.qty=150;
cout<<"\n Codeno ="<<one.codeno;
cout<<"\n Prize = "<<one. prize;
cout<<"\n Quantity = "<<one.qty;
return 0;
}
```

OUTPUT

Codeno = 123

Prize :123.449997

Quantity =150

Explanation: In the above program, the members of class item are declared followed by keyword public. The object one of class item, accesses the member variables directly. The member variables are initialized and values are displayed on the screen.

4.4.2 THE private KEYWORD

The private keyword is used to prevent direct access to member variables or function by the object. Data hiding is nothing but making data variable of the class or struct private. The class by default produces this effect. The structure variables are by default public. To prevent member variables and functions of struct from direct access the private keyword is used. The syntax of private keyword is same as public. The private keyword is terminated by colon. Consider the given example

```
struct item
{
  private:
int codeno;
float prize;
int qty;
}
int main ()
{
item one;
one.codeno=123;
one.price=123.45;
one.qty=150;
}
```

As soon as the above program is compiled, the compiler will display following error messages:

'item::codeno" is not accessible

'item::prize' is not accessible

'item::qty' is not accessible

'item::codeno' is not accessible

'item::prize' is not accessible

From the above discussion, we noticed that by default (without applying public or private keyword) the class members are private (not accessible) whereas the struct members are public (accessible).

The private members are not accessible by the object, directly. To access the private members of a class, member functions of the same class are used. The member functions must be declared in the class in public section. An object can access the private members through the public member function.

4.4.3 THE protected KEYWORD

The access mechanism of protected keyword is same as private keyword. The protected keyword is frequently used in inheritance of classes. Hence, its detailed description is given in Inheritance unit. Table 4.2 illustrates the access difference between private, protected and public keywords.

Table 4.2: Access limits of class members

Access specifies	Access permission	
	Class members	Class object
public	allowed	allowed
private	allowed	disallowed
protected	allowed	disallowed

4.5 Characteristics of Class Members

There can be two types of members in a class namely, variable and function. The above access specifiers can be applied to any of them. The following properties distinguish members of class from other functions and variables.

(1) The difference between normal and member variable/ function is that the former can be invoked freely where as the latter can only by using an object of the same class.

(2) Member variable/ function can not be used in more than one class. This is possible because the scope of the function is limited to their classes and cannot overlap one another.

(3) The private variable or private function can be accessed by public member function of the same class. Outside functions have no access permission.

(4) The member functions of the same class can invoke one another without using any object or dot operator.

While variables in the class can be defined the same way as normal variable, the functions can be defined in more than one way as described below:

4.5.1 Defining Member Function

The member function must be declared inside the class. They can be defined in a) private or public section b) inside or outside the class. Normally small member functions are defined inside the class and large functions are defined outside the class.

If function is defined outside the class, its prototype declaration must be done inside the class. While defining the function, scope access operator and class name should precede the function name. The following program illustrates everything about member functions and how to access private member of the class.

Member Function inside the Class

Member function inside the class can be declared in public or private section. To execute private member function, it must be invoked by public member function of the same class. A member function of a class can invoke any other member function of its own-class. This method of invoking function is known as nesting of member function. The member function can be invoked by its name (and parameters if any) terminated by semicolon only like normal function. The following program illustrates the use of member function inside the class in public as well as in private section.

PROGRAM 4.4

```
// Write a program to declare private member function and access it using public member function.
# include <iostream.h>
# include <constream.h>
class item
 {
  private :
int codeno;
float prize;
int qty;
  void values ()
  {
codeno=125;
prize=195;
qty=200;
   }
 Public:
 void show()
 {
values();
cout<<"\n Codeno ="<<codeno;
cout<<"\n Prize = "<< prize;
```

```
cout<<"\n Quantity = "<<qty;
  }
};
 int main ()
 {
      Item one;

    //one.values();              //not accessibal
one.show();                      // call the public member function
return 0;
  }
```

OUTPUT

Codeno = 125

Prize:195

Quantity =200

Explanation: In the above program, the private section of a class item contains one-member function values(). The function show() is defined in public section. In function main(), one is an object of class item. The object one cannot access the private member function. In order to execute the private member function, the private function must be invoked using public member function. In this example, the public member function show() invokes the private member function values(). In the invocation of function values(), object name and operator are not used.

Member Function outside the Class

In the previous examples, we observed that the member functions are defined directly inside the class. The function prototype is not declared for that. The functions defined inside the class are considered as inline function. When a function is declared as inline, the compiler copies the code of the function in the calling function i.e. function body is inserted in place of function call during compilation. If a function is small, it should be defined inside the class and if large it must be defined outside the class. To define a function outside the class the following care must be taken:

(1) The prototype of function must be declared inside the class.

(2) The function name must be preceded by class name and its return type separated by scope access operator.

The following example illustrates the function defined outside the class.

```
# include <iostream.h>
# include <constream.h>
class item
{
  private :
int codeno;
float prize;
```

```
int qty;
 public :
   void show();
 };
void item::show()
{
codeno=101;
prize=2342;
qty=122;
cout<<"\n Codeno =">>codeno;
cout<<"\n Prize = "<< prize;
cout<<"\n Quantity = "<<qty;
}
int main ()
{
item one;
one.show();                    // call the public member function
return 0;
}
```

OUTPUT

```
Codeno = 101
Prize:2342
Quantity =122
```

Explanation: In the above program, the prototype of function show() is declared inside the class and followed by it class definition is terminated. The body of function show() is defined inside the class. Class name that it belongs to and its return type, precede the function name. The function declared of function show() is as follows:

```
void item:: show ()
```

Here, void is return type i.e. function is not returning a value. The item is a class name. The scope access operator separates the class name and function name and the body of function is defined after it

Outside Member Function Inline

It is a good practice to declare function prototype inside the class and definition outside the class. The inline mechanism reduces overhead relating to accessing the member function. It provides better efficiency and allows quick execution of functions. An inline member function is similar to macros. Call to inline function in the program, puts the function code in the caller program. This is known as inline expansion. Inline functions are also called as open subroutines because their code is replaced at the place of function call in the caller function. The normal functions are known as closed subroutines because when such functions are called, the control passes to the function.

By default, all member functions defined inside the class are inline functions. The member function defined outside the class can be made inline by prefixing the keyword inline to function declaration as shown in Figure 4.3.

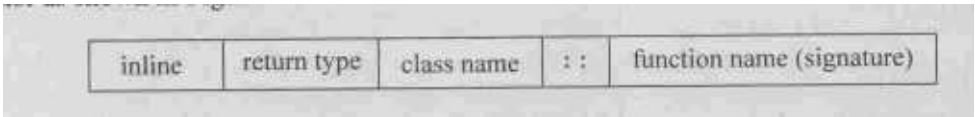


Figure 4.3 Inline function outside the class

The inline is a keyword and acts as function qualifier. The return type is functions return type i.e., the function returns values of this type. The class name is the name of class that the function belongs to. Scope access operator separates class name and function name. Signature means argument list passed function. The following program illustrates inline function outside the class

Program 4.5:

Write a program to declare outside function inline

```
# include <iostream.h>
# include <constream.h>
class item
{
  private :
int codeno;
float prize;
int qty;
 public :
void show(void);
};

inline void item::show()
{
codeno=213;
prize=2022;
qty=150;
cout<<"\n Codeno =">>codeno;
cout<<"\n Prize = "<< prize;
cout<<"\n Quantity = "<<qty;
}
int main ()
{
item one;
```

```
one.show();                // call the public member function
return 0;
}
```

OUTPUT

Codeno = 213

Prize:2022

Quantity =150

Explanation: The above program is the same as the last one. The only difference is that the function show() is defined as inline outside the class. The function declared is inline void item:: show().

4.5.2 Memory Usage by Members

Objects are the identifiers declared for class data type. Object is a composition of one or more variables declared inside the class. Each object has its own copy of public and private member variables. An object can have access to its own copy of member variables and have no access to member variables of other objects.

Declaration of class allocates memory to its member functions only not to its member variables. Memory is reserved for member variables When an object is declared..

Consider the following program.

Write a program to declare object and display their contents.

```
#include<conio.h>
#include<iostream.h>
class month
{
 public :
char *name;
int days;                  //end of the class
 };
int main()
{
month m1,m3;
m1.name="JANUARY";
m1.days=31;
m3.name="MARCH";
m3.days=31;
cout<<"\n Object m1 ";
cout<<"\ Month Name  : "<<m1.month<<" Address :"<<(unsigned)&m1.name;
cout<<\n Days  : " <<m1.days <<"\t\t Address:"<<(unsigned)&m1.days;
```

```
cout<<"\n Object m3 ";
cout<<"\ Month Name  : "<<m3.month <<" Address :"<<(unsigned)&m3.name;
cout<<\n Days  : " <<m3.days <<"\t\t Address:"<<(unsigned)&m3.days;
return 0;
}
```

OUTPUT

Object m1

Month name : JANUARY Address : 65522

Days : 31 Address =65524

Object m3

Month name : MARCHNUARY Address : 65518

Days : 31 Address =65520

Explanation: Ml and M3 are objects of class month. Separate memory is allocated to each object. The contents and address of the member variables are displayed in the output. Figure 4.4 shows it more clearly.

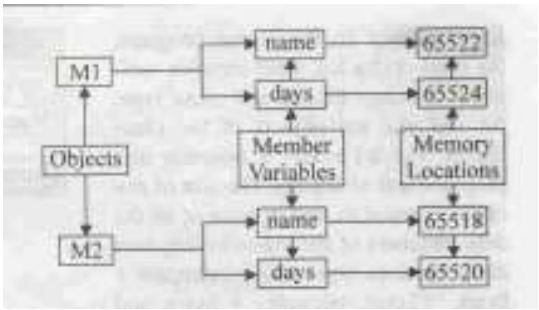


Figure 4.4 Memory occupied by objects

From the last program it is clear that memory is allocated to member variables. What about functions? Member functions are created and memory is allocated to them only once when a class is declared. All objects of a class access the same memory location where member functions are stored. Hence, separate copies of member functions are not present in every object like member variables. The following program and Figure 4.5 illustrates this.

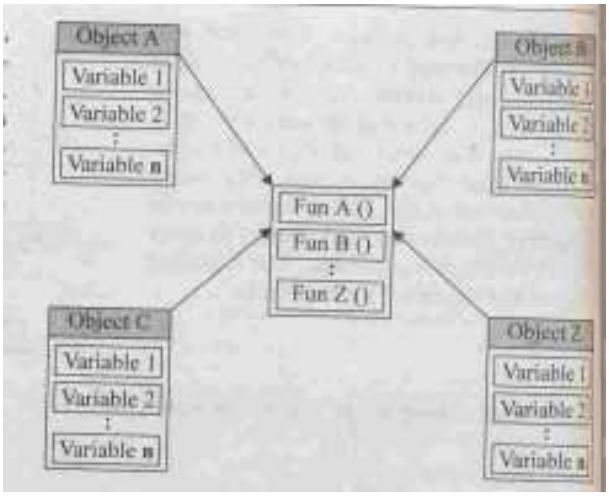


Figure 4.5 Member variables and member functions in memory

Write a program to display the size of the object

```
#include<iostream.h>
#include<conio.h>
class data
{
long i;
float f;
char c;
};
int main()
{
data d1,d2;
cout<<"\n the size of object d1 = <<sizeof(d1);
cout<<"\n the size of object d2 = <<sizeof(d2);
cout<<"\n the size of class = <<sizeof(data);
return 0;
}
```

OUTPUT

```
Size of object d1 = 9
Size of object d2 = 9
Size of class     = 9
```

Explanation: In the above program, the class data has three member variables of long, float and char type. dl and d2 are objects of the class data. The sizeof () operator displays the size of objects. The size of any object is equal to sum of sizes of all the data members of the class. In the class data, the data type long occupies 4 bytes, float occupies 4 bytes and char occupies 1 byte. Their sum is 9 that is the size of an individual object.

The member functions are not considered in the size of the object. All the objects of a class use the same member functions. Only one copy of member function is created and stored in the memory whereas each object has its own set of data members

Self Learning Exercises

1. Is it possible for a member function of a class to activate another member function of the same class?

A. No.

B. Yes, but only public member functions.

C. Yes, but only private member functions.

D. Yes, both public and private member functions can be activated within another member function

2. Can two classes contain member functions with the same name?
   A. No.
   B. Yes, but only if the two classes have the same name.
   C. Yes, but only if the main program does not declare both kinds
   D. Yes, this is always allowed.
3. What is the primary purpose of a default constructor?
   A. To allow multiple classes to be used in a single program.
   B. To copy an actual argument to a function's parameter.
   C. To initialize each object as it is declared.
   D. To maintain a count of how many objects of a class have been created.
4. A constructor is called whenever
   A. An object is declared
   B. An Object is used
   C. A class is declared
   D. A class is used

4.5.3 Static Member Variables

We have noticed earlier that each object has its separate set of data member variables in memory. The member functions are created only once and all objects share the functions. No separate copy of function of each object is created in the memory like data member variables.

It is possible to create common member variables like function using the static keyword. One a member variable is declared as static, only one copy of that member is created for the whole class. The static is a keyword used to preserve value of a variable. When a variable is declared static, it is initialized to zero. A static function or variable is only recognized inside the scope of the present class:

Syntax:

static <variable definition> ;

static <function definition>;

If a local variable is prefixed with static keyword, it preserves the last value of the variable. A static variable is helpful when all the objects of the class share a common data. The static data variable is accessible within the class, but its value remains in the memory throughout the whole program. Figure 4.6 shows static members in memory.

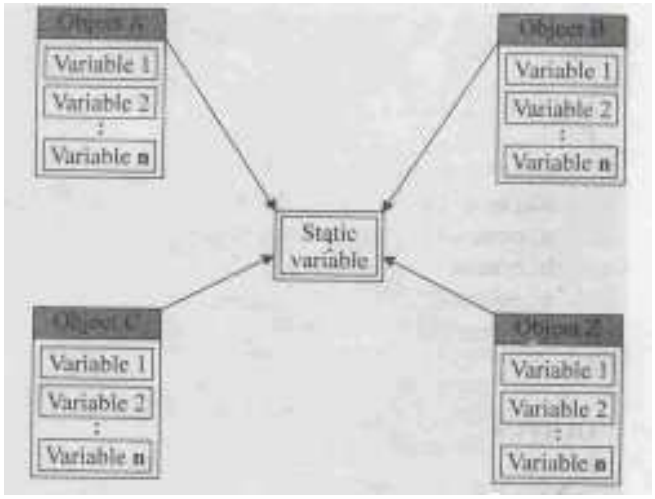


Figure 4.6 Static Members in memory

Examples:

static int C;

static void display () {}

(a) int sumnum :: C=0;

The class and scope of the static member variable is defined outside the class declaration as shown in above statement. The reasons are:

(1) The static member variables are associated with the class and not with any object.

(2) The static member variables are stored individually rather than an element of an object.

(3) The static member variables must be initialized otherwise the linker will generate an error.

(4) The memory for static member variable is allocated only once.

(5) Only one copy of static member variable is created for the whole class for any number of objects. All the objects have common static member variables.

Write a program to declare static member variable. display the value of static member variable.

```
#include<conio.h>
#include<iostream.h>
class number
{
static int c;
public :
void count()
{
++c;
cout<<"\n c = "<<c;
}
};
int number :: c=0;
int main()
{
number a,b,c;
a.count();
b.count();
c.count();
return 0;
}
```

OUTPUT

C=1

C=2

C=3

Explanation: In the above program, the class number has one static member variable c. The count () is a member functions increment value of static member variable c by one when called. The statement int number:: c=O initializes the static member with 0. It is also possible to initialize the static member variables with other values. In the function main (), a, b and c are three objects of class number. Each object calls the function count (). At each call to the function count () the variable c gets incremented and the cout statement displays the value of variable c. The objects a, b and c share the same copy of static member variable c.

Write a program to show difference between static and non static member variables.

```
#include<conio.h>
#include<iostream.h>
class number
{
static int c;
      int k;
public :
void zero()
{
k=0;
}
void count()
{
 ++c ;
++k;
cout<<"\n value of c = "<<c<<" Address of c = "<<(unsigned)&c;
cout<<"\n value of k = "<<k<<" Address of k = "<<(unsigned)&k;
}
}
int number :: c=0;          // initialization of static member variable
int main()
{
number A,B,C;
A.zero();
B.zero();
C.zero();
A.count();
B.count();
return 0;
}
```

OUTPUT

Value of c =1 Address of c = 11138

Value of k =1 Address of c = 65524

Value of c =2 Address of c = 11138

Value of k =1 Address of c = 65522

Value of c =3 Address of c = 11138

Value of k =1 Address of c = 65520

Explanation: This program compares between static and non-static member variables. The class number has two-member variables c and k. The variable c is declared as static and k is a normal variable. The function zero () is used to initialize the variable k with zero. The static member variable c is initialized with zero as shown below:

int number :: c=0 // initialization of static member variable

The function count () is used to increment values of c and k. In function main (), A, B and C are objects of class number. The function zero () is invoked three times by object A, B and C. Each object has its own copy of variable k and hence, each object invokes the function zero () to initialize its copy of k. The static member variable c is common among the objects A, B and C. Hence, it is initialized only once. Figure 4.7 shows the object and member variables in memory.

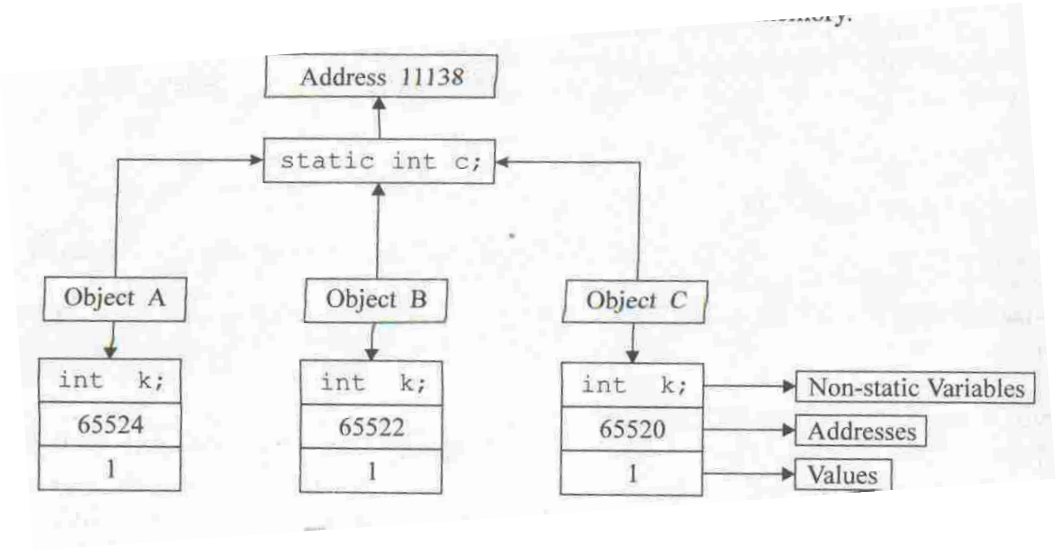


Figure 4.7 Static and non-static members

The static public member variable can also be initialized in function main () like other variables. The static member variable using class name and scope access operator can be accessed. The scope-accessed operator is also used when variables of same name are declared in global and local scope. The following program illustrates this

Write a program to declare static public variable , global and local variable with the same name. Initialize and display their contents.

#include<conio.h>

#include<iostream.h>

class bita

```
{
public :
static int c;
}
int bita :: c=22;
void main()
{
int c=33;          //local variable;
cout<<"\n Class member  c = "bita::c;
cout<<"\n Global Variable  c = "::c;
cout<<"\n Local member  c = "bita::c;
}
```

OUTPUT

Class Member c =22

Global Variable c=11

Local variable c=33

Explanation: In the above program, the variable C is declared and initialized in three different scopes such as global, local and inside the class. The variable c declared inside is static variable and initialized to 22. The global variable c is initialized to 11 and local variable c is initialized to 33.

Static Member Variable The value of static variable is displayed using variable name preceded~ class name and scope access operator as shown in the statement

Cout<<" \n Class member : = "<<bita: :C;

Global Variable The global variable can be accessed using variable name preceded by scope access operator as shown in the statement

cout<<" \nGlobal variable c = "<<:: C ; .

Local Variable The local variable can be accessed only by putting its name as shown in the statement cout <<"\n Local variable c = "<<c;.

4.4.4 Static Member Functions

Like member variables, functions can also be declared as static. When a function is defined as static, it an access only static member variables and functions of the same class. The non-static members are not available to these functions. The static member function declared in public section can be invoked using its class name without using its objects. The static keyword makes the function free from the individual object of the class and its scope is global in the class without creating any side effect for other part of the program. The programmer must follow the following points while declaring static function:

1. Just one copy of static member is created in the memory for entire class. All objects of the class share the same copy of static member.
2. Static member functions can access only static member variables or functions.
3. Static member functions can be invoked using class name.

4. It is also possible to invoke static member functions using objects.

5. When one of the objects changes the value of data member variables, the effect is visible to all the objects of the class.

Write a program to declare static member function and call them from the main() function.

```
#include<conio.h>
#include<iostream.h>
class bita
{
private:
static int c;
public :
static void count ()
{
c++;
}
static void display ()
{
cout<<"\v Value of c :"<<c;
}
};

int bita :: c=0;
void main()
{
bita:: display();
bita::count();
bita::count();
bita::display();
}
```

OUTPUT

Value of c : 0

Value of c : 2

Explanation: In the above program, the member variable c and functions of class bita are static. The function count () when called, increases the value of static variable c. The function display () prints the current value of the variable c. The static function can be called using class name and scope access operator as per statements given next.

```
Bita ::count();          // invokes count() function
Bita ::display();        // invokes display() function
```

Static member function can also be declared in private section. The private static function must be invoked using static public function. The following program illustrates the point.

Write a program to define private static member function and invoke it.

```
#include<conio.h>
#include<iostream.h>
class bita
{
private:
statis int c;
static void count()
{
c++;
}
public :
static void display()
{
count(); // call to private static member function
cout<<"\n value of c :"<<c;
}
};
int bita::c=0;
void main()
{
bita::display();
bita::display();
}
```

OUTPUT

```
Value of c :1
Value of c: 2
```

Explanation: In the above program, count() is a private static member function. The public static function display() invokes the private static function count() . The function display() also displays the value of static variable c.

4.6 Object Usage

Objects can be used in many ways. The following are important usage of objects.

4.6.1 Static Object

It is common to declare variable static that gets initialized to zero. The object is a composition one or more member variables. The keyword static can be used to initialize all class member variables to zero. Declaring object itself as static can do this. Thus, all its associated members get initialized to zero. The following program illustrates full working of static object.

Write a program to declare static object. Display its contents.

```
#include<conio.h>
#include<iostream.h>
class bita
{
private :
int c;
int k;
public :
void plus()
{
c=c+2;
k=k+2;
}
void show()
{
cout<<"\n c = :"<<c<<"\n";
cout<<"k = "<<k;
}
};
void main()
{
static bita A;
A.plus();
A.show();
}
```

OUTPUT

C=2;

K=2

Explanation: The class bita has two member variables c and k and two member functions plus() and show(). In function main(), the object A is declared static. It is also declared as static. The data member of object A gets initialized to zero. The function plus() is invoked, which adds two to the value of c and k. The function displays value of c and k. Declaring objects static, does not mean that the entire class is static including member function. The declaration of static object removes garbage of its member variables and initializes them to zero.

4.6.2 Array Of Objects

Arrays are collection of similar data types. Arrays can be of any data type including user-defined data type, created by using struct, class and typedef declarations. We can also create an array of objects. The array elements are stored in continuous memory locations as shown in Figure 4.8. Con-sider the following example:

```
class player
{
private:
char name [20];
int age;
public:
void input (void) ;
void display (void) ;
} ;
```

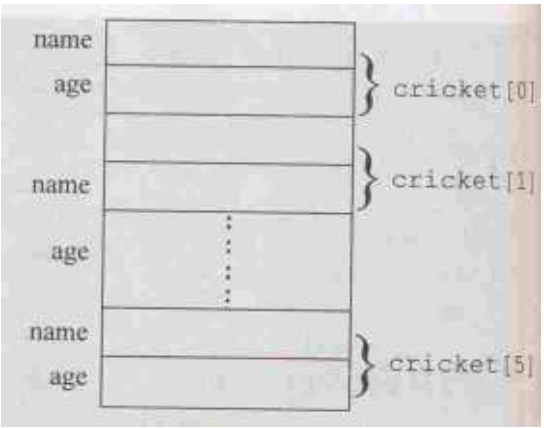


Figure 4.8 Array of Objects

In the example given above player is a user-defined data type and can be used to declare an array of object of type player. Each object of an array has its own set of data variables.

```
player cricket [5] ;
player football [5] ;
player hockey [5] ;
```

As shown above, arrays of objects of type player are created. The array cricket [5] contains name, age and information for five objects. The next two declarations can maintain the same information for other players in arrays hockey [5] and football [5]. These arrays can be initialized or accessed like an ordinary array. The program given below describes the working of array of objects.

Write a program to declare array of objects. Initialize and display the contents of array

```
#include<iostream.h>
#include <constream.h>
class player
{
private:
int age;
public:
input;
void display(void);
} ;
void player:: input ( )
{
cout<<"\n Enter the name :";
cin>>name;
cout <<" Age: ";
cin>>age;
}
void player :: display()
{
cout<<"\n player name : "<<name;
cout<<"\n Age              :<<age;
}
int main ( )
a
player cricket [3] ;
cout <<"\n Enter Name and age of 3 players ";
for (i=0;i<3.;i++)
cricket [i] .display ();
return 0;
}
```

OUTPUT

Enter Name and age of 3 players

Enter Player name : Sachin

Age: 29

Enter Player name : Rahul

Age: 28

Enter Player name : Saurav

Age: 30

Player name : Sachin

Age : 29

Player name : Rahul

Age : 28

Player name : Saurav

Age : 30

Explanation: In the above program, the member function input() reads information of players, The display () function displays information on the screen. In function main() , the statement player cricket [3]; creates an array cricket [3] of three objects of type player. The for loops are used to invoke member functions input () and display(), using array of objects.

4.7 Constructors and Destructors

Declaring static member variables facilitates programmer to initialize member variables with desired values. The drawback of static members is that only one copy of static member is created for entire class. All objects share the same copy, which do not provide security. And the main disadvantage of static object is that its value remains in the memory throughout the program.

It is possible to initialize member variables of object through a member function automatically called when object is created. Such member function is called constructor. C++ run-time arrangement takes care of execution of constructors. When an object is created, constructor is executed. The programmer can also pass values to the constructor to initialize member variables with different values.

Destructor is another member function used to destroy the object. The destructor is executed at the end of the function when objects are of no use or goes out of scope. It is optional to declare constructor and destructor. If the programmer does not define them, implicit constructor and destructors are executed.

Constructors and destructors have many attributes as that of normal member functions. We can declare and define them within the class, or declare them within the class and define them outside, but they have a few unique characteristics.

4.7.1 Constructor Properties

Constructors are special member functions that decide how the objects of a class are created, initialized and copied. They have a few unique characteristics as mentioned below:

(1) Constructor has the same name as that of the class it belongs.

(2) Constructor is executed when an object is declared.

(3) Constructors have neither return value nor void.

(4) Constructor initializes objects and allocates appropriate memory to objects.

(5) Though constructors are executed implicitly, they can be invoked explicitly.

(6) Constructor can have default and can be overloaded.

(7) The constructor without arguments is called as default constructor.

Defining Constructor

The initialization of member variables of class is carried out using constructors. It can be initialized to default values or to initially assigned in the program values or read from keyboard values. The following program illustrates the initialization of member variables read from keyboard using constructor.

Write a program to read values through the keyboard and initialize member variables using constructor.

```
#include<conio.h>
#include<iostream.h>
class num
{
private:
 int a,b,c;
public:
num (void) ;                    // declaration of constructor
void show ( )
{
cout<<"\n"<<" a= "<<a <<" b= "<<b <<" c= "<<c;
};
}
num :: num (void)               // definition of constructor
{
cout<<"\n Constructor called";
cout<<"\n Enter Values for a, b and c : ";
cin>> a>>b>>c;
}
main( )
{
num X;
x.show();
return 0;
}
```

OUTPUT

Constructor called

Enter Values for a, b and c : 1 5 4

a=1 b=5 c= 4

Explanation: In this program, the class num is declared with private member variables a, b and c. This class also has show() function and constructor prototype declaration. The function show() displays the contents of member variables on the screen. Whenever an object is created, the constructor is called and it reads the integer values through the keyboard. Thus, entered constants are assigned to member variables of the class. Here, the constructor is used like other functions.

Parameterized Constructor and Default Arguments

It is also possible to create constructor with arguments and such constructors are called as parameterized constructors. For such constructors, it is necessary to pass values to the constructor when object is created. Consider the example given next.

Write a program to create constructor with arguments and pass the arguments to the constructor.

```
#include<conio.h>
#include<iostream.h>
class num
{
private:
int a,b,c;
public:
num(int m, int j, int k);              //declaration of constructor with arguments
void show()
{
cout<<"\n a= "<<a<<"b = "<<b<<"c ="<<c;
}
num::num(int m , int j, int k)
{
a=m;
b=j;
c=k;
}
main()
{
 num x =num(4,5,7);     //explicit call
 num y(1,2,8)           //implicit call
```

```
x.show();
y.show();
return 0;
}
```

OUTPUT

a=4 b=5 c=7

a=1 b=2 c=8

Explanation: In the above program, x and y are objects of class num. When objects are created, three values are passed to the constructor. These values are assigned to the member variables. The function show() displays the contents of member variables.

It is also possible to declare constructors with default arguments as illustrated through following program.

Write a program to declare default arguments in constructor. Obtain the power of the number.

```
#include<conio.h>
#include<iostream.h>
#include<math.h>
class power
{
  private:
int num;
int power;
int ans;
 public :
    power (int n=9,int P=3) ;      // declaration of constructor with default arguments
   void show ( )
   {
   cout<<"\n"<<num <<" raise to "<<power<<" is "<<ans;
   }
};
  power:: power(int n, int p )
 {
  num=n;
   power=p;
 ans=pow(n,p);
}
 main ( )
 {
```

```
 class power pI, p2 (5) ;
 pl.show ();
p2.show ();
return 0;
}
```

OUTPUT

9 raise to 3 is 729

5 raise to 3 is 125

Explanation: In the above program, the class power is declared. It has three integer member variables and one member function show(). The show() function is used to display the values of member data variables. The constructor of class power is declared with two default arguments namely n with default value 9 and p with default value 3. In the function main (), p1 and p2 are two objects of class power. The p1 object is created without argument and hence, the constructor uses default arguments in pow() function. The p2 object is created with one argument. In this call of constructor, the second argument is taken as default i.e. 3. Both the results are shown in output.

Constructor with Overloading

It is also possible to overload constructors. In previous examples, we declared single constructor with and without arguments. A class can contain more than one constructor. This is known as constructor overloading. All constructors are defined with the same name as the class. All the constructors contain different number of arguments. Depending upon number of arguments, the compiler executes appropriate constructor. The following program illustrates multiple constructors in one class:

Write a program with multiple constructors for the single class.

```
#include,conio.h>
#include<iostream.h>
class num
{
private :
int a;
float b;
char c;
public;
num(int m, float j , char k);
num (int m, float j);
num () ;
void show ( )
 {
cout <<"\n\t a="<<a<<"b= "<<b <<" c= <<c;
```

```
 }
};
 num :: num(int m, float j , char k)
{
cout<<"\n Constructor  with three arguments" ;
a=m;
b=j;
c=k;
}
num:: num (int m, float j)
{
cout <<"\n Constructor with two arguments";
a=m;
b=j;
c=' ';
}
num :; num ( )
{
cout<<"\n Constructor without arguments";
{
          a=b=c=NULL;
}
main ( )
 {
class num x(4,.5.5,'A');
x. show () ;
class num y(1,2.2);
y.show();
class num z;
z.show();
return 0;
}
```

OUTPUT

Constructor with three arguments

a= 4 b= 5.5 c= A

Constructor with two arguments

a=1 b= 2.2 c=

Constructor without arguments

a=0 b=0 c=

Explanation: In the above program, three constructors are declared. The first constructor is with three arguments, second with two and third without any argument. While creating objects, arguments are passed. Depending on the number of arguments the compiler decides which constructor is to be called. In this program x, y and z are three objects created. The x object passes three arguments. The y object passes two arguments and z object passes no arguments. The function show() is used to display the contents of the class members.

Table 4.3 describes object declaration and appropriate constructor to it.

Table 4.3: Overloaded constructors

Constructor declaration	Object declaration
num (int m, float j char k) ; num(int m, float j); num () ;	num x (4,5.5, 'A') ; num y(l, 2 .2) ; num z;

In the above example, statements in first column describe constructor declarations and second column describe corresponding object declarations. The compiler decides which constructor to be called depending on number of arguments present with the object.

When object x is created, the constructor with three arguments is called because the declaration of an object is followed by three arguments. For object y, constructor with two arguments is called and lastly object z, which is without any argument, is candidate for constructor without argument.

Copy Constructors

The constructor can accept arguments of any data type including user-defined data types and an object of its own class.

Statement (a)	Statement (b)

class num
{
private:
- - -
- --
- - -
public:
num (num) ;
 	
}
	class num
	{
	private:
	
	- - -
	- --
	- --
	public:
	num (num&) ;
	}

In statement (a) an argument of the constructor is same as that of its class. Hence, this declaration is wrong. It is possible to pass reference of object to the constructor. Such declaration is known as copy constructor. The statement (b) is valid and can be used to copy constructor.

When we pass an object by value into a function, a temporary copy of that object is created. All copy constructors require one argument, with reference to an object of that class. Using copy constructors, it is possible for the programmers to declare and initialize one object using reference of another object. Thus, whenever a constructor is called a copy of an object is created.

Write a program to pass an object with reference to constructor. Declare and initialize other objects.

```
#include <iostream.h>
# include <conio.h>
class num
{
int n;
public:
num() { }                    // constructor without argument
num (int k)                  // constructor with one argument
 {
n=k;
}
num (num &j)
{
n=j.n;
}
void show(void)
{
cout<<n;
}
};
main()
{
numJ(50);
numK(J);
numL=J;
num M;
M=J;
Cout<<"\n Object J Value of n :";
J.show();
Cout<<"\n Object K Value of n :";
K.show();
```

```
Cout<<"\n Object L Value of n :";
L.show();
Cout<<"\n Object M Value of n :";
M.show();
}
```

4.7.2 Destructor Usage

Destructors are opposite to the constructor. The process of destroying the class objects created by constructors is done in destructor. The destructors have the same name as their class, preceded by a tilde (~). A destructor is automatically executed when object goes out of scope. It is also invoked when delete operator is used to free memory allocated with class pointer. Like constructor, it is not possible to define overloaded destructor and passing arguments to them. The class can have only one destructor. Destructors are called when these objects go out of scope.

The properties of destructor are described below:

1. Destructor has the same name as that of the class it belongs to and preceded by ~ (tilde).
2. Like constructor, the destructor does not have return type and not even void.
3. Constructor and destructor cannot be inherited, though a derived class can call the constructors and destructors of the base class.
4. Destructors can be virtual, but constructors cannot.
5. Only one destructor can be defined in the destructor. The destructor does not have any argument.
6. The destructors neither have default values nor can be overloaded.
7. Programmer cannot access address of destructor.
8. Constructors and destructors can make implicit calls to operators new and delete if memory allocation or de-allocation is needed for an object.
9. An object with a constructor or destructor cannot be used as a member of a union.

For local and non-static objects, the destructor is executed when the object goes out of scope. In case the program is terminated by using return or exit() statements, the destructor is executed for every object existing at that time. It is not possible to define more than one destructor. The destructor is only one way to destroy the object. Hence, they cannot be overloaded.

A destructor neither requires any argument nor returns any value. It is automatically called when object goes out of scope. Destructor releases memory space occupied by the objects.

The program given below explains the use of destructor.

Write a program to demonstrate execution of constructor and destructor.

```
#include<conio.h>
#include<iostraeam.h>
class text
{
text();              //constructor
{
```

```
cout<<"\n Constructor executed :";
}
~text();            //Destructor
{
cout<<"\n Destructor executed :";
}

};
void main()
{
text t;             //object declaration ;
}
```

OUTPUT

Constructor executed

Destructor executed

Explanation: In the above program, the class text contains constructor and destructor. In function main(), object t is executed. When object t is declared, constructor is executed. When object goes out of scope, destructor is executed.

Write a program to create an object and release them using destructor.

```
# include <iostream.h>
#include<conio.h>
int c=O;            // counter for counting objects created and destroyed.
class num
{
public:
num ( )
{
c++;
cout <<"\n Object Created  : Object("<<c <<")";
}
~num ( )
 {
cout «"\n Object Released: Object(<<c <<")
C—;
}

};
```

```
main()
{
cout<<"\n In main() \n";
num a,b;
cout<<"\n\n in Block A \n";
{
class num c;
}
cout<<"\n \n Again In main()\n";
return 0;
}
```

OUTPUT

Object Created " Object (1)

Object Created " Object (2)

In Block A

Object Created " Object (3)

Object Created " Object (3)

Again In main()

Object Created " Object (2)

Object Created " Object (1)

Explanation: In the above program, the variable c is initialized with zero. The variable c is incremented in constructor and decremented in destructor. When objects are created the compiler calls the constructor. Objects are destroyed or released when destructor is called. Thus, the value of a variable changes as the constructors and destructors are called. The value of a variable of c shows number of objects created and destroyed. In this program the object a and b are created in main () . The object c is created in another block i.e., in block A. The object c is created and destroyed in the same block. The objects are local to the block in which they are defined. The object a and b are released as the control passes to main () block. The object created last is released first.

4.8 Summary

In structure of C or C++, one or more dissimilar data types can be combined and a new custom data type can be created.

In C++, classes and structures contain member variables and member functions in their declarations with private and public access blocks that restrict the unauthorized use. The defined classes and structures further can be used as custom data type in the program to declare objects.

In C++, private and public are two new keywords. The private keyword is used to protect specified data and functions from illegal use whereas the public keyword allows access permission

The member function can be defined as (a) private or public (b) inside the class or outside the class.

To access private data members of a class, member functions are used.

The difference between member function and normal function is that the normal can be invoked freely whereas the member function can be invoked only by using the object of the same class.

static is the keyword used to preserve value of a variable. When a variable is declared as static, it is initialized to zero. A static function or data element is only recognized inside the scope of the present class.

When a function is defined as static, it can access only static member variables and functions of the same class. The static member functions are called using its class name without using its objects.

C++ provides a pair of in-built functions called constructors and destructors. The compiler automatically executes these functions. When an object is created constructor is executed. The programmer can also pass values to the constructor to initialize member variables with different values. The destructor destroys the object. It is executed at the end of program when objects are of no use.

Constructors and destructors decide how the objects of a class are created, initialized, copied, and destroyed. They are member functions. Their names are distinguished from all other member functions because they have the same name as the class they belong to.

It is also possible to create, overload and assign default arguments to constructor like normal functions.

When we pass an object by value into a function, a temporary copy of that object is created. All copy constructors require one argument, with reference to an object of that class. Using copy constructors, it is possible for the programmers to declare and initialize one object using reference of another object. Thus, whenever a constructor is called a copy of an object is created.

The compiler automatically calls the constructor and destructor. We can also call the constructor and destructor in the same fashion as we call the normal user-defined function. The calling methods are different for constructors and destructors.

4.9 GLOSSARY

- OOP Object Oriented Programming
- A **class** is a user defined type; for example, **string** is a **class**.
- The **class membership** operator, **::**, indicates which **class** a function belongs to. Thus, the full name of the default constructor for the **string class** is **string::string()**.
- **Class scope** describes the visibility of member variables: that is, those that are defined within a **class**. These variables can be accessed by any member function of that **class**; their accessibility to other functions is controlled by the access specifier in effect when they were defined in the **class** interface.
- A **constructor** is a member function that creates new objects of a (particular) **class** type. All constructors have the same name as the **class** for which they are constructors; therefore, the constructors for the **string class** also have the name **string**
- A **destructor** is a member function that cleans up when an object expires; for an object of the **auto** storage class, the destructor is called automatically at the end of the function where that object is defined.

4.10 Further Readings

1. Introduction To Object Oriented Programming And C++ ,Kanetkar , BPB,**ISBN:** 8176568635
2. A complete Reference to C/C++,Herbert Shildt
3. C++: An introduction to programming by Jense Liberty Tim Keogh: BPB Publications, New Delhi
4. OO Programming in C++ by Robert Lafore: , Galgotia Publications Pvt. Ltd., Daryaganj,New Delhi
5. Object Oriented Programming Using C++, Sanjeev Sofat, Cyber Tech. Publication, New Delhi
6. Object Oriented Programming in C++ by E. Balaguruswamy, TMH Publishing Co. Ltd.,New Delhi

4.11 Answers to Self Learning Exercises

1. B 2. D
3. C 4. A

4.12 Unit End Questions

(1) Explain class and struct with their differences.

(2) Which operators are used to access members?

(3) Explain the uses of private and public keywords. How are they different from each other?

(4) Explain features of member functions.

(5) What are static member variables and functions?

(6) How are static variables initialized? Explain with the statement.

(7) How are static functions invoked?

(8) List the keywords terminated by colon with their use.

(9) Can member functions be private?

(10) What is the concept of data hiding? What are the advantages of its applications?

(11) What are static objects?

(12) What is the difference between object and variable?

(13) What are constructors and destructors?

(14) Explain the characteristics of constructors and destructors?

(15) Explain constructors with arguments. How are arguments passed to the constructor?

(16) What do you mean by overloading of constructors? How does it benefit theprogrammer?

(17) Explain constructor with default arguments?

(18) What is copy constructor?

(19) What is static object? How is it different from normal object?

(20) How are private constructors and destructors executed?

UNIT - V

Function and Operator overloading

Structure of the Unit

5.0 Objective

After completion of this chapter, you will be able to understand

- Concept of function overloading
- Precaution with function overloading
- Some library functions
- Rules for operator overloading
- Overloading of unary and binary operators
- Overloading of stream operators

5.1 Introduction

Overloading of constructors was discussed in previous unit. It is possible to extend the concept of overloading to member functions and operators. A meaningful name to a member function is always preferred. Sometimes a similar manipulation of different types of member variables is required. For similar manipulation, preferred member function name should be the same. Similarly, same operator can be used for similar operation on different data types. This can be possible through function and operator loading. This unit describes the principles and precautions for function overloading followed by fundamentals and rules of operator overloading.

5.2 Function Overloading

It is possible in C++ to use the same function name for a number of times for different intentions. Defining multiple functions with same name is known as function overloading or function polymorphism. Polymorphism means one function having many forms. The overloaded function must be different in its argument list and with different data types. The examples of overloaded functions are given below. All the functions defined should be equivalent to their prototypes.

```
int sqr (int);
float sqr (float);
long sqr (long);
```

Program 5.1:

Write a program to calculate square of an integer and float number.

Define function sqr () and use function-overloading concept.

```
# include <iostream.h>
# include <conio.h>
int sqr(int);
float sqr(float);
 main ()
{
int a=15;
float D=2.5;
cout <<"Square = "<<sqr(a) <<"\n";
cout <<"Square "<< sqr(b) <<"\n";
return 0;
```

```
}
int sqr(int  s)
 {
   return  (s*s);
 }

float sqr (float j)
 {
  return (j *j);
 }
```

OUTPUT

Square =225

Square=6.25

Explanation: In the above program, the function sqr () is overloaded for integer and float. In the first call of the function sqr (), an integer 15 is passed. The compiler executed integer version of the function and returns result 225. In the second call, a float value 2.5 is passed to function sqr (). In this call, the compiler executed the float version of the function and returns the result 6.25. The selection of function to execute is made at run- time by the compiler according to the data type of variable passed.

5.2.1 Principles Of Overloading

This section describes principles of function overloading.

(1) For overloading, two functions should have same name but different number of arguments or different type of arguments or both. Return type could be same or different. They must not have same type and number of arguments. For example,

(a) sum (int, int, int);

sum (int, int);

Here, the above function can be overloaded. Though data type of arguments in both the functions are similar but number of arguments are different.

(b) sum (int, int, int);

sum (float, float, float);

In the above example, number of arguments in both the functions is same, but data types are different. Hence, the above function can be overloaded.

(2) Passing constant values directly instead of variables also result in ambiguity. For example,

int sum (int, int, int);

float sum (float ,float ,float);

Here, sum () is an overloaded function for integer and float values. If values are passed as follows

sum (2 , 3, 4);

sum (1.1, 2.3 , 4.3);

Compiler will flag an error because the compiler cannot distinguish between these two functions. Here, internal conversion of float to int is possible. Hence, in both the above calls, only one version of function sum () may be executed.

To overcome this problem, the user needs to do following things:

(i) Declare prototype of all overloaded functions before function main ().

(ii) Pass arguments using variables as follows:

sum (a,b); // a and b are integer variables

sum (e,r,t,y); // e,r,t and y are float variables

(3) The compiler attempts to find an accurate function definition that matches in types and number of arguments and invokes that function. The arguments passed are checked with all declared functions. If matching function is found then that function gets executed.

(4) If there is no accurate match found, compiler makes the implicit conversion of actual argument. For example, char is converted to int and float is converted to double. If all above steps fail then compiler performs user built functions.

The following example illustrates this point.

Program 5.2 :

Write a program to overload a function and create a situation such that the compiler does integral conversion.

```
# include <conio.h>
#include<iostream.h>
int sqr (int);
float sqr (double );
main ( )
{
int a=15;
float b=3.5;
cout<<" Square =" <<sqr('A')<<"\n";
cout<<"Square = "<<sqr(b)<<"\n";
return 0;
}
int sqr(int s)
{
return (s*s);
}
float sqr (double j)
{
return (j*j);
```

```
}
```

OUTPUT

Square = 4225

Square = 12.25

Explanation: In the above program, the function sqr () is overloaded. The first prototype of function sqr (int) is proposed for integer variable and second for double variable. In function main (), the 'A' character data type value is passed to the function sqr (). The compiler invokes the integer version of the function sqr () because the char data type is compatible with integer. In the second call, afloat value is passed to the function sqr (). This time the compiler invokes the double data type version of function sqr ().

When accurate match is not found the compiler makes internal conversion as seen in the above example. C++ automatically attempts to convert the arguments used to call a function into the type of arguments expected by the function.

(5) If internal conversion fails user-defined conversion is carried out with implicit conversion and integral promotion. The user-defined conversion is used with class objects.

(6) Sometime while making internal conversion, ambiguity is created if one data type is compatible with two or more data types. If such situation occurs, the compiler displays an error message.

Consider the following example in which two versions of sqr () are given. One is for long data type and other for double data type.

```
long sqr ( long);
double sqr (double);
sqr (10);                    // function call
```

If an integer value is passed to the function sqr (), the compiler will fall in confusion about the version to be executed and will result in an error message "Ambiguity between 'sqr (long)' and 'sqr(double)'".

(7) If a program has two versions of functions i.e., one for float and second for double data type, then if we pass a float number, the double version of function is selected for execution. But the same is not applicable with int and long int.

Program 5.3 :

Write a program to define overloaded function add () for int and float and perform the addition.

```
# include <conio.h>
#include<conio.h>
int add (int , int, int);
float(float, float, float);
int main ( )
{
float fa, fb, fc, fd;
int ia, ib, ic, id;
cout <<"\n Enter values integer values for ia, ib and ic : ";
```

```
cin>>ia>>ib>>ic;
cout <<"\n Enter float values fqt fa,fb and fc :";
cin »fa »fb »fc;
id=add (ia,ib,ic);
fd=add(fa,fb,fc);
cout <<"\n Addition :"<<id;
cout <<"\n Addition : "<<fd;
return 0;
}
add ( int j, int k, int l)
 {
return (j+k+l);
 }
float add (float a, float b, float c )
 {
 return (a+b+c);
}
```

OUTPUT

Enter values integer values for ia,ib and ic : 1 2 4

Enter float values for fa,tb and fc : 2.2 3.1 4.5

Addition: 7

Addition :9.8

Explanation: In the above program, two versions of function add () are declared. First version is declared for int values and second for float values. Three integer and float values are entered through the keyboard. According to the data type the compiler executes appropriate function. The prototype of functions should be written before main () function. If we write the prototype inside the main (), the float version of the function will not be executed.

5.2.2 Precautions With Overloading

Function overloading is a powerful feature of C++. But, this facility should not be overused. Otherwise it becomes an additional overhead in terms of readability and maintenance. Following precautions must be taken:

(1) Only those functions that basically do the same task, on different sets of data, should be overloaded. The overloading of function with identical name but for different purposes should be avoided.

(2) In function overloading, more than one function has to be actually defined and each of these occupy memory.

(3) Instead of function overloading, using default arguments may make more sense and fewer overheads.

(4) Declare function prototypes before main () and pass variables instead of passing con-

stant directly. This will avoid ambiguity that frequently occurs while overloading functions.

5.3 Operator Overloading

Operator overloading is one of the most valuable concepts introduced by C++ language. It is a type of polymorphism. Polymorphism permits to write multiple definitions for functions and operators. The concept of operator overloading is quite similar to that of function overloading. An operator is a symbol that indicates an operation. It is used to perform operation with constants and variables. Without an operator, programmer cannot build an expression.

C++ frequently uses user-defined data types, which is a combination of one or more basic data types. C++ has an ability to treat user-defined data type like the one they were built-in type. User-defined data types created from class or struct are nothing but combination of one or more variables of basic data types. The compiler knows how to perform various operations using operators for built-in types. However, for the objects those are instance of the class, the operation routine must be defined by the programmer.

5.3.1 Overloading Fundamentals

In traditional programming languages, the operators such as +, -, <=, >= etc., can only be used with basic data types such as int or float etc. The (+) plus operator can be used to perform addition of two variables, but the same is not applicable for objects. The compiler cannot perform addition of two objects. The compiler would throw an error if addition of two is carried out. The compiler must be made aware about addition process of two objects. When an expression including operation with objects is encountered, a compiler searches for the definition of the operator, in which a code is written to perform operation with two objects. Thus, to perform operation with objects, we need to redefine the definition of various operators. For example, for addition of objects A and B, we need to define operator (+) plus. Redefining the operator plus does not change its natural meaning. It can be used for both variables of built-in data types as well as objects of user-defined data types.

C++ has number of standard data types like int, float, char etc. The operators +, -, * and = are used to carry operations with these data types. Operator overloading helps programmer to use these operators with the objects of classes. The outcome of operator overloading is that objects can be used in a natural manner as the variables of basic data types. Operator overloading provides the capability to define the language in which working operator can be changed.

Consider an example,

```
a=c+d;
c=a-d;
```

Where a, b, c and d are variables of basic data types like int or float. The use of operators +, - and = is valid. However, if we try these operators with the object, the compiler displays error message “illegal structure operation”.

```
class number
{
```

```
  public :
int x;
int y;
};
```

For example, A, B and C are three objects of class number. Each object holds individual copy of member variable X and Y. We want to perform addition of A and B and store the result in C. For the sake of understanding the member variables are declared in public section. The addition of A and B means addition of member variables of A and member variables of B. The result of this operation will be stored in member variables of C. This feature can be implemented as shown below:

Program 5.4 :

Write a program to perform addition of two objects and result in the third object. Display contents of all the three objects.

```
#include<conio.h>
#include<iostream.h>
class number
{
 public:
int x;
int y;
number() {              }       //zero argument constructor
number (int j, int k)   //two argument constructor
void show()
 {
cout<<"\n x= "<<x<<"y = "<<y;
  }
};
void main()
{
 number A(2,3), B(4,5), C;
 A.show();
 B.show();

 C.x=A.x+B.x;           //addition of two constructor
 C.y=A.y+B.y;           //using member variable directly
 C.show();
}
```

OUTPUT

X=2 y=3

X=4 y=5

X=6 y=8

Explanation: In the above program, A, B and C are objects of class number. Using constructor, objects are initialized. Consider the following statements:

C.x=A.x+B.X;

C.Y=A.Y+B.Y;

In the above statements, addition of members of objects A and B is performed and stored in C. Each member variable is accessed individually and stored in member variable of C. For example, member x of A and member x of B are added and stored in x of C. Similarly, addition of other members is carried out.

In this program we cannot perform the operation C=A+B. The operation with objects is complicated because it involves operation of one or more data member variables which are part of objects.

5.3.2 THE KEYWORD Operator

The keyword operator defines a new action or operation to the operator.

Syntax:

```
Return type operator    operator symbol          (parameters)
{
statement1;
statement2;
}
```

The keyword 'operator', followed by an operator symbol, defines a new (overloaded) action of the given operator.

Example:

```
number operator + (number D)
{
number T;
T.x=x+D.X;
T.y=y+D.Y;
return T;
}
```

Overloaded operators are redefined within a C++ class using the keyword operator followed by an operator symbol. When an operator is overloaded, the produced symbol is called the operator function name. The above declarations provide an extra meaning to the operator. Operator functions should be either member functions or friend functions. A friend function requires one argument for unary operators and two for binary operators. The member function requires one argument for binary operators and no argument for unary operators. When the member function is called, the calling object is passed implicitly to the function and hence available for member function. While using friend functions, it is essential to pass the objects by value or reference. The

prototype of operator functions in classes can be written as follows:

```
void operator ++();
void operator - ();
num operator+ (num);
friend num operator * (int , num);
```

Operator overloading can be carried out in the following steps:

(a) Define a class which is to be used with overloading operations.

(b) Public declaration section of the class should contain prototype of the function operator ().

(c) Define the operator () function with proper operations for which it is declared.

Program 5.5:

Write a program to illustrate the addition of two objects using operator overloading.

```
#include<conio.h>
#include<iostream.h>
class number
{
 public:
int x;
int y;
 number()       //default constructor
 {
  }
 number (int j , int k)
 {
  x=j;
  y=k;
 }
 number operator+(number D)
 {
  number T;
  T.x=x+D.x;
  T.y = y+D.y;
  return D;
 }
 void show()
 {
 cout<<"\n  x= "<<x<<"y= "<<y;
```

```
  }
};
 void main()
 {
    number A(2,3),B(4,5),C;
     A.show();
     B.show();
      C=A+B;
    C.show();
  }
```

OUTPUT

X=2 y=3

X=4 y=5

X=6 y=8

Explanation: In the above program, A, B and C are objects of class number. Here, the addition has been performed using statement C=A+B. Remember, in the last program we were not able to execute this statement. Instead of this, two separate statements were used to perform addition.

In this program, the statements that perform addition operation of each individual member of objects are written in function operator. The operator has return type and single argument. It also uses a local object T to hold addition as long as the operator function is active. Whenever the statement C=A+B is executed, the compiler searches for definition of operator +. The object A invokes the operator function and object B is passed as argument. The copy of object B is stored in the formal argument D. The member variables of A are directly available in operator function as the function is invoked by the same object. The addition of individual members are carried out and stored in member variable of object T. The return type of operator function is same as that of its class. The function returns object T and it is assigned to variable C.

5.3.3 Overloading Unary Operators

Overloading devoid of explicit argument to an operator function is called as unary operator overloading. The operator ++, —, and - are unary operators. The unary operators ++ and — can be used as prefix or suffix with the functions. These operators have only single operand. The examples given below illustrate the overloading of unary operators.

Program 5.6 :

Write a program to increment member variables of object using unary ++ Operator overloading.

```
#include<iostream.h>
#include<conio.h>
class num
{
   private :
```

```
int a,b,c,d;
    public :
num (int j, int k, int m , int l)
{
  a=j;
  b=k;
  c=m;
  d=l;
}
  void show (void);
  void operator++();
};
 void num :: show()
 {
          cout<<"A= "<<a<<"B = "<<b<<"C= "<<c<<"D ="<<d;
  }
 void num :: operator ++()
 {
++a;  ++b;  ++c;  ++d;
  }
 main ()
 {
num X(3,2,5,7);
cout<<"\n Before Increment of X :";
x.show ();
++X;
cout<<"\n After Increment of X :";
X.show ();
return 0;
}
```

OUTPUT

Before Increment of X : A=3 B=2 C=5 D=7

After Increment of X : A=4 B=3 C=6 D=8

Explanation: In the above example, the class num contains four integer variables a, b, c and d. The class also has two-member functions show () and operator ++() and one parameterized constructor. The constructor is used to initialize object. The show () displays the contents of the member variables. The operator ++() overloads the unary operator ++. When this operator is

used with Integer or float variables, its value is increased by one. In this function, ++ operator precedes each member variable of class. This operation increments the value of each member variable by one. In function main (), the statement ++X calls the function operator ++ (), where, X is an object of the class num. The function can also be called using statement X. operator ++ (). In the output, values of member variables before and after increment operations are displayed.

5.3.4 Operator Return Type

In the last few examples we declared the operator () of void types i.e., it will not return any value. However, it is possible to return value and assign to it other objects of the same type. The return value of operator is always of class type, because the operator overloading is only for objects. An operator cannot be overloaded for basic data types. Hence, if the operator returns any value, it will always be of class type. Consider the following program.

Program 5.7 :

Write a program to return values from operator () function.

```
# include <iostream.h>
# include <conio.h>
class plusplus
 {
   private:
int num;
   public:
plusplus()
{
  num=0;
}
 int getnum ()
  {
return num;
    }
 plusplus operator ++ (int)
  {
plusplus tmp ;
num=num+1;
tmp.num;=num
return tmp;
    }
};
 void main ( )
  {
```

```
plusplus p1, p2 ;
cout <<"\n p1 = "<<p1. getnum ();
cout <<"\n p2 = "<<p2.getnum() ;
p1=p2++;
cout <<"\n p1 = "<<p1. getnum ();
cout <<"\n p2 = "<<p2.getnum() ;
p1++;
        cout <<"\n p1 = "<<p1. getnum ();
cout <<"\n p2 = "<<p2.getnum() ;
  }
```

OUTPUT

P1=0

p2 =0

p1= 1

p2=1

p1=2

p2=1

Explanation: In the above program, class plusplus is declared with one private integer num. The class constructor initializes the object with zero. The member function getnum () returns current value of variable num. The operator ++ () is overloaded and it can handle as postfix increment of the objects. In case of prefix increment, it will flag an error.

The p1 and p2 are objects of the class plusplus. The statement pl=p2++, first increments the value of p2 and then assigns it to the object p1. The values displayed will be one for the objects. The object p1 is increased. This time the values of object displayed will be two and one.

5.3.5 Cnstraint On ++ And -- Operators

When an operator (increment/decrement) is used as prefix with object, its value is incremented/ decremented before operation and on the other hand the postfix use of operator increments/ decrement the value of variable after its use.

When ++ and — operators are overloaded, no difference exists between postfix and prefix overloaded operator functions. The system has no way of determining whether the operators are overloaded for postfix or prefix operation. Hence, the operator must be overloaded in such a way that it will work for both prefix and postfix operations. The ++ or — operator overloaded for prefix operation works for both prefix as well as postfix operations but with a warning message, but not vice-versa. To make a distinction between prefix and postfix notation of operator, a new syntax is used to indicate postfix operator overloading function. Syntax of the above is as follows:

```
Operator ++( int)        // postfix notation
Operator ++()            // prefix notation
```

The argument followed by operator (++ or —) should have type 'int '. When a postfix operator ++ or operator — is declared, the last parameter must be declared with the type int. No other

type such as float, long etc., is allowed. We can use this operator with all types of variables including float, long etc. Declaring int does not mean that it is only for integer type. The following program illustrates overloading of ++ operator in postfix and prefix style.

Program 5.8 :

Write a program to overload ++ and – operator for prefix and postfix use.

```
#include<conio.h>
#include<iostream.h>
class number
{
float x;
  public :
   number (float k)
   {
    x=k;
   }
   void operator ++(int)                    //postfix notation
   {
    x++;
   }
  void operator —(int)                      //prefix notation
   {
    —x;
   }
  void show()
   {
    cout<<“\n x =”<<x;
   }
void main()
 {
number N(2.3);
cout<<“\n Before Increment:”;
N.show();
cout<<“\n After Increment:”;
N++;
N.show()
cout<<“\n After Decrement:”;
—N;
```

```
N.show();
 }
```

OUTPUT

Before Increment:

X=2.3

After Increment:

X=3.3

After Decrement:

X=2.3

Explanation: In this program, operator ++ and — are overloaded. The ++ operator is overloaded for postfix use and — operator is overloaded for prefix use. It can be seen the keyword (int) is followed by the operator ++, which is necessary for postfix notation of operator. The operator — is overloaded for prefix operation. Here, a value of float member variable is incremented and decremented.

5.3.6 Overloading Binary Operators

Overloading with a single parameter is called as binary operator overloading. Like unary operators, binary operator can also be overloaded. Binary operators require two operands. Binary operators are overloaded by using member functions and friend functions.

(1) Overloading Binary Operators Using Member Functions

If overloaded as a member function they require one argument. The argument contains value of the object, which is to the right of the operator. If we want to perform the addition of two objects o1 and o2, the overloading function should be declared as follows:

operator (num o2) ;

Where, num is a class name and o2 is an object.

To call function operator the statement is as follows: o3=o1+o2;

We know that using class of that object can call a member function. Hence, the object always precedes the called member function. Here, in the above statement, the object o1 invokes the function operator () and the object o2 is used as an argument for the function. The above statement can also be written as follows:

```
o3=o1.operator +(o2);
```

Here, the data members of o1 are passed directly and data members of o2 are passed as an argument. While overloading binary operators, the left-hand operand calls the operator function and right hand operand is used as an argument.

(2) Overloading Binary Operators Using Friend Functions

Alternatively, a friend function can also be used with member functions for overloading of binary operators. The friend function requires two operands to be passed as arguments.

o3=o1+o2;

o3=operator + (o1,o2);

Both the above statements have the same meaning. In the second statement, two objects are passed to the operator function.

The use of member function and friend function produces the same result. Friend functions are useful when we require performing an operation with operand of two different types. Consider the statements:

X =Y+3;

X =3+Y;

Where, X and Y are objects of the same type. The first statement is valid. However, the second statement will not work. The first operand must be an object of the same class. This problem can be overcome by using friend function. The friend function can be called without using object. The friend function can be used with standard data type as left-hand operand and with an object as right-hand operand.

Following programs are illustrated based on the above discussion.

Program 5.9 :

Write a program to overload + binary operator.

```
#include<conio.h>
#include<iostream.h>
class num
{
  private :
int a,b,c,d;
 public :
        void input(void );
void show(void);
num operator +(num);
 };
void num :: input ()
{
cout<<"\n Enter Values for a,b,c and d = ";
cin>>a>>b>>c>>d;
 }
void num :: show ()
{
cout<<" A= "<<a <<" B= " <<b <<" C = "<<c <<" D = "<<d <<"\n";
}
num num ::operator + (num t)
{
num temp;
tmp.a=a+t.a;
```

```
tmp.b=b+t.b;
tmp.c=c+t.c;
tmp.d=d+t.d;
return (tmp);
}
main()
{
num X,Y,Z;
cout<<"\n Object X";
X.input();
cout <<"\n Object Y";
Y.input() ;
Z=X+Y;
cout <<"\nX : ";
X. show () ;
cout<<"Y = ";
Y .show () ;
cout «"Z : ";
z.show();
return 0;}
```

OUTPUT

```
Object X
Enetr Values for a,b,c and d: 1 4 2 1
Object Y
Enetr Values for a,b,c and d: 2 5 4 2
X : A=1 B=4 C=2 D=1
Y : A=2 B=5 C=4 D=2
Z : A=3 B=9 C=6 D=3
```

Explanation: In the above program, binary operator + is overloaded. Using the overloading operator +, addition of member variables of two objects is performed and results are assigned to member variables of third object. In this program x, Y and Z are objects of class num. The statement Z=X+Y invokes the operator function. In this statement the object Y is assigned to object t of operator function and member variables of X are accessed directly. The object tmp is used for holding the result of addition and it is returned to object Z after function execution. The function show () displays the values of three objects.

5.3.7 Overloading With friend Function

Friend functions are more useful in operator overloading. They offer better flexibility which is not provided by the member function of the class. The difference between member function and friend function is that the member function takes arguments explicitly. Quite the opposite, friend

function needs parameters to be explicitly passed. The syntax of operator overloading with friend function is as follows:

```
friend return-type operator operator-symbol (variable, variable2)
 {
statement1;
statement2;
}
```

The keyword friend precedes function prototype declaration. Prototype must be declared inside the class. However, the function can be defined inside or outside the class. The arguments used in friend function are generally objects of the friend classes. A friend function is similar to normal function. The only difference is that friend function can access private members of the class through the objects. Friend function has no permission to access private members of a class directly. However, it can access the private members via objects of the same class.

Program 5.10 :

Write a program to overload unary operator using friend function.

```
# include <iostream.h>
# include <constream.h>
class complex
{
float real, img;
  public:
    complex ( )  //zero argument constructor
    {
          real=imag=o;
      }
   complex (float r, float i)        // two argument constructor
    {
       real=r;
      imag=i;
      }
friend complex operator - ( complex c)
{
c.real=-c.real;
c.imag=-c.imag;
return c;
}
void display ()
```

```
{
cout<<"\n Real :"<<real;
cout<<"\n Imag :"<<imag;
}
void main ( )
(
complex c1(1.5,2.5)
        complex.display ();
c2=-c1;
cout <<"\n\n After negation \n ";
c2 . display
}
```

OUTPUT

Real : 1.5

Imag : 2.5

After Negation Real: -1.5

Imag : -2.5

Explanation: In the above program, operator — is overloaded using friend function. The operator function is defined as friend. The statement c2 = - c1 invokes the operator function. This statement also returns the negated values of c1 without affecting actual value of c1 and assigns it to object C2.

The negation operation can also be used with an object to alter its own data member variables. In such a case the object itself acts as a source and destination object.

5.4 Type Conversion

When constants and variables of various data types are clubbed in a single expression, automatic type conversion takes place. This is so for basic data types. The compiler has no idea about the user-defined data types and about their conversion to other data types. The programmer should write the routines that convert basic data types to user-defined data types or vice versa. There are three possibilities of data conversion as given below:

(1) Conversion from basic data type to user-defined data type (class type).

(2) Conversion from class type to basic data type.

(3) Conversion from one class type to another class type.

(1) Conversion from Basic to Class Type

The conversion from basic to class type is easily carried out. The compiler automatically does it with the help of in-built routines or by applying typecasting. In this type the left-hand operand of = sign is always class type and right-hand operand is always basic type. The program given below explains the conversion from basic to class type.

Program 5.10 :

Write a program to define constructor with no argument and with float argument.

Explain how compiler invokes constructor depending on data type.

```
#include<conio.h>
#include<iostream.h>
class data
{
int x;
float f;
public :
  data()
  {
x=0;  f=0;
   }
 data (float m)
    {
x=2;  f=m;
    }
 void show()
    {
cout<<"\n x= "<<x<<"f = "<<f;
cout<<"\n x= "<<x<<"f = "<<f;
     }
};
int main()
{
data z;
z=1;
z.show();
z=2.5;
z.show();
return 0;
}
```

OUTPUT

X=2 f=1

X=2 f=1

X=2 f=2.5

X=2 f=2.5

Explanation: In the above program, the class data has two member variables each of integer

and float types respectively. It also has two constructors one with no argument and second with float argument. The member function show () displays the contents of the data members. In function main () , z is an object of class data. When z is created the constructor with no argument is called and data members are initialized to zero. When z is initialized to one the constructor with float argument is invoked. The integer value is converted to float type and assigned member variable f. Again when z is assigned to 2.5, same process is repeated. Thus, the conversion from basic to class type is carried out.

(2) Conversion from Class Type to Basic Data Type

In the previous example, we studied how compiler makes conversion from basic to class type. The compiler does not have any knowledge about the user-defined data type built using classes. In this type conversion, the programmer explicitly needs to tell the compiler how to perform conversion from class to basic type. These instructions are written in a member function. Such type of conversion is also known as overloading of type cast operators. The compiler first searches for the operator keyword flowed by data type and if it is not defined, it applies the conversion functions. In this type, the left-hander and is always of basic data type and right-hand operand is always of class type. While carrying this conversion, the statement should satisfy the following conditions:

- The conversion function should not have any argument.
- Do not mention return type.
- It should be a class member function.

Program 5.11 :

Write a program to convert class type data to basic type data.

```
#include<conio.h>
#include<iostream.h>
class data
{
int x;
float f;
  public :
data()
{
x=0;
f=0;
}
operator int()
{
return (x);
        }

operator float ()
```

```
{
return f;
}
data (float m)
{
x=2;
f=m;
}
void show()
{
cout<<"\n x= "<<x<<"f = "<<f;
cout<<"\n x= "<<x<<"f = "<<f;

}
};
int main()
{
int j;
float f;
data a;
a=5.5;

j=a;
f=a;
cout<<"\n Value of J :"<<j;
cout<<"\n Value of f :"<<f;
return 0;
}
```

OUTPUT

Value of j: 2

Value of f: 5.5

Explanation: In the above program, the class data has two member variables each of integer and float data type. It also contains constructors as per described in the last example. In addition, it contains overloaded data types int and float. These functions are useful for conversion of data from class type to basic type. Consider the following statements:

```
j=a;
f=a;
```

In the first statement object a is assigned to integer variable j. We know that class type data is a combination of one or more basic data types. The class contains two-member functions operator int () and operator float (). Both these functions are able to convert data types from class to basic. In first statement, the variable j is of integer type, the function operator int () is invoked and integer value data member is returned. In second statement, f is of float type; the member function operator float () is invoked.

(3) Conversion from One Class Type to Another Class Type

When an object of one class is assigned to object of another class, it is necessary to give clear-cut instruction to the compiler. How to make conversion between these two user-defined data types? The method must be instructed to the compiler. There are two ways to convert object data type from one class to another. One is to define a conversion operator function in source class or a one-argument constructor in a destination class. Consider the following example:

X=A;

Here, X is an object of class XYZ and A is an object of class ABC. The class ABC data type is converted to class XYZ. The conversion happens from class ABC to XYZ. The ABC is a source class and XYZ is a destination class.

We know the operator function operator data-type (). Here, data type may be built-in data or user-defined data type. In the above declaration, the data type indicates target type of object. Here, conversion takes place from class ABC (source class) to class XYZ (destination class).

Program 5.12 :

Write a program to convert integer to date and vice versa using conversion function in source class.

```
#include<conio.h>
#include<iostream.h>
#include<string.h>
#include<stdlib.h>
class date
{
 char d[10];
  public :
date()
{
 d[10]=NULL;
}
date(char *e)
{
strcpy(d,e);
}
void show()
{
```

```
cout<<d;
}
};
class dmy
{
int mt, dy, yr;
 public :
dmy()
{
mt=dy=yr=0;
}
dmy(int m, int d, int y)
{
mt=m;
dy=d;
yr=y;
}
operator date()
{
char tmp[3], dt[9];

iota(dy,dt,10);
strcat(dt,"-");
iota(mt,tmp,10);
strcat(dt,tmp);
return (date(dt));
}
void show()
{
cout<<dy<<" "<<mt<<" "<<yr;
}
};

int main()
{
date D1;
```

```
dmy D2(1,7,99);
D1=D2;
cout<<endl<<"D1";
D1.show();
cout<<endl<<"D2";
D2.show();
return 0;
}
```

OUTPUT

D1=7-1-99

D2=7 1 99

Explanation: In the above program, date and dmy are two classes declared. In function main (), D1 is an object of class date and D2 is an object of class dmy. The object D2 is initialized.

The statement D1=D2 initializes D1 with D2. Here, both the objects D1 and D2 are of different types hence the conversion function date () is called to perform the conversion from one object to another object.

Self Learning Exercises

1. Overload function in C++
   - A. a group function with the same name
   - B. all have the same number and type of arguments
   - C. functions with same name and same number and type of arguments
   - D. All of the above
2. The fields in a class of a c++ program are by default
   - A. protected
   - B. public
   - C. private
   - D. None of These
3. What will be the output of the following code.

```
class base
 {
public:
    void baseFun(){ cout«"from base"«endl;}
 };
class deri:public base
 {
public:
```

```
        void baseFun(){ cout« "from derived"«endl;}
    };
void SomeFunc(base *baseObj)
{
    baseObj->baseFun();
}
int main()
{
base baseObject;
SomeFunc(&baseObject);
deri deriObject;
SomeFunc(&deriObject);
}
```

4. What will be the output of the following code

```
 class some{
public:
        ~some()
        {
                cout«"some's destructor"«endl;
        }
};
void main()
{
        some s;
        s.~some();
}
```

5.5 Rules For Overloading Operators

This section describes the rules to be followed for overloading operators.

- Overloading of an operator cannot change the basic idea of an operator. When an operator is overloaded, its properties like syntax, precedence, and associativity remain constant. For example A and B are objects. The statement

  A+=B;

  assigns addition of objects A and B to A. The overloaded operator must carry the same task like original operator according to the language. The following statement must perform the same operation like the last statement.

  A=A+B;

- Overloading of an operator must never change its natural meaning. An overloaded op-

erator + can be used for subtraction of two objects, but this type of code decreases the utility of the program. Remember that the aim of operator overloading is to comfort the programmer to carry various operations with objects.

- The overloaded operator should contain one operand of user-defined data type. Overloading operators are only for classes. We cannot overload the operator for built-in data types.
- Overloaded operators have the same syntax as the original operator. They cannot be prevailing over the original operators.
- There is no upper limit to the number of overloading for any operator. An operator can be overloaded for a number of times if the arguments are different in each overloaded operator function.
- Operator overloading is applicable within the scope (extent) in which overloading occurs. Only existing operators can be overloaded. We cannot create a new operator.
- C++ has wide range of operators. However, few operators cannot be overloaded to operate in the same manner like built-in operators. The operators given in Table cannot be overloaded

NON-overloaded operators

Operator	Description
.	Member operator
.*	Pointer to member operator
::.	Scope access operator
?:	Conditional operator
Sizeof ()	Size of operator
# and ##	Preprocessor symbols

NON-overloaded operators with friend function

Operator	Description
()	Function call delimiter/operator
=	Assignment operator
[]	Subscribing operator
->	Class member access operator

- When unary operator are overloaded using member function, it requires no explicit friend argument and returns no value where as when it is overloaded using friend function, it require one reference argument.
- When binary operators are overloaded using friend function, it requires two arguments where as when overloaded using member function. it require one argument

5.6 Summary

C++ makes it possible for the programmer to use the same function name for various times for different intentions. This is called as function overloading or function polymorphism.

Operator overloading is one of the most helpful concepts introduced by the C++ language, Operator overloading provides the capability to redefine the language in which working of operator can be changed.

Overloaded operators are redefined within a C++ class using the keyword operator fol. lowed by an operator symbol. When an operator is overloaded, the produced symbol is called the operator function name.

Overloading of operator cannot change the basic idea of an operator. When an operator is overloaded, its properties like syntax, precedence, and associativity remain constant.

The keyword operator defines a new action or operation to the operator.

The operators ++, —, and - are unary operators. The unary operators ++ and — can be used as prefix or suffix with the functions. These operators have only single operand.

Binary operators require two operands. Binary operators are overloaded by using member functions and friend functions.

The conversion routine may be single argument constructor or an operator function. It depends on the object whether it is source or destination object.

Defining multiple conversion routines puts the complier in an uncertain condition. The com. compiler fails to select appropriate conversion routines.

There are three possibilities of data conversion. They are given below:

(i) Conversion from basic data type to user-defined data type (class type)- The conversion from basic to class type is easily carried out. The compiler automatically does it with the help of in-built routines or by applying type casting.

(ii) Conversion from class type to basic data type- The compiler does not have any knowledge about the user-defined data type built using classes. In this type of conversion the programmer, needs explicitly to tell the compiler how to perform conversion from class to basic type. These instructions are written in a member function such type of conversion is also known as overloading of type cast operator.

(iii) Conversion from one class type to another class type- when an object of one class is assigned to an object of another class it is necessary to give clear cut instruction to the compiler about how to make conversion between these two user –defined data types. Using compiler about how to make conversion this function can be performed

5.7 Glossary

- . An object is a variable of a class type, as distinct from a variable of a native type. The behavior of an object is defined by the code that implements the class to which the object belongs. For example, a variable of type string is an object whose behavior is controlled by the definition of the string class
- The keyword operator is used to indicate that the following symbol is the name of a C++ operator that we are redefining, either globally or for a particular class. For example, to redefine =, we have to specify operator = as the name of the function we are writing, rather than just =, so that the compiler does not object to seeing an operator when it expects an identifier
- Polymorphism:The ability to use the same identifier to name two or more functions within the same scope of a program; gives the appearance that a single function exhibits different behavior depending on the nature (i.e., classes/types and number) of its parameters.

5.8 Further Readings

1. Introduction To Object Oriented Programming And C++ ,Kanetkar , BPB,**ISBN:** 8176568635
2. A complete Reference to C/C++,Herbert Shildt
3. C++: An introduction to programming by Jense Liberty Tim Keogh: BPB Publications, New Delhi
4. OO Programming in C++ by Robert Lafore: , Galgotia Publications Pvt. Ltd., Daryaganj,New Delhi
5. Object Oriented Programming Using C++, Sanjeev Sofat, Cyber Tech. Publication, New Delhi
6. Object Oriented Programming in C++ by E. Balaguruswamy, TMH Publishing Co. Ltd.,New Delhi

5.9 Answers to Self Learning Exercises

1. A
2. C
3. from base

   from base
4. some's destructor

   some's destructor

5.10 Unit End Questions

Answer the following questions.

(1) What is function overloading?

(2) What are the rules for defining overloaded functions?

(3) What precautions should we take while overloading function?

(4) What do you mean by operator overloading?

(5) What is the use of the keyword operator?

(6) What are the rules for overloading operators?

(7) What is the difference between operator overloading and function overloading?

(8) What is the difference between overloading of binary operators and unary operators?

(9) How are friend functions used to carry out overloading of operators? In which situation are they helpful?

(10) Explain conversion of data from basic to class type. Explain the role of compiler.

(11) Explain conversion from class to basic type.

(12) Explain conversion from class type to class type.

(13) What are source and destination objects?

(14) List the keywords that cannot be overloaded.

UNIT - VII

Pointers, Arrays and Memory

Structure of the Unit

7.0 Objective

After Reading this Unit you will be able to

- Understand the concept of pointers
- write programs using pointers and arrays
- understand the memory models

7.1 Introduction

Most of the learners feel that pointer is not a easy topic. But, pointers can make programs quicker, straightforward and memory efficient. C++ gives more importance to pointers. So, it is important to know the operation and applications of pointers

Like C, in C++ variables are used to hold data values during program execution. Every variable when declared occupies certain memory locations. It is possible to access and display the address of memory location of variables using '&' operator. Memory is arranged in series of bytes. These series of bytes are numbered from zero onwards. The number specified to a cell is known as memory address.

Variable stores the address of any type of variable in the memory. The data type of variable and pointer should be of the same type. The pointer is represented by (*) asterisk symbol.

A byte is combination of 8 bits. The binary numbers 0 and 1 are known as bits. Each byte in the memory is specified with a unique address in the memory, The number of memory locations pointed by a pointer depends on the data type of pointer.

When the memory is allocated at run-time is called as dynamic memory allocation to run time allocation. Such type of memory allocation is essential for data structures and can efficiently handle them using pointers. Another reason to use pointers is in array. Arrays are used to store more values. Actually, the name of array is a pointer.

7.2 Pointer

A pointer is a variable, which holds the address of another variable in the memory. The pointer can be declared by any name like other variables but its name should be started by '*' .

Advantages of Pointers

(1) Pointers save the memory space.

(2) Execution time with pointer is faster.

(3) The memory is accessed efficiently with the pointers.

(4) Pointers are used with data structures. They are useful for representing two-dimensional and multi-dimensional arrays.

(5) In C++ a pointer declared to a base class could access the objects of derived class. But its reverse is not possible. Means the derived class pointer cannot point to the base class object.

7.2.1 Pointer Declaration

The pointer name is started with (*) we can declare pointer as below.

```
int *ptr;
float *fptr;
char *cptr;
```

1. The statement int *ptr informs to the compiler that it holds the address of any integer variable. Similarly the statement float *fptr will hold the address of any float variable and' y' will hold the address of any character variable.
2. We can access the value of any variable by the name of variable. In case of pointer we can access the value of pointed variable using the (*) with name of pointer. If we use the name of pointer without (*) it will display the address of pointed variable.
3. The '&' is a address operator and it represents the address of the variable. The address of any variable is a whole number. The operator '&' immediately preceding the variable returns the address of the variable.

PROGRAM: Write a program to display the address of the variable.

```
# include <iostream.h>
# include <conio.h>
main()
{
        int n ;
clrscr() ;
cout<<"Enter a Number = ";
cin<< n;
cout<<"Value of n = " <<n;
cout<<"Address of n = " <<(unsigned)&n ;
getch() ;
}
```

OUTPUT

```
Enter a Number = 10
```

Explanation: The memory location of a variable is system dependent. Hence, address of a variable cannot be predicted immediately. The address of variable depends on various things for instance memory model, addressing scheme and present system settings

PROGRAM: Write a program to declare a pointer. Display the value and address of the variable using pointers.

```
# include <studio.h>
# include <iostream.h>
# include <conio.h>
void main()
{
        int *p ;
        int x = 10;
        p = &x ;
        clrscr() ;
        printf("\n x = %u \t (using printf() ",x,p) ;
```

```
        cout<<"\n x = "<<x <<" &x = " <<&x<<" (Using cout())";
        printf("\n x=%d &x=%u \t (Using pointer)",*p,p) ;
        cout<<"\n *p="<<*p <<"\t &p ="<<p<<"\t (Contents of pointer)" ;
}
```

OUTPUT

x = 10 &x = 65524 (Using printf())

x = 10 &x = 0x8f94fff4 (Using cout())

x = 10 &x = 65524 (Using pointer)

*p = 10 &p = 0x8f94fff4 (Contents of pointer)

Explanation: In the above program, *p is an integer pointer and x is an integer variable. The address of variable x is assigned to pointer p. Ampersand (&) operator preceded with variable displays the memory location of that variable. The printf() and cout() statements display address in different formats. The printf() statement displays the address as an unsigned integer whereas the cout() statement displays the address in hexadecimal format. We can access the contents of variable x using pointer p. Output of the program is shown above.

PROGRAM: Write a program to display memory address of a variable. Typecast the memory address from hexadecimal to unsigned integer.

```
# include <iostream.h>
# include <conio.h>
void main()
{
        int a ;
        float b ;
        char c ;
        clrscr();
        cout<<"\n Enter interger number = ";
        cin<<a;
        cout<<"\n Enter float number = " ;
        cin<<b ;
        cout<<"\n Enter character = ";
        cin<<c;
        cout<<"\n The entered int is = "<<a;
        cout<<"\n The entered float is = "<<b ;
        cout<<"\n The entered char is = "<<c;
        cout<<"\n The entered number is stored at location = "<<(unsigned)&sa;
        cout<<"\n The entered float is stored at location = "<<(unsigned)&sb ;
        cout<<"\n The entered char is stored at location = "<<(unsigned)&c;
        getch() ;
}
```

OUTPUT

Enter integer number = 34

Enter float number = 343.34

Enter character = g

The entered int is = 34

The entered float is = 343.339996

The entered char is = g

The entered number is stored at location =4096

The entered float is stored at location = 4092

The entered char is stored at location = 4091

Explanation: In the above program, variables a, b and c of int, float and char type are declared. The entered values with their addresses are displayed. The addresses of the variables are displayed by preceding ampersand (&) operator with variable name. The typecasting syntax (unsigned) is done to typecast hexadecimal address into unsigned integer.

7.2.2 Void Pointer

We can also be declare a pointer variable as void type. Void pointers cannot be dereferenced without explicit type conversion. This is because being void the compiler cannot determine the size of the object that the pointer points to.

We can declare void pointer, but we cannot declare void variable . so, the declaration void ptr will display an error message "sizeof 'ptr' is unknown or zero" after compilation.

A void pointer can point to any type of variable with proper typecasting. The size of void pointer displayed will be two. When pointer is declared as void, two bytes are allocated to it.

Void variables cannot be declared because memory is not allocated to them and there is no place to store the address. Therefore, void variables cannot actually do the work for which they are made.

PROGRAM: Write a program to declare a void pointer. Assign address of int, float, and char variables to the void pointer using typecasting method. Display the contents of various variables.

```
// void pointers //
# include <stdio.h>
# include <iostream.h>
# include <conio.h>
int p ;
float d ;
char c;
                    void *pt = &p ;          // pt points to p
void main(void)
{
        clrscr() ;
(int *) pt = 12 ;
```

```
cout <<"\n p =" <<p ;
pt = &d ;
(float *)pt = 5.4 ;
cout <<"\n r = "<<d;
pt = &c ;
(char*) pt = 'S' ;
cout <<"\n C="<<c;
}
```

OUTPUT

p = 12

r = 5.4

c = S

Explanation: In the above example, variables p, d, and c are variables of type int, float, and char respectively. Pointer pt is a pointer of type void. These entire variables are declared before main() . The pointer is initialized with the address of integer variable p i.e., the pointer p points to variable x.

The statement * (int *) pt = 12 assigns the integer value 12 to pointer pt i.e., to a variable p. The contents of variable p are displayed using the succeeding statements. The declaration * (int *) tells the compiler the value assigned is of integer type. Thus, assignment of float and char type is carried out. The statements * (int *) pt = 12, * (float *) pt = 5.4 and * (char*) pt= 151 help the compiler to exactly determine the size of data types.

7.2.3 Pointer to Class

We know that pointer is a variable that keeps the address of another data variable. The variable can be of any data type i.e., int, float or double. In the same way, we can also define pointer to class. by which starting address of the member variables can be accessed. Such pointers are called class pointers.

```
class emp
{
char name[20];
char dept[25];
int salary;
};
class emp *obj1;
```

In the above example, *obj1 is pointer to class emp.

The syntax for using pointer with member is given below.

1) obj1->name
2) obj1->dept
3) obj1->salary

By executing these three statements, we can access the value of member of the calss emp.

PROGRAM: Write a program to declare a class. Declare pointer to class. Initialize and display the contents of the class member.

```
# include <iostream.h>
# include <conio.h>
void main()
{
        class man
        {
                public:
                char name[10] ;
                int age ;
        };
man m={"RAVINDRA",15} ;
man *ptr ;
ptr=&(man)m ;
// *ptr=(man)m ;
// *ptr=man(m) ;
// ptr=&m ;
clrscr() ;
cout <<"\n"<<m.name <<" "<<m.age ;
cout <<"\n" <<ptr->name<<" "<<ptr->age ;
}
```

OUTPUT

RAVINDRA 15

RAVINDRA 15

Explanation: In the above program, the pointer ptr points to the object m. The statement ptr=& (man) m; assigns address of first member element of the class to the pointer ptr. Using the dot operator and arrow operator, contents can be displayed. The display of class contents is possible because the class variables are declared as public. The statements given below can also be used to assign address of objects to the pointer.

- ptr=& (man)m;
- *ptr=(man)m;
- *ptr=man(m);
- *ptr=m;
- ptr=&m;

7.2.4 Pointer to Object

We can also have a pointer to object. The pointer which points to a object is known as pointer to object. The following program explains it.

PROGRAM: Write a program to declare an object and pointer to the class. Invoke member functions using pointer.

```
# include <iostream.h>
# include <conio.h>
class Bill
{
        int qty ;
        float price ;
        float amount ;
        public :
        void getdata (int a , float b, float c)
        {
                qty = a;
                price = b;
                amount = c;
        }

        void show()
        {
        cout <<"Quantity : "<<qty <<"\n";
        cout <<" Price  : "<<price <<"\n";
        cout <<"Amount : "<<amount <<"\n";
        }
} ;
int main()
{
                clrscr() ;
                Bill s;
                Bill *ptr = &s;
                Ptr->getdata(45,10,25,45,*10.25;) ;
                (*ptr).show() ;
                return 0;
}
```

OUTPUT

Quantity : 45

Price : 10.25

Amount : 461.25

Explanation: In the above program, the class Bill contains two float and one integer member. The class Bill also contains member functions get data() and show() to read and display the data. In function main(), s is an object of class Bill and ptr is pointer to the same class. The address of object s is assigned to pointer ptr. Using pointer ptr with arrow operator(->) and dot operator(.), member functions are invoked. The statements used for invoking functions are given next.

PROGRAM: Write a program to create dynamically an array of objects of class type. Use new operator.

```
# include <iostream.h>
# include <conio.h>
class Bill
{
        int qty ;
        float price ;
    float amount ;
    public:
    void getdata(int a, float b, float c)
        {
            qty = a;
            price = d ;
            amount = c;
        }
        void show()
        {
            cout <<"Quantity : "<<qty <<"\n" ;
            cout <<"Price : "<<price <<"\n";
            cout <<"Amount : " <<amount <<"\n";
        }
};
int main ( )
{
        clrscr() ;
        Bill *s = new Bill[2];
        Bill *d = s;
        Int x,i ;
```

```
        float y;
        for (i = 0; i<2; i++ )
        {
        cout <<"\n Enter Quantity and Price : ";
        cin >>x >> y;
        s->getdata ( x,y,x*y);
        s++ ;
        }
        for (i = 0; i<2 ; i++)
        {
        cout <<endl;
        d->show() ;
        d++ ;
        }

        return 0;
}
```

OUTPUT

```
Enter Quantity and Price : 5 5.3
Enter Quantity and Price : 8 9.5
Quantity : 5
Price    : 5.3
Amount   : 26.5
Quantity : 8
Price    : 9.5
Amount   : 76
```

Explanation: In the above program, the class Bill is same as in the previous example. In main(), using new memory allocation operator, memory required for two objects is allocated to pointer s i.e., 10 bytes. The first for loop accepts the data through the keyboard. Immediately after this, the data is sent to the member function getdata() . The pointer s is incremented. After incrimination, it points to the next memory location of its type. Thus, two records are read through the keyboard. The second for loop is used to display the contents on the screen. Here, the function show() is invoked. The logic used is same as in the first loop. The functions are invoked using pointer and it is explained in the previous example.

7.2.5 this pointer

The this is a keyword . this pointer is also known as magic pointer.

We know that objects are used to invoke the non-static member functions of the class.

The keyword this does not need to be declared. The this is a pointer variable which always Points to the object for which function is calling. When we create a object the c++ compiler creates a this pointer in the memory. And when we call any member function by using object this pointer always point to that object. Thus member function of every object has access to a pointer called this , which points the object with which the member function is associated. The this pointer cab be used to access the data in the object it points to.

PROGRAM: Write a program to use this pointer and return pointer reference.

```
# include <iostream.h>
# include <conio.h>
        classe number
        {
        int num;
        public :

        void input()
        {
        cout <<"\n Enter a Number : ";
        cin >>num;
        }
        void show()
        {
  cout <<"\n The Minimum Number : "<<num;
  }
        number min (number t )
        {if (t.num<num)
                return t ;
                else
                return *this ;     }
        }
        void main()
{
        clrscr() ;
number n, n1, n2;
n1.input() ;
n2.input() ;
n = n1.min(n2) ;
n.show() ;
}
```

OUTPUT

Enter a Number : 152

Enter a Number : 458

The Minimum Number : 152

Explanation: In the above program, the class number contains one integer number variable num. The class also contains member functions input(), show() and min() . The input() function reads an integer through the keyboard. The show() function displays the contents on the screen. The min() function finds minimum number out of two. The variables n, n1, and n-2 are objects of class number.

In function main() , the objects n1 and n2 calls the function input() and read integer. Both the objects n1and n2 are passed to the function min() with the statement n=n1.min (n2) . The object n receives the returned value by the function min() . The object n1 calls the function min (). The object n2 is passed as an argument in function min() .

In function num() , the formal object t receives the contents of argument n2. In the same function, the object n1 is also accessible. We know that the pointer this is present in the body of every non-static member function. The pointer this points to the object n1. Using pointer this we can access the individual member variables of object n1. The if statement compares the two objects and return statement returns the smaller objects.

7.2.6 Pointer To Derived Classes and Base Classes

We can declare a pointer which points to the base class as well as the derived class. Like normal pointer a object pointer can point to different classes. For example, X is a base class and Y is a derived class. The pointer pointing to X can also point to Y class.

PROGRAM : Write a program to declare a pointer to the base class and access the member variable of base and derived class.

```
                                        / / Pointer to base object / /
# include <iostream.h>
# include <conio.h>
        class A
        {
                public:
        int b ;
        void display()
        {
        cout <<"b = "<<b <<"\n " ;
        }
        } ;
class B : public A
{
        public :
        int d ;
```

```
        void display()
        {
        cout <<"b = "<<b <<"\n" <<" d = <<d <<"\n ";
        }
} ;
main()
{
clrscr() ;
A * cp ;
A base ;
cp =&base
cp->b = 100;
// cp->d = 200 ; Not Accessible
cout <<"\n cp points to the base object \n";
cp->display() ;
B d;
cout <<" \n cp points to the derived class \n";
cp = &b ;
cp->b = 150 ;
// cp->d=300; Not accessible
cp->display() ;
return 0;
}
```

OUTPUT

```
cp  points to the base object
b = 100
cp points to the derived class
b = 150
```

Explanation: In the above program, A and B are two classes with one integer member variable and member function. The Class B is derived from class A. The pointer cp points to the class A. The variable base is an object of the class A. The address of object base is assigned to pointer cpo The pointer cp can access the member variable b, a member of base class but cannot access variable d, a member of derived class. Thus, the following statement is invalid.

The variable b is an object of class B. The address of object b is assigned to the pointer cpo How" ever, b is an object of derived class and access to member variable d is not possible. Thus, the following statement is invalid.

(b) cp->d=300;

Here, cp is pointer to the base class and could not access the members of derived class

PROGRAM : Write a program to declare a pointer to the derived class and access the member variable of base and derived class.

```
                                        / / Pointer to derived object / /
# include <iostream.h>
# include <conio.h>
class A
{
        public:
        int b;
        void display()
{
        cout <<"b = "<<d <<"\n";
}
};
class B : public A
{
        public:
        int d;
        void display()
        {
        cout <<"\tb = " <<b <<"\n" <<"\td = "<<d <<"\n";
}
};
main()
{
clrscr() ;
B * cp ;
B b ;
cp = &b;
cp->b = 100;
cp->d = 350;
cout <<" \n cp points to the derived object \n ";
cp->display() ;
return 0;
}
```

OUTPUT

cp points to the derived object

b = 100

d = 350

Explanation: The above program is same as the previous one. The only difference is that the pointer cp points to the objects of derived class. The pointer cp is pointer to class B. The variable b is an object of class B. The address of b is assigned to the pointer cpo The pointer cp can access the member variables of both base and derived classes. The output of the program is shown above.

PROGRAM : Write a program to declare object and pointer to class. Assign value of pointer to object. Display their values. Also carry out conversion from basic type to class type.

```
# include <iostream.h>
# include <conio.h>
# include <alloc.h>
class A
{
        private:
        int a;
        public:

        A()
        {
  a = 30;
        }
        void show()
        {
   cout <<"\n a = "<< a ;
   }
A(int x)
{
this->a = x;
}
};
int main()
{
clrscr() ;
A k,*b,a ;
*b = 50 ;
```

```
k = *b ;
b->show() ;
a.show() ;
k.show() ;
return 0;
}
```

OUTPUT

```
a = 50
a = 30
a = 50
```

Explanation: In the above program, a and k are objects of class A. The *b is a pointer object. The objects a and k are initialized to 30 by the constructor. The statement *b=50 assigns 50 to data member a. This is carried out by the conversion constructor A (int). The value of *b is assigned to object k. The member function show() is called by all the three objects. The output of the program is given above.

7.2.7 Pointer to Members

We can also have address of member variables and store it to a pointer. The following programs explain how to obtain address and accessing member variables with pointers.

PROGRAM : Write a program to initialize and display the contents of the structure using dot (.) and arrow (->) operator.

```
# include <iostream.h>
# include <conio.h>
int main ( )
{
        clrscr ( ) ;

        struct c
{
        char *name;
};
c b. * bp ;
bp ->name = "CPP ";
b. name =" C &";
cout <<b. name ;
cout <<bp - >name ;
return 0;
}
```

OUTPUT

C & CPP

Explanation: In the above program, structure c is declared. The variable b is object and bp is a pointer object to structure c. The elements are initialized and displayed using dot (.) and arrow (- » operators. These are the traditional operators which are commonly used in C. C++ allows two new operators. * and- > * to carry the same task. These C++ operators are recognized as pointer to member operators.

PROGRAM : Write a program to initialize and display the contents of the structure using . *and arrow - >*operator.

```
# include <iostream.h>
# include <conio.h>
struct data
{
        int x ;
        float y ;
};
int main()
{
        clrscr() ;
        int data::*xp = &data::x;
        float data::*yp =&data::y ;

        data d ={11,1.14};
        cout <<endl<< d.*xp << endl <<d.*yp;
        data *pp;
        pp = &d;
        cout << endl<< pp->*xp << endl<< pp->*yp;
        d.*xp = 22;
        pp->*yp = 8.25 ;
        cout <<endl <<d. *xp <<endl<<d. *yp;
        cout <<endl<<pp->&xp <<endl<<p->*yp;
        return 0;
}
```

OUTPUT

11

1.14

11

1.14

22

8.25

22

8.25

Explanation: In the above program, struct data is defined and it has two data members of integer and float type. Consider the following statements:

int data: : *xp=&data: :X;

float data: :*yp=&data: :Y;

The *xp and *yp are pointers. The class name followed by scope access operator points that they are pointers to member variables of structures in t x and float y. The rest part of the statement initializes the pointers with addresses of x and y respectively.

"The rest part of the program uses operators.* and ->* to initialize and access the member elements in the same fashion like the operator dot (.) and arrow (->).

PROGRAM : Write a program to declare and initialize an array . Access and display the elements using .*and ->*operators.

```
# include <iostream.h>
# include <conio.h>
struct data
{
int x ;
float y ;
};
int main()
{
clrscr() ;
int data::*xp =&data::x ;
float data::*yp =&data::y;
data darr[] = { {12,2.5},
                {58,2.4},
                {15,5.7}
               };
```

```
        for (int j = 0 ; j <=2; j++)
        cout <<end<<darr[j].*xp <<"“<<darr[j].*yp ;
        return 0;
}
```

OUTPUT

12 2.5

58 2.4

15 5.7

Explanation: The above program is the same as the last one. An array darr[] is initialized. The for loop and the operator. * accesses the elements and elements are displayed on the console.

PROGRAM : Write a program to declare variables and pointers as members of class and access them using pointer to members.

```
# include <iostream.h>
# include <conio.h>
class data
{
        public:
                int x ;
        float y ;
        int *z;
        float *k;
        int * *1;
        float **m;
};
void main()
{
        clrscr() ;
        int data::*xp =&data::x ;
        float data::*yp = &data::y ;
        int * data::*zp = &data::z;
        float *data::*kp=&data::k;
        int **data::*lp =&data::l;
        float **data::*mp=&data::m;
        data ob = {51, 4.58,&ob.x, &ob.z, &ob.k};
        data *pp;
        pp = &ob ;
        cout<<endl<<ob.&xp<<endl<<ob.&yp;
```

```
        cout<<endl<<*(ob.*zp)<<endl<<*(ob.*kp) ;
        cout<<endl<<*(ob.*lp)<<endl<<**(ob.*mp);
        cout<<endl<<pp->*xp<<endl<<pp->*yp;
        cout<<endl<<*pp->*zp)<<endl<<*(pp ->*kp);
        cout<<endl<<**(pp->*lp)<<endl<<**(pp->*mp);
        *(ob.*zp) = 11;
        **(pp->*mp) = 8.24;
        cout<<endl <<ob.*xp<<endl<<ob.*yp;
}
```

OUTPUT

51
4.58
51
4.58
51
4.58
51
4.58
51
4.58
51
4.58
11
8.24

Explanation : in the above program, the class and data are declares and all members are public. The ob is an object of the class data and it is initialized. The variable pp is a pointer and initialized with address of objec ob. The rest program statements use the pointer to member operators and display the contents of the class on the screen.

PROGRAM : Write a program that invokes member functions using pointer to functions.

```
# include <iostream.h>
# include <conio.h>
class data
{
        public:
        void joy1()
 {
cout <<endl<<(unsigned)this<<"in joy1";
}
```

```
void joy2()
{
cout<<endl<<(unsigned)this<<"in joy2";
}
void joy3()
{
cout <<endl<<(unsigned)this<<"in joy3"<<endl;
}
} ;

void main()
{
        clrscr() ;
        data od[4];
        void(data::*m[3])()={&data::joy1,&data::joy2,&data::joy3};
for (int x = 0; x<= 3 ; x++ )
{
        for (int y = 0; y <= 2 ; y++ ) (od[x].*m[y])() ;
}
}
```

OUTPUT

```
65522 in joy 1
65522 in joy 2
65522 in joy 3
65523 in joy 1
65523 in joy 2
65523 in joy 3
65524 in joy 1
65524 in joy 2
65524 in joy 3
65525 in joy 1
65525 in joy 2
65525 in joy 3
```

Explanation: In the above program, class data has three-member variables joy1, joy2, and joy3. The array m[3] holds the addresses of member functions. The nested f or loops and statement within it invokes member functions.

PROGRAM : Write a program to declare pointer to member variable and display the contents of the variable.

```
# include <iostream.h>
# include <conio.h>
class A
{
public :
int x ;
int y ;
int z ;
};
void main()
{
clrscr() ;
A a;
int * p;
a.x =5;
a.y=15;
p = &a.x;
cout <<endl<<"x = "<<*p ;
p++ ;
cout <<endl <<"y = "<<*p;
p++ ;
cout <<endl <<"z = "<<*p ;
}
```

OUTPUT

x = 5

y = 10

z = 15

Explanation: In the above program the class A is declared with three public member variables x, y and z. In function main(), a is an object of class A and p is an integer pointer. The three member variables are initialized with 5, 10 and 15 using assignment operation.

The address of variable x is assigned to a pointer. The postfix incrementation operation gives successive memory locations, and values stored at those locations are accessed and displayed. Thus, member variables are stored in successive memory locations. Using the same concept we can also access private members.

7.2.8 Accessing Private Members With Pointers

In the last topic we learned how to access public members of a class using pointers. The public and private member variables are stored in successive memory locations. The following program explains how private members can also be accessed using pointer

PROGRAM : Write a program to access private members like public members of the class using pointers.

```
# include <iostream.h>
# include <conio.h>
class A
{
private:
int j ;
public:
int x ;
int y ;
int z ;
A()
 {
  j = 20 ;
 }

} ;
void main()
{
        clrscr() ;
        A a ;
        Int *p
x = 11 ;
a.y = 10 ;
a.z = 15;
p = &a. x;
p-- ;
cout <<endl <<"j = " <<*p ;
p++ ;
cout <<endl <<"x = " <<*p ;
p++ ;
cout <<endl <<"y = " <<*p ;
p++ ;
cout <<endl <<"z = " <<*p ;
}
```

OUTPUT

j = 20

x = 11

y = 10

z = 15

Explanation: This program is the same as the last one. In addition, the class A has one private member variable. The private member variables can be accessed using functions and no direct access is possible to them. However, using pointers we can access them like public member variables.

7.3 Direct Access to Private Members

So far, we have Initialized private members of the class using constructor or member function of the same class. In the last sub-topic we also learned how private members of the class can be accessed using pointers. In the same way. We can initialize private members and display them. Obtaining the address of any public member variable and using address of the object one can do this. The address of the object contains address of the first element.

PROGRAM : Write a program to initialize the private members and display them without the use of members functions.

```
# include <iostream.h>
# include <conio.h>
class A
{
        private:
        int x ;
        int y ;
} ;
void main()
{
        clrscr()  ;
A a ;
int *p(int*)&a ;
p = 9 ;
p-- ;
cout <<endl <<"x = "<<* p;
p + + ;
cout <<endl << "y = " <<*p ;
}
```

OUTPUT

x = 3

y = 9

Explanation: In the above program, a is an object of class A. The address of the object is assigned to integer pointer p applying typecasting. The pointer p points to private member x. Integer value is assigned to *p i.e., x. By increase, operation next memory location is accessed and nine is assigned i.e. y. The p-- tatement sets memory location of x using cout statement and contents of pointer is displayed.

7.4 Address of Object and Void Pointer

The size of object is equal to number of total member variable declared inside the class the sizeof member function is not considered in object. The object itself contains address of first member variable by obtaining the address we can access the member variables directly, no matter whether they are private. or public. The address of object cannot be assigned to the pointer of the same type. This is because increase operation on pointers will set the address of object according to its size (size of object==size of total member variables). The address of object should be increased according to the basic data type. To overcome this situation a void pointer is useful. The type of void pointer can be changed at runtime using typecasting syntax. Thus, all the member variables can be accessed directly. Consider the following program.

PROGRAM : Write a program to declare void pointers and access member variables using void pointers.

```
# include <iostream.h>
# include <conio.h>
# include <stdio.h>
class A
{
        protected:
        int x ;
        int y ;
} ;
class B : public A
{
        public:
        int z ;
        B() { x = 10 ; y = 20 ; z = 30 ; }
} ;
void main()
{
        clrscr() ;                  // clears the screen
        B b ;                       // object declaration
        int j ;                     // declaration of integer j
        void *p = &b ;              // declaration & initialization of void pointer
        for ( j = 0 ; j<3; j++)     // for loop executed for three times
        printf ("\n member variable[%d] = %d" ,j+1,*((int*)p+j));
```

```
        //cout <<j+1<<" "<<*((int*)p+j)<< endl;
}
```

OUTPUT

member variable [1] = 10

member variable [1] = 20

member variable [1] = 30

Explanation: In the above program, b is an object of derived class B. The address of object b is assigned to void pointer p. Using for loop and typecasting operation, integer values are accessed. The address of void pointer is increased by two bytes. The cout statement now and then will not display the result properly and hence, the printf() statement is used. The typecasting operation is done according to the data type of member variables. Here, all the class member variables are of integer type. In case, the class 'contains variables of integer, float and char type, then it is necessary to typecast according to the sequence of the member variables.

7.5 ARRAY

Array is a collection of variables of same data types in which each element is unique and located in separate. memory locations. The elements of array have different index no or subscript no by which it can be access.

(1) Array Declaration and Initialization

We can declare an array like this :

int x [5];

It tells to the compiler that 'x' is an integer type of array and it can store five integers. The compiler reserves two bytes of memory for each integer array element. In the same way array of different data types are declared as below:

char c[20];

float f[15];

long n[10];

we can initialization the array in following manner :

```
int x[5] = { 1,2, 3,4, 5 };
```

Here, five elements are stored in an array 'x' . The array elements are stored sequentially in separate locations. Then question arises how to call individually to each element from this bunch of integer elements. Reading of array elements begins from zero.

Here

X[0] refers to the 1st element 1

X[1] refers to the 2nd element 2

X[2] refers to the 3rd element 3

X[3] refers to the 4th element 4

X[4] refers to the 5th element 5

7.5.1 Characteristic of Array

(1) The declaration int x[5] will create five variables of integer type in memory. Instead of declaring five variables for five values, the programmer can define them in an array.

(2) All the elements of an array share the same name, and they are access with the help of index number.

(3) The index number in an array plays major role in calling each element.

(4) Any particular element of an array can be modified separately without disturbing other elements.

int x[5]={1,2,3,4,8};

If the programmer needs to replace 8 with 10, he/she is not required to change all other numbers except 8. To carry out this task the statement x[4] = 10 can be used. Here the other three elements are not disturbed.

(5) Any element of an array x[] can be assigned/equated to another ordinary variable or array variable of its type.

For example

b= x[2];

x[2]=x[3];

(a) In the statement b= x[2] or vice versa, value of x[2] is assigned to b; where b is an integer.

(b) In the statement x[2] = x[3] or vice versa, value of x[2] is assigned to x[3] , where both the elements are of the same array.

(6) The array elements are stored in continuous memory locations. The amount of storage required for holding elements of the array depends upon its type and size.

self learning exercises

1. Which of the following statements are true?

a)The String class is implemented as a char array, elements are addressed using the stringname[] convention

b) Strings are a primitive type in Java that overloads the + operator for concatenation

c) Strings are a primitive type in Java and the StringBuffer is used as the matching wrapper type

d) The size of a string can be retrieved using the length property.

2. Given the following declarations

String s1=new String(“Hello”)

String s2=new String(“there”);

String s3=new String();

Which of the following are legal operations?

1. s3=s1 + s2;
2. s3=s1 - s2;
3. s3=s1 & s2
4. s3=s1 && s2

7.5.2 Array of Classes or [ARRAY with in the class]

As we know that array is a collection of similar data types. In the same way, we can also define array of the classes. In such type of array, every element is of class type. Array of class objects can be declared as below:

```
class emp
{
public:
char name[15]; // class declaration
int salary;
char dept[20];
};
class emp e[3]; // declaration of array of class objects
```

In the above example, e[3] is an array of three elements containing three objects of class stud. Each element of e[3] has its own set class member variables i.e., char name[12] int salary and char dept[20] .

PROGRAM : Write a program to display names, rollnos, and grades of 3 students who have appeared in the examination. Declare the class of name, rollnos, and grade. Create an array of class objects. Read and display the contents of the array.

```
# include <conio.h>
# include <stdio.h>
# include <iostream.h>
void min()
{
int k = 0
class stud
{
public :
char name[12] ;
int rollno ;
char grade[2] ;
};
class stud st[3] ;
while (k<3)
```

```
{ clrscr() ;
gotoxy(2,4) ;
cout <<"Name            : " ;
gotoxy(17,4) ;
cin >>st[k].name ;
gotoxy(2,4) ;
cout <<" Roll No. :";
gotoxy(17,5) ;
cin >>st[k].rollno ;
gotoxy(2,6) ;
cout <<"Grade :" ;
gotoxy(17,6) ;
cin>>st[k].grade ;
st[k].grade[1] = '\ 0' ;
puts ( " press any key . . ") ;
getch() ;
k++ ;
}
k = 0 ;
clrscr() ;
cout << "\n Name \t Rollno Grade \n ";
while (k<3)
{
cout<<st[k].name<<"\t"<<st[k].rollno<<" \t" <<st[k].grade<<"\n" ;
k++;
}
}
```

OUTPUT

Name	Rollno	Grade
Balaji	50	A
Manoj	51	B
Sanjay	55	C

Explanation: In the above program, class stud is declared. Its members are char name[12] ; int salary and char grade[2] and all are public. The array e[3] of class stud is declared. The first while loop and cin() statements within the first while loop are used for repetitive data reading. The second while loop and printf() statements within it display the contents of array. In the above program all the member variables are public. If the member variables are private, the above program will not work. Then, we need to declare member functions to access the data.

7.6 Memory Models

The memory model sets the supportable size of code and data areas. Before compiling and linking the source code, we need to specify the appropriate memory model. Using memory models, we can set the size limits of the data and code. C/C++ programs always use different segments for code and data. The memory model you opt decides the default method of memory' addressing. The default memory model is small. Table 11.1 describes properties of all memory models.

(1) Tiny

Use the tiny model when memory is at an absolute premium. All four segment registers (CS, DS, ES, SS) are initialized with same address and all addressing is accomplished using 16 bits. Total memory capacity in this case is 64 K bytes. In short, memory capacity is abbreviated as KB. This means that the code, data, and stack must all fit within the same 64 KB segment. Programs are executed quickly if this model is selected. Near pointers are always used. Tiny model programs can be converted to .COM format.

(2) Small

All code should fit in a single 64 KB segment and all data should fit in a second 64 KB segment. All pointers are 16 bits in length. Execution speed is same as tiny model. Use this model for average size programs. Near pointers are always used.

(3) Medium

All data should fit in a single 64 KB segment. However, the code is allowed to use multiple segments. All pointers to data are 16 bits, but all jumps and calls require 32 bit addresses. With this, access to data is fast. However, slower program execution is observed with this model. This, model is suitable for big programs that do not keep a large amount of data in memory. Far pointers are used for code but not for data.

(4) Compact

All code should fit in 64 KB segment but the data can use multiple segments. However, no data item can surpass 64 KB. All pointers to-data are 32 bits, but jumps and calls can use 16 bit addresses. Slow access to data and quick code execution will be observed in this setting. Compact model is preferred if your program is small but you require pointing large amount of data: The compact model is the opposite of the medium model. Far pointers are preferred for data but not for code. Code is limited to 64 KB, while data has a 1 MB range. All functions are near by default and all data pointers are far by default.

(5) Large

Both code and data are allowed to use multiple segments. All pointers are 32 bits in length. However, no single data item can exceed 64 KB. Code execution is slower. Large model is preferred for very big programs only. Far pointers are used for both code and data, specifying both a 1 MB range. All functions and data pointers are far by default.

(6) Huge

Both code and data are allowed to use multiple segments. Every pointer is of 32 bits in length. Code execution is the slowest.

Huge model is preferred for very big programs only. Far pointers are used for both code and data. Turbo C++ usually bounds the size of all data to 64 K. The huge memory model sets

aside that limit, permitting data to hold more than 64 K. The huge model permits multiple data segments (each 64 K in

size), up to I MB for code, and 64 K for stack. All functions and data " pointers are supposed to be far.

Memory models.

Sr. No.	Memory Model	Code	Segments		Type of pointer	
			Data	Stack	Code	Data
1	Tiny		64 K		near	near
2	Small	1 MB	64-K		near	near
3	Medium	64 K	64 K		far	near
4	Compact	1 MB	1 MB		near	far
5	Large	1 MB	1 MB		far	far
6	Huge	1 MB	64 K each 64 K stack		far	far

7.7 The new and delete Operators

So far, we have used the new and delete operators in short programs, but for applying them in huge applications we need to understand them completely. A small mistake in syntax may cause a critical error and possibly corrupt memory heap. Before studying memory heap in detail let us repeat few points regarding new and delete operators

(1) The new operator not only creates an object but also allocates memory.

(2) The new operator allocates correct amount o~ memory from the heap that is also called as a free store.

(3) The object created and memory allocated by using new operator should be deleted by the delete operator otherwise such mismatch operations may corrupt the heap or may crash the system. Accord ing to ANSI standard, it is a valid outcome for this invalid operation and the compiler should have routines to handle such errors.

(4) The delete operator not only destroy the object but also releases allocated memory.

(5) The new operator creates an object and it remains in the memory until It IS released using delete operator sometimes the object deleted using the-delete operator remains in memory.

(6) If we send a null pointer to delete operator, it is secure. Using delete to zero has no result.

(7) The statement delete x does not destroy the pointer x. It destroys the object associated with it.

(8) Do not apply C functions such as malloc() , realloc() or free() with new and delete operators. These functions are unfit to object-oriented techniques.

(9) Do not destroy the pointer repetitively or more than one time. First time the object is destroyed and memory is released. If for the second time the same object is deleted, the bject is sent to the destructor and no doubt, it will corrupt the heap.

(10) If the object created is not deleted, it occupies the memory unneces sarily. It is a good habit to destroy the object and release the system resources.

Difference Between new and malloc()

new	malloc()
Creates objects Returns pointer of relevant type It ,is possible to overload new operator	Allocates memory Returns void pointer Malloc () cannot be overloaded

PROGRAM : Write a program to allocate memory to store 3 integers. Use new and delete operators for allocating and deallocating memory. Initialize and display the values.

```
# include <iostream.h>
# include < conio.h>
int main()
{
        clrscr() ;
        inte i,*p;
        p= &i;
        p= new int[3] ;
        *p=2;                   // first element
        *(p+1)=3                // second element
        *(p+2)=4                // third element
cout << "Value Address";
for (int x = 0; x<3; x++)
cout <<endl<<*(p+x)" \t" << (unsigned) (p+x) ;
delete []p;
return 0;
}
```

OUTPUT

Values	Address
2	3350
3	3352
4	3354

Explanation: In the above program, integer variable i and pointer *p are declared. The pointer p is initialized with address of variable i and using new operator memory for three integers is allocated to it. The pointer *p can hold three integers in successive memory locations. The pointer variable is initialized with numerical values. The for loop is used to display the contents of the pointer *p. The delete operator releases the memory.

PROGRAM : Write a program to allocate memory for two objects. Initialize and display the contents and deallocate the memory.

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

```
# include <conio.h>
struct boy
{
        char *name;
        int age;
};
int main()
{
        clrscr();
boy *p;
p=new boy[2] ;
p->name="Rahul1";
 p->age=20;
(p+1)->name="Raj";
(p+1)->age = 21;
for (int x=0;x<2;x++)
{
cout<<"\nName :"<<(p+x)->name<<endl<<"Age:"<<(p+x)->age;
}
delete []p;
return 0;
}
```

OUTPUT

```
Name   : Rahul
Age    : 20
Name   : Raj
Age    : 21
```

Explanation: In the above program, structure boy is declared and memory for two objects is allocated to the pointer p. The pointer p is initialized and the first for loop displays the contents of the pointer on the screen. Finally, the delete operator de-allocates the memory.

7.8 Heap Consumption

The heap is used to allocate memory during program execution i.e., run-time: In assembly language, there is no such thing as a heap. All memory is for programmer and he/she can use it directly. In various ways, C/C++ has a better programming environment than assembly language. However, a cost has to be paid to use either C or C++. The cost is separation from machine. We cannot use memory anywhere; We need to ask for it. The memory from which we receive is called the heap.

The C/C++ compiler may place automatic variables on the stack. It may store static variables

earlier than loading the program. The heap is the piece of memory that we allocate.

Local variables are stored in the stack and code is in code space. The local variables are destroyed when a function returns. Global variables are stored in the data area. Global variables are accessible by all functions. The heap can be thought of as huge section of memory in which large number of memory, locations are placed sequentially

As stated earlier all local variables are stored in the stack. As soon as the function execution completes, local variables are destroyed and the stack becomes empty. The heap is not cleared until program execution completes. It is the user's task to free the memory. The memory allocated from heap remains available until the user explicitly deallocates it.

While solving problems related to memory allocation, we always believe that there is large memory available and the heap is at no time short of memory. It is bad for a program to depend on such guesswork, which may cause an error in application. In C, the function, malloc() is used to allocate memory and if this function fails to allocate memory, returns null pointer. By checking the return value of function malloc() , failure or success of the memory allocation is tested and appropriate sub-routines are executed.

7.9 Overloading new and delete operators

In c++ any time when we are concerned with memory allocation and deallocation, the. new and delete operators are used. These operators are invoked from compiler's library functions. These operator are part of C++ language and are very effective. Like other operators the new and delete operators are overloaded. The program given next explain this.

PROGRAM : Write a program to overload new and delete operators.

```
# include <iostream.h>
# include <stdlib.h>
# include <new.h>
# include <conio.h>
void main()
{
        clrscr() ;
        void warning() ;
        void *operator new (size_t, int) ;
        void operator delete (void*) ;
        char *t = new('#') char[10] ;
        cout<<endl<<"First allocation:p="<<(unsigned)long(t)<<endl;
        for (int k = 0 ; k<10; k++)
        cout <<t[k] ;
        delete t;
        t= new('*') char [64000u] ;
        delete t;
}
void warning()
```

```
{
        cout<<"\n insufficient memory" ;
        exit(1) ;
}
voidd *operator new (size_t os, int setv)
{
        void *t;
        t=ma11oc(os) ;
        if (t==NULL) warning() ;
        memset (t, setv, os) ;
        return (t);
}
void operator delete (void *ss)  {        free (ss) ;        }
```

OUTPUT

First allocation : p=3376

##########

Insufficient memory

Explanation : in the above program the new and delete operators are overloaded . the size_t is used to determine the size of object . the new operator calls warning() function when malloc() function return null.

See the statement t=new('*') char[64000u]. when memory allocation is requested by this statement, the new operator fails to allocate memory and calls the function warning(). The delete operator when called releases the memory using free() function. Internally, in this program malloc() and free() functions are used to allocate and deallocate the memory.

7.10 Execution Sequence of Constructors and destructor

Overloaded new and delete operators function within the class and are always static. We know that static function can be invoked without specifying object. Hence, this pointer is absent in the body of static functions. The compiler invokes the overloaded new operator and allocates memory before executing constructor. The compiler also invokes overloaded delete operator function and deallocates memory after execution of destructor. The following program illustrates this concept.

PROGRAM :Write a program to display the sequence of execution of constructor and destructors in classes when new and delete operators are overloaded.

```
# include <iostream.h>
# include <stdlib.h>
# include <new.h>
# include <conio.h>
class boy
{
```

```
        char name[10] ;
        public :
        void *operator new (size_t) ;
        void operator delete (void *q) ;
        boy() ;
~boy() ;
};
char limit [sizeof (boy)] ;
boy::boy()
{
        cout <<endl<< "In constructor ";
}
boy::~boy()
{
        cout <<endl<<"In Destructor " ;
}
void *boy::operator new ( size_t s )
{
        cout <<endl<< " In boy::new operator";
        return limit ;
}
void boy::operator delete (void * q)
{       cout << endl<< " In boy::delete operator"; }
void main()
{
        clrscr() ;
        boy *e1 ;
        e1 = new boy ;
        delete e1 ;
}
```

OUTPUT

```
In boy :: new operator
In Constructor
In Destructor
In boy :: delete operator
```

Explanation: In this program, the new and delete operators are overloaded. The class also has constructor and destructor. The overloaded new operator function is executed first followed by

class constructor. We know that constructor is always used to initialize members. The constructor also allocates memory by calling new operator implicitly. Here, the new operator is overloaded. As soon as control of program reaches to constructor it invokes overloaded new operator before executing any statement in constructor.

Similarly, when object goes out of scope destructor is executed. The destructor invokes overloaded delete operator to release the memory allocated by new operator.

7.11 Specifying Address Of an Object

The compiler assigns address to objects created. The programmer has no power over address of the object. However, it is possible to create object at memory location given by the program explicitly. The address specified must be big enough to hold the object. The object, created in this way must destroyed by invoking destructor explicitly. The following program clears this concept.

PROGRAM : Write a program to create object at given memory address.

```
# include <iostream.h>
# include <constream.h>
class data
{
        int j;
        public:
        data(    int k)    {j=k;    }
        ~data()
        {
        }
void *operator new(size_t void *u)
{
        return (data * )u;
}
void show()
{ cout << "j = " << j; }
};
void main()
{
        clrscr() ;
        void *add ;
        add=(void*)0x420 ;
        data *t = new(add)data(10);
        cout <<"\n Address of object : "<<t;
        cout << endl ;
        t->show() ;
```

```
        t->data::~data() ;
}
```

OUTPUT

Address of object : 0x8f800420

j = 10

Explanation: In the above program, the operator new is overloaded. The void pointer * add is declared and initialized with address Ox420:"The pointer object *p is declared and address is assigned to it. The new operator is also invoked to allocate memory. In the same statement constructor is also invoked and a value is passed to it. The member function show() displays the value of a member variable. Finally, the statement t->data::~data(); invokes the destructor to destroy the object. We can confirm the address of object by displaying it. The address of object would be same as the specified one.

7.12 Dynamic Objects

c++ supports dynamic memory allocation. C++ allocates memory and initializes the member variables. An object can be created at run-time. Such object is called as dynamic object. The construction and destruction of dynamic object is explicitly done by the programmer. The dynamic objects can be created and destroyed by the programmer. The operator new and delete are used to allocate and deallocate memory to such objects. A dynamic object can be created using new operator as follows:

ptr= new classname;

The new operator returns the address of object created and it is stored in the pointer pt r. The variable ptr is a pointer object of the same class. The member variables of object can be accessed using pointer and->(arrow) operator A dynamic object can be destroyed using delete operator as follows:

```
        delete ptr;
```

It destroys the object pointed by pointer ptr. It also invokes the destructor of a class. The follow

PROGRAM : Write a program to create dynamic object

```
# include <iostream.h>
# include <constream.h>
class data
{
        int x,y;
        public :
        data()
{
        cout<<"\n Constructor ";
        x = 10 ;
        y = 50;
}
```

```
~data()
{ cout <<"\n Destructor ";}
void display()
{
        cout <<"\n x =" << x;
        cout <<"\n y =" << y ;
}
};
void main()
{
        clrscr() ;
        data *d;                    // declaration of object pointer
        d = new data;               // dynamic object
        d->display();
        delete d;                   // deleting dynamic object
}
```

OUTPUT

```
Constructor
x = 10
y = 50
Destructor
```

Explanation: The d is a pointer object. The statement d= new data creates an anonymous object and assigns its address to pointer d, Such creation of object is called as dynamic object. The constructor is executed when dynamic object is created. The statement d->display() invokes the member function and contents of members are displayed. The statement delete d destroys the object by releasing memory and invokes destructor.

CALLING CONVENTION

Calling convention means how parameters are pushed on the-stack when a function is invoked. Table describes calling convention method of few languages.

Table calling convention

Language	Order	Parameter passed
Basic	In order (L ->R)	By address
Fortran	In order (L ->R)	By address
C	In reverse order (R->L)	By value and address
C++	In reverse order (R->L)	By value and address & ref.

In C/C++ parameters are passed from right to left whereas in other languages such as Basic, Fortran calling convention is from left to right order. The following program explains calling convention of C++.

PROGRAM : Write a program to demonstrate calling convention of C++.

```
#include <iostream.h>
#include <constream.h>
void main()
{
clrscr();
void show (int, int) ;
int x = 2, y = 3;
show (x,y) ;
}
void show (int x, int y)
{
        cout <<x;
        cout <<endl;
        cout <<y;
}
```

OUTPUT

2

3

Explanation: In this program, integer variables x and y are initialized to two and three respectively. The function show () is invoked and x and yare, passed as per the statement show (x , y). The parameters are passed from right to left. The variable y is passed first followed by x.

7.13 SUMMARY

(1) Like C, in C++ variables are used to hold data values during program execution. Every variable when declared occupies certain memory locations.

(2) Pointer saves the memory space. The execution time with pointer is faster because data is manipulated with the address.

(3) The indirection operator (*) is also called the deference operator. When a pointer is dereferenced, the value at that address stored by the pointer is retrieved.

(4) The & is the address operator and it represents the address of the variable. The address of any variable is a whole number.

(5) Pointers can also be declared as void type. Void pointers cannot be dereferenced without explicit type conversion. This is because being void the compiler cannot determine the size of the object the pointer points to.

(6) The pointer this is transferred as an unseen parameter in all calls to non-static member function .The keyword this is a local variable, always present in the body of any non-static member function.

(7) Array is a collection of similar data types in which each element is a unique one and located in separate memory locations.

7.14 Glossary

array : collection of data items, all of the same type, in which each item's position is uniquely designated by an integer

new**:** Java language keyword used to create, or instantiate, an object from a class definition.

class method :A method that is invoked without reference to a particular object. Class methods affect the class as a whole, not a particular instance of the class. Also called a *static method*.

7.15 Further Readings

1. Introduction To Object Oriented Programming And C++ ,Kanetkar , BPB,**ISBN:** 8176568635
2. A complete Reference to C/C++,Herbert Shildt
3. C++: An introduction to programming by Jense Liberty Tim Keogh: BPB Publications, New Delhi
4. OO Programming in C++ by Robert Lafore: , Galgotia Publications Pvt. Ltd., Daryaganj,New Delhi
5. Object Oriented Programming Using C++, Sanjeev Sofat, Cyber Tech. Publication, New Delhi
6. Object Oriented Programming in C++ by E. Balaguruswamy, TMH Publishing Co. Ltd.,New Delhi

7.16 Answers to Self Learning Exercises

1. B

2. A

7.17 Unit End Questions

(1) What are pointers?

(2) What is void pointer?

(3) What is wild pointer?

(4) What is this pointer?

(5) What are the features and uses of pointers?

(6) In which situation does the pointer become harmful?

(7) Explain any two characteristics of pointers.

(8) What are arrays?

(9) How can array initialization be carried out using function?

(10) Explain any three characteristics of arrays.

(11) How can private members be accessed using pointers?

(12) Why is declaration of void variable not permitted?

(13) What do you mean by caning conventions?

(14) Explain the process of calling convention in C/C++.

(15) Explain the difference between the operator new and malloc () function.

UNIT - IX

Generic Programming With Template

Structure of the Unit

9.0 Objective

After reading this unit, you will be able to

- Understand the concept of template
- Function template and class template
- Different features of template function
- Difference between template and macro

9.1 Introduction

Template is one of the most useful characteristics of C++. Few old compilers do not allow template mechanism. Templates are part of ANSI C++ standard. All major compilers support templates in their new versions. Instantiation is the activity of creating particular type using templates. The specific classes are known as instance of the template. The template provides generic programming by defining generic classes. In templates, generic data types are used as arguments and they can handle a variety of data types.

A function that works for all C++ data types is called as generic function. Templates help the programmer to declare group of functions or classes. When used with functions they are called function templates. For example, we can create a template for function square(). It helps us to calculate square of a number of any data type including int, float, long, and double. The templates associated with classes are called as class templates. A template is almost same as macro and it is type secured.

9.1.1 Need of Template

Template is a technique that allows using a single function or class to work with different data types. Using template we can create a single function that can process any type of data i.e., the formal arguments of template functions are of template (generic) type. They can accept data of any type such as int, float, long etc. Thus, a single function can be used to accept values of different data types. Normally, we overload function when we need to handle different data types. This approach increases the program size. The disadvantages are that not only program length is increased but also more local variables are created in the memory. Template safely overcomes all the limitations occurring while overloading function and allows better flexibility to the program. The portability provided by a template mechanism can be extended to classes. The following sections describe use of template in different concepts of C++ programming. Let us understand the process of declaration of a template.

9.1.2 Definition of Class Template

To declare a class of template type, following syntax is used.

Template Declaration

template class <T>

class name of class

{

//class data member and function

}

The first statement template class<T> tells the compiler that the following class declaration can use the template data type. T is a variable of template type that can be used in the class to define

variable of template type. Both template and class are keywords. The <> (angle bracket) is used to declare variables of template type that can be used inside the class to define variables of template type. One or more variables can be declared separated by comma. Templates cannot be declared inside classes or functions. They must be global and should not be local.

T k;

Where, k is the variable of type template. Most of the authors use T for defining template; instead of T, we can use any alphabet.

PROGRAM : Write a program to show values of different data types using overloaded constructor.

```
# include <iostream.h>
# include <conio.h>
class data
{
public:
data(char c)
{       cout <<"\n"<< " c = "<<c <<"Size in bytes : "sizeof (c) ;
 }
data(int c)
{
cout <<"\n " << "c = "<<c <<" Size in bytes :"<<sizeof(c) ;
}
data(double c)
{
cout <<"\n" <<" c = "<<c <<" Size in bytes : "<<sizeof(c) ;
}
};
int main()
{       clrscr() ;
        data h('A') ;           // passes character type data
        data i(100) ;           // passes integer type data
        data j(68.2) ;          // passes double type data
        return 0;
}
```

OUTPUT

```
c = A size in bytes         : 1
c = 100 Size in bytes       : 2
c = 68.2 Size in bytes      : 8
```

Explanation: In the above program, the class data contains three overloaded one argument

constructor. The constructor is overloaded for char, int, and double type. In function main() , three objects h, i and j are created and values passed are of different types. The values passed are char, int, and double type. The compiler invokes different constructors for different data types. Here, in order to manipulate different data types we require overloading the constructor i.e., defining separate function for each non-compatible data type. This approach has the following disadvantages:

(1) Redefining the functions separately for each data type increases the source code and requires more time.

(2) The program size is increased. Hence, occupies more disk space.

(3) If function contains bug, it should be corrected in every function.

From the above program it is clear that for each data type we need to define separate constructor function. According to data type of argument passed respective constructor is invoked. c++ provides templates to overcome such problems and helps a programmer to develop generic program. The same program is illustrated with template as follows.

PROGRAM : Write a program to show values of different data types using constructor and template.

```
# include <iostream.h>
# include <conio.h>
template <class T>

class data
{
        public :
        data(T c)
        {
        cout <<"\n"<< " c = "<<c <<" Size in bytes : " <<sizeof(c ) ;
        }
};

int main()
{
        clrscr()
        data <char> h('A') ;
        data <int> i(100) ;
        data <float> j(3.12) ;
        return 0;
}
```

OUTPUT

c = A Size in bytes : 1

c = 100 Size in bytes : 2

c = 3.12 Size in bytes : 4

Explanation: In the above program, the constructor contains variable of template T. The template class variable can hold values of any data type. While declaring an object the data type name is given before the object. The variable of template type can accept values of any data type. Thus, the constructor displays the actual values passed. The template variable c can hold values of any data type. The value and space in bytes required by these variables are displayed as the output. The size of data type changes according to the data types used in the program.

In the above program, different values are passed using constructor, but for all data types same template function is used.

9.2 Function Template

9.2.1 Normal Function Template

In the last section, we saw how to make a template class. In the same fashion a normal function (not a member function) can also use template arguments. The difference between normal and member function is that normal functions are defined outside the class. They are not members of any class and hence can be invoked directly without using object of dot operator. The member functions are the class members. They can be invoked using object of the class to which they belong. The declaration of template member function is described later in this chapter. In C++ normal functions are commonly used as in C. However, the user who wants to use C++ as better C can utilize this concept. The declaration of template normal function can be done as follows.

Normal Template Function Declaration

```
template class <T>
Function_name ( )
{
        // code
}
```

The following program shows practical working of template function.

PROGRAM : Write a program to define normal template function.

```
# include <iostream.h>
# include <conio.h>
template <class T>
void show ( T x )
{       cout << "\n x = "<<x ;
}
void main()
{       clrscr() ;
        char c = 'A' ;
```

```
        int i = 65;
        double d = 65.254 ;
        show(c) ;
        show(i) ;
        show(d) ;
}
```

OUTPUT

x = A

x = 65

x = 65.254

Explanation: Before the body of the function show(), template argument T is declared. The function show() has one argument x of template type. As explained earlier, the template type variable can accept all types of data. Thus,the normal function show() can be used to display values of different data types. In function(), the show() function is invoked and char, int and double type of values are passed. The same is displayed in the output.

we are now familiar with utilities of templates. One more point to remember is that when we declare a class template, we can define class data member of template type as well as the member function of the class can also use the template member. For making member function of template type, no separate declaration is needed. The following program explains the above point.

PROGRAM :

Write a program to define data members to template type.

```
# include <iostream.h>
# include <conio.h>
template <class T>
class data
{
        T x;
        public :
        data(T u)
        {
        x =u;
        }
void show (T y)
        {
                cout <<"x = " <<x;
                cout <<"y= "<<y<<"\n";
}
```

```
};
int main()
{
        clrscr() ;
        data <char> c('B') ;
        data <int> i(100) ;
        data <double> d(48.25);
        c.show('A') ;
        i.show(65) ;
        d.show(68.25) ;
        return 0;
}
```

OUTPUT

x = B y=A

x= 100 y=65

x = 48.25 y=68.25

Explanation: In this program, before declaration of class data, template <class T> is declared. This declaration allows entire class including member function and data member to use template, type argument .We have declared data member x of template type. In addition The one argument constructor and member function show() also have one formal argument of template type.

PROGRAM

Write a program to create square() function using template.

```
# include <iostream.h>
# include <conio.h>
# include <class S>
class sqr
{
        public:
sqr (S c)
        {
        cout <<"\n"<<"c="<<c*c;
        }
};
int main()
{
clrscr() ;
```

```
sqr <int> i(25) ;
sqr <float> f(15.2) ;
return 0;
}
```

OUTPUT

```
c = 625
c = 231.039993
```

Explanation: In the program in the previous page, the class sqr is declared. It contains a constructor with one argument of template type. In main() function, the object i indicates int type and f indicates float type. The object i and f invokes the constructor sqr() with values 15 and 15.2. The constructor displays the squares of these numbers.

9.2.2 Working of Function Templates

In the last few examples, we learned how to write a function template that works with all data types. After compilation, the compiler cannot guess with which-type of data the template function will work. When the template function is called at that moment, from the type of argument passed to the template function, the compiler identifies the data type. Every argument of template type is then replaced with the identified data type and this process is called as instantiating. Thus, according to different data types respective versions of template functions are created. Though the template function is compatible for all data types, it will not save any memory. When template functions are used, four versions of functions are used. They are data type int, char, float, and double. The programmer need not write separate functions for each data type.

9.3 Class Template with More Parameters

Like functions, classes can be declared to handle different data types. Such classes are known as class templates. These classes are generic type and member functions of these classes can operate on different data types. The class template may contain one or more parameters of generic data type. The arguments are separated by comma with template declaration. The declaration is as follows:

Template with Multiple Parameters

```
template <class T1, class T2>
class name_of_Class
{
// class declarations and definitions
}
```

PROGRAM :

Write a program to define a constructor with multiple template variables.

```
# include <iostream.h>
# include <conio.h>
template <class T1, class T2>
class data
{       public :
```

```
        data (T1 a,T2 b)
        {
        cout <<"\n a = "<<a <<" b = "<<b;
        }
};
int main()
{
clrscr() ;
data <int, float > h(2,2.5) ;
data <int, char> i(15,'C') ;
data <float, int > j (3.12 , 50) ;
return 0;
}
```

OUTPUT

a = 2 b = 2.5

a = 15 b = C

a = 3.12 b = 50

Explanation: In the above program, the data constructor has two arguments of generic type. While creating objects, type of arguments is mentioned in the angle bracket. When arguments are more than one they are separated by comma. Consider the statement data <int , float> h (2,2.5). It tells the compiler that the first argument is of integer type and second argument is of float type. When objects are created, constructors are called and values are received by the template arguments. The output of the program is given above.

Function template with more parameters

In previous examples, we defined constructors with arguments. In the same fashion, we can also define member functions with generic arguments. The format is given below:

Function Template Declaration

```
template <class T>
return_data_type function_name (parameter of template type)
{
 statement 1;
 statement2;
 statement3;
}
```

PROGRAM :

Write a program to display the elements of integer and float array using template variables with member function.

```
# include <iostream.h>
# include <conio.h>
template <class T1, class T2>
```

```
class data
{
        public :
void show (T1 a, T2 b)
        {
        cout <<"\na = "<<a <<"b = "<<b ;
        }
};
int main()
{
clrscr() ;
int I[] = {3, 5, 2, 6, 8, 9, 3} ;
float f[] = {3. 1, 5.8 , 2.5 , 6.8, 1.2 , 9.2 , 4.7 } ;
data <int , float > h;
for (int m = 0 ; m <7; m++)
h. show (i[m], f [m] ) ;
return 0;
}
```

OUTPUT

a = 3 b = 3.1

a = 5 b = 5.8

a = 2 b = 2.5

a = 6 b = 6.8

a = 8 b = 1.2

a = 9 b = 9.2

a = 3 b = 4.7

Explanation: In the above program, array elements of integer and float are passed to function show(). The function show() has two arguments of template type i.e., a and b. The show() function receives integer and float values and displays them. The output of the program is shown above.

9.4 Overloading of Template Function

A template function also supports overloading mechanism. It can be overloaded by normal function or template function. While invoking these functions an error occurs if no accurate match is met. No implicit conversion is carried out in parameters of template functions. The compiler follows the following rules for choosing appropriate function when program contains overloaded functions.

(1) Searches for accurate match of functions; if found it is invoked.

(2) Searches for a template function through which a function that can be invoked with

accurate match can be generated; if found it is invoked.

(3) Attempts normal overloading declaration for the function.

(4) In case no match is found, an error will be reported.

PROGRAM :

Write a program to overload a template function.

```
# include <iostream.h>
# include <conio.h>
template <class A>
void show(A c)
{
        cout <<"\n Template variable c = "<<c;
}
void show(int f)
{
        cout <<"\n Integer variable f = "<<f;
}
int main()
{
clrscr() ;
show('C') ;
show('C') ;
show(50) ;
show(50.25) ;
return 0;
}
```

OUTPUT

```
Template variable c = C
Integer variable f = 50
Template variable c = 50.25
```

Explanation: In the above program, the function show() is overloaded. One version contains template arguments and other version contains integer variables. In main() , the show() function is invoked three times with char, int and float and values are passed. The first call executes the template version of function show() , the second call executes integer version of function show() and the third call again invokes the template version of function show(). Thus, in the above fashion template functions are overloaded.

Member function template

In the previous example, the template function defined were in line i.e., they were defined inside the class. It is also possible to define them outside the class. While defining them outside, the

function template should define the function and the template classes are parameterized by the type argument.

PROGRAM :

Write a program to define of member function template outside the class and invoke the function.

```
# include <iostream.h>
# include <conio.h>
template <class T>
class data
{
        public:
        data (T c) ;
} ;
        template <class T>
        data <T> :: data (T c)
                {
                cout <<"\n"<< " c = "<<c ;
                }
int main()
{
clrscr() ;
data <char> h('A') ;
data <int>      i(100) ;
data <float > j(3.12) ;
return 0;
}
```

OUTPUT

```
c = A
c = 100
c = 3.12
```

Explanation: In the above program, the constructor is defined outside the class. When we define outside the class, the member function should be preceded by the template name as given below:

```
Statements of program
template <class T>
data<T>::data (T c)
```

9.5 Recursion with Template Function

Like normal function and member function, the template function also supports recursive execution of itself. The following program explain this:

Explanation: The template function generates random number and displays it each time when the function display() is invoked. The function rand() defined in stdlib.h is used. The function calls it self recursively until the value of d becomes 1. The assert() statement checks the value of d and terminates the program when condition is satisfied. The assert() is defined in assert.h. We can also use if statement followed by exit() statement as given in comment statement

9.6 Class Template with Overloaded Operators

The template class permits to declare overloaded operators and member functions. The syntax for creating overloaded operator function is similar to class template members and functions. The following program explains the overloaded operator with template class.

PROGRAM :

Write a program to overload + operator for performing addition of two template based class objects.

```
# include <iostream.h>
# include <conio.h>
template <class T>
class num
{       private:
        T number:
        public:
        num()
        {       number = 0;     }
void input()
{
        cout <<"\n Enter a number : ";
        cin >>number ;
}
num operator+(num) ;
void show()
{       cout <<number ;
} ;
template <class T>
num <T> num <T> : : operator + (num <T> c)
{
        num <T> tmp ;
        tmp. number = number+c.number ;
        return (tmp) ;}
void main()
{
        clrscr() ;
        num <int> n1, n2, n3 ;
        n1. input() ;
        n2. input() ;
        n3= n1+n2 ;
        cout <<"\n\t n3 = ";
        n3. show() ;
}
```

OUTPUT

```
Enter a number : 8
Enter a number : 4
n3 = 12
```

Explanation: In the above program, class num is declared with template variable number. The input () function is used to read number through the keyboard. The operator+() function performs addition of elements of two template objects.

In function main(), n1, n2 and n3 are three objects of num class. The statement n3=n1+n2 invokes the overloaded operator and addition of two objects is performed. The show () function displays the contents of the object.

9.7 Class Templates and Inheritance

The template class can also act as base class. When inheritance is applied with template class it helps to compose hierarchical data structure known as container class.

(1) Derive a template class and add new member to it. The base class must be of template type.

(2) Derive a class from non-template class. Add new template type member to derived class.

(3) It is also possible to derive classes from template base class and omit the template features of the derived classes.

As said earlier, the template characteristics of the base class can be omitted by declaring the type while deriving the class. All the template based variables are substituted with basic data types. The following declaration illustrates derivation of a class using template featured base class.

```
Template <class TL, ...>
class XYZ
{
// Template type data members and member functions
}
Template <class TL, . . .>
class ABC : public XYZ <TL,...>
{
// Template type data members and member functions
}
```

PROGRAM :

Write a program to derive a class using template base class.

```
# include <iostream.h>
# include <constream.h>
template <class T>
class one
```

```
{
        protected :
        T x, y ;
        Void display()
        {
                cout <<x ;
        }
} ;
template <class S>
class two : public one <S>
{
        S z;
        public ;
        two (S a, S b, S c)
        {
        x = a;
        y = b;
        z = c;
}
void show()
{
cout << " \n x = " <<x <<" y="<<"z = " <<z ;
}
};
void main()
{
        clrscr() ;
        two <int> i(2, 3, 4) ;
i.show() ;
two <float> f(1.1, 2.2,3.3) ;
f.show() ;
}
```

OUTPUT

x = 2 y = 3 z = 4

x = 1.1 y = 2.2 z = 3.3

Explanation: In the above program the class one is a base class and class two is a template-derived class. Consider the statement given below:

template <class S>

class two: public one <S>

Using the above statement derivation is carried out. The base class name is followed by template class name S. The function show () is a member of derived class two(). In function main(), and I and f are objects of class two. The object is used to pass integer elements and f is used to pass float numbers.

9.8 Guidelines for Templates

(1) Templates are applicable when we want to create type secure class that can handle different data types with same member functions.

(2) The template classes can also be involved in inheritance. For example

template <class T>

class data : public base<T>

Both data and base are template classes. The class data is derived from template class base.

(3) The template variables also allow us to assign default values. For example,

template <class T, int x=20>

Class data t

num [x];

}

(4) The name of the template class is written differently in different situations.
While class declaration, it is declared as follows:

class data { };

For member function declaration is as follows:

void data <T> :: show (T d) { }

Where, show() is a member function.

Finally, while declaring objects the class name and specific data is specified before object name.

data < int> i1 // object of class data supports integer values

data < float> f1 // object of class data supports float values

(5) All template arguments declared in template argument list should be used for definition of formal arguments. If one of the template arguments is not used, the template will be specious.

9.9 Difference Between Template and Macros

(1) Macros are not type safe i.e., a macro defined for integer operation cannot accept float data. They are expanded with no type checking.

(2) It is difficult to find errors in macros.

(3) In case a variable is post-incremented or decremented, the operation is carried out twice.

```
# define max(a)  a+ ++a
void main ()
{
 int a=10, c;
        c=max (a) ;
        cout<<c;
}
```

The macro defined in the above macro definition is expanded twice. Hence, it is a serious limitation of macros. The limitation of this program can be removed using templates as shown below:

PROGRAM :

Write a program to perform incrimination operation using template.

```
# include <iostream.h>
# include <constream.h>
template <class T>
T max (T k )
{
        ++k;
        return k ;
}
void main()
{
        clrscr() ;
        int a = 10, c ;
        c = max (a) ;
        cout << c ;
}
```

OUTPUT

11

Explanation: In the above program, template function max() is defined. An integer value is passed to this function. The function max() increments the value and returns it.

Self Learning Exercises (State True/False)

1. The class template may contain one or more parameters of generic data type.
2. Function template can't be defined with one or more parameters.
3. A template function also supports overloading mechanism.

9.10 Linked List with Templates

The linked lists are one among popular data structures. The following program explains the working of linked list with templates.

PROGRAM : Write a program to create linked list with template class.

```
# include <iostream.h>
# include <constream.h>
template <class L>
class list
{
        L t;
        list  *next ;
        public ;
        list ( L t) ;
}
void add_num (list *node)
{
node->next= this;
next = NULL ;
}
list *obtainnext() {return next ; }
L getnum()        { return t ;}
} ;
template <class L> list <L>::list (L y)
{
        t = y
        next = 0 ;
}
void main()
{
        clrscr() ;
        list <int> obj(0) ;
        list <int> *opt ,*last ;
        last = &obj ;
        int j = 0;
while (j <15)
{
        opt = new list<int>(j+1) ;
        opt->add_num(last ;
        last = opt ;
```

```
        j++ ;
}
opt = &obj;
j = 0;
while (j<15)
{
        cout <<opt->getnum() << " " ;
        opt = opt->obtainnext() ;
        j++ ;
}
}
```

OUTPUT

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

Explanation: In the above program, the class template is declared in the same way as in previous programs. The class list has two arguments. One is of template L type(t) other is of object of the class list(*next). The pointer next points to the next element of the linked list. The add_num() function is used to add successive numbers in the linked list. The function obtainnext() returns address of the next element. The function declarator of obtainnext() is preceded by symbol * because it returns address of the element. In function add_num() the pointer next is initialized with zero. If we do not initialize the next with zero it will become wild pointer and may point to any location of the system. Using loop, successive numbers are added and displayed.

9.11 Summary

(1) Template is one of the most useful characteristics of C++. It is newly added in C++. Instantiation is the activity of creating particular type using templates. The specific classes are known as instance of the template. The template provides generic programming by defining generic classes. In templates, generic data types are used as arguments and they can handle a variety of data types.

(2) Declaration and definition of every template class starts with the keyword template followed by parameter list.

(3) The class template may contain one or more parameters of generic data type. The arguments are separated by comma with template declaration.

(4) Function template can be defined with one or more parameters.

(5) A template function also supports overloading mechanism. It can be overloaded by normal function or template function.

(6) It is also possible- to define member function definition outside the class. While defining them outside, the function template should define the function and the template classes are parameterized by the type argument. .

9.12 Glossary

template :The template provides generic programming by defining generic classes. In templates, generic data types are used as arguments and they can handle a variety of data types.

9.13 Further Readings

1. Introduction To Object Oriented Programming And C++ ,Kanetkar , BPB,**ISBN:** 8176568635
2. A complete Reference to C/C++,Herbert Shildt
3. C++: An introduction to programming by Jense Liberty Tim Keogh: BPB Publications, New Delhi
4. OO Programming in C++ by Robert Lafore: , Galgotia Publications Pvt. Ltd., Daryaganj,New Delhi
5. Object Oriented Programming Using C++, Sanjeev Sofat, Cyber Tech. Publication, New Delhi
6. Object Oriented Programming in C++ by E. Balaguruswamy, TMH Publishing Co.Ltd.,New Delhi

9.14 Answers to Self Learning Exercises

1. True

2. False

3, True

9.15 Unit End Questions

Answer the following question.

(1) What are templates?

(2) How can normal function be declared as template function?

(3) How can member function be declared as template function?

(4) How can templates be used for generic programming?

(5) Explain need of templates.

(6) What do you mean by overloading of template function?

UNIT - X

Introduction of Java Programming

Structure of the Unit

10.0 Objective

After going through this unit you will be able to:

- Describe importance and features of java, java tokens and keywords.
- Describe java applets and applications, variables, operators and control structure
- Describe java comments, loops and command line arguments.
- Describe class, object constructor and garbage collection etc.

10.1 Introduction

Java is an object-oriented programming language developed at Sun Microsystems in 1991 as part of a research project to develop software for consumer electronics devices-television sets, VCRs, toasters, and the other sorts of machines.

The most important feature of the language is that it is a platform-neutral (independent) language. Java is the first programming language that it is not tied to any particular hardware or operating system. Programs developed in Java can be executed anywhere on any system.

Java programming is most special in relation to other programming languages i.e. it lets you write special program called *applets* that can be downloaded from the Internet and played safely within a web browser.

10.1.1 Key Features of Java

Java has following features :

1. Simplicity
2. Object-oriented
3. Distributed
4. Interpreted
5. Robust
6. Secure
7. Architecturally Neutral
8. Portable
9. High-Performance
10. Dynamic

1. Simplicity

Java is developed by taking the best points from C and C++, means that if one know about c and c++ can easily work on java also. Java utilizes algorithms and methodologies that are already proven. Error prone tasks such as pointers and memory management have either been eliminated or are handled by the Java environment automatically rather than by the programmer. Since Java is primarily a derivative of C++, which most programmers are conversant with, it implies that Java has a familiar feel rendering it easy to use.

2. Object Oriented

Even though Java has the look and feel of C++, it is a fully independent language which has been designed to be object-oriented from the ground up. In object-oriented programming (OOP), data is treated as objects to which methods are applied. Java's basic execution unit is the class. Advantages of OOP include: reusability of code, extensibility and dynamic applications.

3. Distributed

Commonly used Internet protocols such as HTTP and FTP as well as calls for network access are built into Java. Internet programmers can call on the functions through the supplied libraries and be able to access files on the Internet as easily as writing to a local file system.

4. Interpreted Language

When Java code is compiled, the compiler outputs the Java Bytecode which is an executable for the Java Virtual Machine. The Java Virtual Machine does not exist physically but is the specification for a hypothetical processor that can run Java code. The bytecode is then run through a Java interpreter on any given platform that has the interpreter ported to it. The interpreter converts the code to the target hardware and executes it.

5. Robustness

Java compels the programmer to be thorough. It carries out type checking at both compile and runtime making sure that every data structure has been clearly defined and typed. Java manages memory automatically by using an automatic garbage collector. The garbage collector runs as a low priority thread in the background keeping track of all objects and references to those objects in a Java program. When an object has no more references, the garbage collector tags it for removal and removes the object either when there is an immediate need for more memory or when the demand on processor cycles by the program is low.

6. Secure

The Java language has built-in capabilities to ensure that violations of security do not occur.

Even though the Java compiler produces only correct Java code, there is still the possibility of the code being tampered with between compilation and runtime. Java guards against this by using the bytecode verifier to check the bytecode for language compliance when the code first enters the interpreter, before it ever even gets the chance to run.

The bytecode verifier ensures that the code does not do any of the following:

- Forge pointers
- Violate access restrictions
- Incorrectly access classes
- Overflow or underflow operand stack
- Use incorrect parameters of bytecode instructions
- Use illegal data conversions

At runtime, the Java interpreter further ensures that classes loaded do not access the file system except in the manner permitted by the client or the user.

7. Architecturally Neutral

The Java compiler compiles source code to a stage which is intermediate between source and native machine code. This intermediate stage is known as the bytecode, which is neutral. The bytecode conforms to the specification of a hypothetical machine called the Java Virtual Machine and can be efficiently converted into native code for a particular processor.

8. Portable

By porting an interpreter for the Java Virtual Machine to any computer hardware/operating system, one is assured that all code compiled for it will run on that system. This forms the basis for Java's portability.

Another feature which Java employs in order to guarantee portability is by creating a single standard for data sizes irrespective of processor or operating system platforms.

9. High Performance

The Java language supports many high-performance features such as multithreading, just-in-time compiling, and native code usage.

Java has employed multithreading to help overcome the performance problems suffered by interpreted code as compared to native code. Since an executing program hardly ever uses CPU cycles 100 % of the time, Java uses the idle time to perform the necessary garbage cleanup and general system maintenance that renders traditional interpreters slow in executing applications. [NB: Multithreading is the ability of an application to execute more than one task (thread) at the same time e.g. a word processor can be carrying out spell check in one document and printing a second document at the same time.]

Since the bytecode produced by the Java compiler from the corresponding source code is very close to machine code, it can be interpreted very efficiently on any platform. In cases where even greater performance is necessary than the interpreter can provide, just-in-time compilation can be employed whereby the code is compiled at run-time to native code before execution.

An alternative to just-in-time compilation is to link in native C code. This yields even greater performance but is more burdensome on the programmer and reduces the portability of the code.

10. Dynamic

By connecting to the Internet, a user immediately has access to thousands of programs and other computers. During the execution of a program, Java can dynamically load classes that it requires either from the local hard drive, from another computer on the local area network or from a computer somewhere on the Internet.

Java provides a rich set of operators for manipulating program data. Most of these were taken directly from C/C++, but you must be careful because there are some significant differences to keep in mind. In this tutorial we will cover the most commonly used operators.

10.2 Java Applet and Application

According to Sun "An applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application....The Applet class provides a standard interface between applets and their environment."

Four definitions of applet:

A small application

A secure program that runs inside a web browser

A subclass of java.applet.Applet

An instance of a subclass of java.applet.Applet

10.2.1 Applet Elements

Applets are embedded in web pages using the <APPLET> and </APPLET> tags. The APPLET element is similar to the IMG element. Like IMG APPLET references a source file that is not part of the HTML page on which it is embedded. IMG elements do this with the SRC attribute. APPLET elements do this with the CODE attribute. The CODE attribute tells the browser where to look for the compiled .class file. It is relative to the location of the source document.

Thus if you're browsing http://www.naukri.com/index.html and that page references an applet with CODE="Animation", then the file Animation.class should be at the URL http://www.ibiblio.org/Animation.class.

If the applet resides somewhere other than the same directory as the page it lives on, you don't just give a URL to its location. Rather you point at the CODEBASE. The CODEBASE attribute contains a URL that points at the directory where the .class file is. The CODE attribute is the name of the .class file itself. For instance if on the HTML page of the previous section you had written

<applet code="HelloWorldApplet" codebase="classes"

width="200" height="200">

</applet>

then the browser would have tried to find HelloWorldApplet.class in the classes directory in the same directory as the HTML page that included the applet. On the other hand if you had written

<applet code="HelloWorldApplet"

codebase="http://www.foo.bar.com/classes" width="200" height="200">

</applet>

then the browser would try to retrieve the applet from http://www.foo.bar.com/classes/HelloWorldApplet.class regardless of where the HTML page was.

In short the applet viewer will try to retrieve the applet from the URL given by the formula (CODEBASE + "/" + code). Once this URL is formed all the usual rules about relative and absolute URLs apply.

If the applet is in a non-default package, then the full package qualified name must be used. For example,

<applet code="com.macfaq.greeting.HelloWorldApplet"

codebase="http://www.example.com/classes" width="200" height="200">

</applet>

In this case the browser will look for

http://www.example.com/classes/com/macfaq/greeting/HelloWorldApplet.class so the directory structure on the server should also mirror the package hierarchy.

The HEIGHT and WIDTH attributes work exactly as they do with IMG, specifying how big a rectangle the browser should set aside for the applet. These numbers are specified in pixels and are required.

10.2.2 Applet Life cycle

1. The browser reads the HTML page and finds any <APPLET> tags.
2. The browser parses the <APPLET> tag to find the CODE and possibly CODEBASE attribute.
3. The browser downloads the .class file for the applet from the URL found in the last step.
4. The browser converts the raw bytes downloaded into a Java class, that is a java.lang.Class object.
5. The browser instantiates the applet class to form an applet object. This requires the applet to have a noargs constructor.

6. The browser calls the applet's init() method.
7. The browser calls the applet's start() method.
8. While the applet is running, the browser passes any events intended for the applet, e.g. mouse clicks, key presses, etc., to the applet's handleEvent() method. Update events are used to tell the applet that it needs to repaint itself.
9. The browser calls the applet's stop() method.
10. The browser calls the applet's destroy() method.

All applets have the following four methods:

init(), start(), stop(), and destroy()

The init() method is called exactly once in an applet's life, when the applet is first loaded. It's normally used to read PARAM tags, start downloading any other images or media files you need, and set up the user interface. Most applets have init() methods.

The start() method is called at least once in an applet's life, when the applet is started or re-started. In some cases it may be called more than once. Many applets you write will not have explicit start() methods and will merely inherit one from their superclass. A start() method is often used to start any threads the applet will need while it runs.

The stop() method is called at least once in an applet's life, when the browser leaves the page in which the applet is embedded. The applet's start() method will be called if at some later point the browser returns to the page containing the applet. In some cases the stop() method may be called multiple times in an applet's life. Many applets you write will not have explicit stop() methods and will merely inherit one from their superclass. Your applet should use the stop() method to pause any running threads. When your applet is stopped, it should not use any CPU cycles.

The destroy() method is called exactly once in an applet's life, just before the browser unloads the applet. This method is generally used to perform any final clean-up. For example, an applet that stores state on the server might send some data back to the server before it's terminated. many applets will not have explicit destroy() methods and just inherit one from their superclass.

For example, in a video applet, the init() method might draw the controls and start loading the video file. The start() method would wait until the file was loaded, and then start playing it. The stop() method would pause the video, but not rewind it. If the start() method were called again, the video would pick up where it left off; it would not start over from the beginning. However, if destroy() were called and then init(), the video would start over from the beginning.

In the JDK's appletviewer, selecting the Restart menu item calls stop() and then start(). Selecting the Reload menu item calls stop(), destroy(), and init(), in that order. (Normally the byte codes will also be reloaded and the HTML file reread though Netscape has a problem with this.)

The applet start() and stop() methods are not related to the similarly named methods in the java.lang.Thread class.

Your own code may occasionally invoke start() and stop(). For example, it's customary to stop playing an animation when the user clicks the mouse in the applet and restart it when they click the mouse again.

Your own code can also invoke init() and destroy(), but this is normally a bad idea. Only the

environment should call init() and destroy().

10.3 Java Tokens and Keywords

The java programming language has a term for its building blocks, or basic elements: tokens. Becoming familiar with these words and concepts is a prerequisite to programming in java. At compile time, the javac compiler takes the code and pulls out specific information, called a token, for further processing.

First, javac takes out escape sequences from the raw byte codes. It determines if the escape sequences are line terminators or input characters. Next, javac takes out white space that is not internal to a string, including the ASCII space character, carriage return, line feed, and horizontal tab. Then the compiler gets down to the business of pulling out tokens.

There are several types of tokens: identifiers, keywords, literals, operators, separators, and comments.

Identifiers

The javac compiler needs to know the names of items in the program. For example, it must recognize the names of variables, methods, and component elements of classes. Identifiers are either Java reserved words or titles given to variables, classes, and methods. Reserved words are names that may be used only by Java. Using these words in any other way will cause a compile error.

Identifier Name Size

Java identifiers are case sensitive. For example, char is a reserved word in Java. This means that it is possible to assign a variable the identifier Char, chHR, or etc

Keywords

Keywords are reserved words, which means that they cannot be used in any way other than how Java intends for them to be used. Keywords are, therefore, special tokens. They are always lowercase.

Java keywords are used as application flow controls, declarations, class identifiers, and expressions. Table lists all the reserved keywords in Java. Keywords that pertain to the fundamentals of Java are explained in the appropriate section of this chapter.

Data Declaration Keywords							
boolean	byte	char	double	float	int	long	short
Loop Keywords							
break	continue	do	for	While			
Conditional Keywords							
case	else	if	switch				
Exception Keywords							
catch	finally	throw	try				
Structure Keywords							
abstract	class	default	extends	implements	instanceof	interface	
Modifier and Access Keywords							
final	native	new	private	protected	public	static	synchronized
thread	transient	void					
Miscellaneous Keywords							
false	import	null	package	return	super	this	true

Table 10.1 eserved keywords in Java.

Separators

Java uses the following separators: (), {}, [], ;, ,, and . . The compiler uses these *separators* to divide the code into segments. They can also force arithmetic precedence evaluation within an expression. Separators are also useful as visual and logical locators for programmers.

10.4 Java Comments

Comments have an initial indicator (usually text) and an end indicator. Sometimes comments are used to separate logic, in which case there may be no text. There are three ways to indicate comments in Java.

Comment indicators.

Start	***Text***	***End Comment***
`/*`	`text`	`*/`
`/**`	`text`	`*/`
`//`	`text`	(everything to the end of the line is ignored by the Compiler)

The /* comment is familiar to C programmers. The /** comment is typically used for machine-generated comments. The third style is familiar to C++ programmers.

The following are examples of comments:

```
/* this is an example of a comment */
/** this is another example of a comment. I can
   go to multiple lines and the compiler will
   look at this as a comment until I do the proper
   end comment as in */
//  with this method I can't go to multiple lines unless
// I start each line with the comment marker.
```

Comments can be used anywhere in code without affecting the code's execution or logic.

10.5 Java Data Types and Variables

A *variable* is something that changes the values . In Java, a variable stores data. Data types define the kind of data that can be stored in a variable and the limits of the data.

An example of the use of a data type is

```
char ch;
```

This example demonstrates the two essential parts of a variable: the type (char) and the identifier (ch). The type notifies the compiler that the variable ch will be of type char.

Another example of defining data types follows:

```
int no, b, n;
```

we can enter multiple variables of the same type on the same line when separated by a comma. The default value for integers is 0.

There are two major data types in Java: reference types and primitive types.

Data types can be stored in variables, passed as arguments, returned as values, and operated on.

Primitive Data Types

Primitive data types can have only one value at a time. They do not reference other data or indicate sequence in a group of data. They are primitive in that they are the simplest built-in forms of data in Java. All other data types are made up of combinations of primitive data types. The primitive data type keywords are shown in Table.

Primitive data type keywords.							
Boolean	char	Byte	short	int	long	double	float

Integer Data Types

There are four integer data types: byte, short, int, and long. Each can handle different ranges of numbers, as summarized in Table 10.2

Type	***Length***	***Minimum Value***	***Maximum Value***
byte	8 bits	-128	127
short	16 bits	-32768	32767
int	32 bits	-2147483648	2147483647
long	64 bits	-9223372036854775808	9223372036854775807

Table 10.2 :Integer data type ranges.

char Data Types

Type char is really a 16-bit unsigned integer that represents a Unicode value. In other words, it is possible to determine Unicode characters and escape codes by using a numeric representation. each printable and nonprintable character has a Unicode value. Because Java stores all characters as Unicode, it can be used with virtually any written language in the world.

Floating-Point Data Types

Type float designates that the variable is a single-precision, 32-bit, floating-point number. Type double is a double-precision, 64-bit, floating-point number. Here are examples of declaring floating-point variables:

float f;

double d;

boolean Data Types

Type boolean can only have a value of true or false. Internal to Java, a boolean value is a 1-bit logical quantity.

Reference Data Types

There are situations in which variables must be logically grouped together for manipulation, possibly because they are going to be accessed in sequence or identifiers must point to dynamically

allocated objects. These are called *reference data types.*

10.6 Literals And Array

Data is represented by *literals* in Java. Similar to literals in other programming languages, Java literals are based on character and number representations. The types of literals are integer, floating-point, boolean, character, and string.

Every variable consists of a literal and a data type. The difference between the two is that literals are entered explicitly into the code. Data types are information about the literals, such as how much space will be reserved in memory for that variable, as well as the possible value ranges for the variable

Integer Literals

Integers are whole numbers, such as 1 and 5280. *Integer literals* can be decimal (base 10), octal (base 8), or hexadecimal (base 16).

Decimals can be positive, zero, or negative. Decimal literals cannot start with 0, as in 01234. After the beginning number, a decimal literal can consist of the numbers 0-9.

Floating-Point Literals

A *floating-point literal* represents a number that has a decimal point in it, such as 3.7. Single-precision floating-point numbers consist of a 32-bit space and are designated by uppercase or lowercase *f*.

Boolean Literals

A Boolean literal is either of the words *true* or *false*. Unlike other programming languages, no numeric value such as 0 or 1 is assigned. Therefore, the value of a Boolean literal is, literally, true or false. Booleans are used extensively in program control flow logic.

Character Literals

Programmers often use a single character as a value. In Java, this is represented by *character literals*. The value of a character literal is enclosed by single quotes. An example is 'y'.

Assigning a value to a character literal gets more interesting if the value must be a single quote, a backslash, or other nonprintable characters. A backslash (\) is used to designate certain nonprintable characters or characters that are properly part of the command. For example, assign the value '\" to characterize the single quote. Table shows some examples of assigning values to character literals.

Description or Escape Sequence	***Sequence***	***Output***
Any character	'y'	y
Backspace (BS)	'\b'	Backspace
Horizontal tab (HT)	'\t'	Tab
Linefeed (LF)	'\n'	Linefeed
Formfeed (FF)	'\f'	Form feed
Carriage return (CR)	'\r'	Carriage return
Double quote	'\"'	"
Single quote	'\''	'
Backslash	'\\'	\
Octal bit pattern	'\ddd'	Octal value of ddd
Hex bit pattern	'\xdd'	Hex value of dd
Unicode character	'\udddd'	Actual Unicode character of dddd

Table 10.3 :Specifying character literals

Compile-time errors occur if anything other than a single quote follows the value designator or the character after '\' is anything but b, t, n, f, r, ", ', \, 0, 1, 2, 4, 5, 6, or 7.

String Literals

String literals are a sequence of characters enclosed in double quotes, such as "hello how are you". This could also be "Hello World" or even "" for a null character string.

Arrays

Arrays are collection of variables that can single- or multi dimensional groups of variables. Elements can be of a primitive type, as in float, char, or int. Elements can also be a class or interface type. Arrays can consist of other arrays.

we can declare an array without allocating it. In other words, the variable itself is created, but no space is allocated in memory for array objects until the array is initialized or values are assigned to the elements of the array. Arrays are generally initialized with the new command, which creates a new instance of a reference data type.

```
class Array {
  public static void main (String args[])
  {
    int LSIZE = 5;
    String[] SList;
    int i = LSIZE;
    // create array
    SList = new String[LSIZE];
    // initialize array
    SList[0] = "AAA";
    SList[1] = "BBB";
    SList[2] = "CCC";
    SList[3] = "DDD";
    SList[4] = "EEE";
    for (i=0; i < LSIZE; i++)
{
      System.out.println(SList[i]);
}  } }
```

More on Arrays

There is only one way to reference the items: by using a subscript to indicate which element number to access. The number used to reference the specific element of an array is called the *component.* This concept is represented in Java as

```
char SArray[] = new char[5];
```

In this example, SArray is the name of the array. The [5] declares that there are five elements in the array. All elements are of type char. The elements are referenced by SArray[0] to SArray[4].

In Java, the subscripts always start at zero and go to the length of the array minus 1.

All the elements of an array must be of the same type.

Two-dimensional arrays need two reference points because the data is stored in a grid of rows and columns.. An element is referenced by the row and column number of its location, as in [3][4]. This means that the current element being manipulated is in row 4, column 5 of an array.

10.7 Operators and their Precedence

Operators are the symbols used for arithmetic and logical operations. Arithmetic symbols define operations that apply to numbers (for example, 2 + 3). Operators, except the plus sign (+), are used only for arithmetic calculations. The + operator can be used with strings as well, as shown in the previous example of string literal concatenation. Tables list all the Java operators.

Operator	*Operation*	*Example*
+	Addition	g + h
-	Subtraction	g - h
*	Multiplication	g * h
/	Division	g / h
%	Modulus	g % h

Table 10.4 : arithmetic operators.

Operator	*Operation*	*Example*	*Meaning*
=	Assign value	a = 7	a = 7
+=	Add to current variable	g += h	g = g + h
-=	Subtract from current	g -= h	g = g - h
*=	Multiply current	g *= h	g = g * h
/=	Divide current	g /= h	g = g / h
%=	Modulus current	g %= h	g = g % h

Table 10.5 : assignment operators

Operator	*Operation*	*Example*	*Meaning*
++	Increment by 1	g++ or ++g	g = g + 1
--	Decrement by 1	g-- or --g	g = g -1

Table 10.6: increment and decrement operators.

Operator	*Operation*	*Example*	*Meaning*
==	Equal	g == h	Is g equal to h?
!=	Not equal	g != h	Is g not equal to h?
<	Less than	g < h	Is g less than h?
>	Greater than	g > h	Is g greater than h?
<=	Less than or equal	g <= h	Is g less than or equal to h?
>=	Greater than or equal	g >= h	Is g greater than or equal to h?

Table 10.7: relational operators (which return true or false)

Operator	*Operation*
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
<<	Left shift
>>	Right shift
>>>	Zero fill right shift
~	Bitwise complement
<<=	Left shift assignment
>>=	Right shift assignment
>>>=	Zero fill right shift assignment
x&=y	AND assignment
x|=y	OR assignment
x^=y	NOT assignment

Table 10.8: bitwise operators.

Expressions

The main reasons that code is created are to manipulate, display, and store data. *Expressions,* or formulas, are Java's way of performing a computation. Operators are used in expressions to do the work of the program and to operate on variables.

Variables must be declared before they are used. An error will occur if you try to operate on an undeclared variable.

The expression book = 4 uses the operator = to assign the value 4 to variable book.

Operator	*Operation*
-	Unary negation
~	Bitwise complement
++	Increment
--	Decrement
!	Not

Table 10.9: Unary operators

Increment and decrement operators are operators that actually change the value of the original variable.

Assignment Operations

An assignment operator stores a specific value in the memory area allocated to a variable. For example, in the expression g += 5;, 5 is added to the original value of variable g, and the old value is replaced. Here are more examples:

g = 5; // assigns to g the value 5

g -=7; // assigns to g the result of g - 7

g *=8; // assigns to g the result of g * 8

g /=4; // assigns to g the result of g / 4

g %=3; // assigns to g the result of g % 3

Binary Operations

Unary operators act on only one variable, and *binary operators* act on two and return one value. Only the variable that receives the value as a result of the expression is changed.

Highest				**Lowest**
[]	()			
--	!	~	instanceof	
new (type) expression				
*	/	%		
+	-			
<<	>>	>>>		
<	>	<=	>=	
==	!=			
&				
^				
&&				
||				
?:				
=	op=			

Table 10.10 : precedence of Operator (from high to low)

The separators [] and () change precedence. Everything within these separators will be computed before Java looks outside them.

Declarations

The *declaration statement* defines the type of variable. This section gives more code examples of declaration use.

Declarations can happen anywhere in sections of code, although it is best for readability to group them together in the beginning of a code section.

Blocks are sections of code beginning and ending with curly braces ({}). Think of a block as a logical unit of code that can call methods, perform expressions, display output, and so on. A variable declared in a block is valid for that block and all its sub-blocks; this is the *scope* of the identifier. However, the scope of a variable does not move outward to the enclosing block; that is, a variable that is declared in a block has no meaning in the area outside its curly braces.

Here is an example of *hiding* a variable by using a second declared variable of the same name:

```
class ShowHiding {
  public static void main (String args[])
  {
  int TableTop;
  TableTop = 2;
  ...
  switch  (AnyCommand) {
    case '1':
      int TableTop = 4;    // TableTop has just been declared again
                           // and assigned a value of 4
      break;
      ...
  }
  ....  /*return to the main loop
  System.out.println(TableTop);    /*TableTop still has a value of 2*/
}
```

In this situation, the second time variable TableTop has been declared as valid for the case command block only. Any values assigned to it are good only for the scope of that block. When program execution returns to the main block, any values assigned to the new variable are lost. The upshot is that there must be a compelling reason to use a variable name more than once in a program. And be careful not to declare two variables with the same name accidentally!

The previous example also shows the combination of a declaration and an assignment statement:

```
int TableTop = 4;
```

This declares a variable of type int, with identifier name TableTop and a value of 4.

Declaring integer Types

Variables of type integer are declared by the amount of memory space they will be allotted. (This is discussed more fully in the "Integer Data Types" section.) Following are examples of integer declarations:

```
byte ByteVar;      //8 bits
short ShortVar;    //16 bits
int IntVar;        //32 bits
long LongVar;      //64 bits
```

Declaring Floating-Point Types

Floating-point variables are 32 or 64 bits in length. Following are examples of floating-point declarations:

```
float FloatVar;      //32 bits
double DoubleVar;    //64 bits
```

Declaring character Types

A character type variable holds only one character. This is different from the Strings class, which contains groups of characters. Remember, the char type holds an integer that references the Unicode character. The following sample code declares two character types:

```
char MyChar;          // holds one character
char MyChar = 'y';    // declares variable MyChar and assigns y to it
```

Declaring Arrays

Arrays are covered in depth in the "Data Types" and "Expressions" sections of this chapter. Arrays are one-dimensional lists of objects that are referenced by component numbers or subscripts. They can consist of other arrays, resulting in classic multidimensional arrays.

Arrays are declared as follows:

```
char MyCharArray[];       //one-dimensional array
char AnotherArray[][];    //two-dimensional array
int IntegerArray[];       //one-dimensional array of integers
int []IntegerArray;       //equivalent to Integer Array[]
```

10.8 Control Flow Statements

Control flow instructs the program how to make a decision and how further processing should proceed on the basis of that decision. Control flow gets your computer to start doing some of the thinking for us.

The building blocks of control flow are the { and } block delimiter characters and the if, while, do, for, and switch keywords. Each of these can be used to control how our program executes by determining if a condition is true and then executing a different section of code, based on the result. This is called a *conditional expression*.

10.8.1 Blocks and Statements

A *statement* is any line of code ending in a semicolon. A statement can be an expression, a

method call, or a declaration. A *block* is a group of statements that form a single compound statement. Think of blocks as statements logically grouped together for program flow and readability.

How do you tell Java where a block starts and ends? The characters { and } group such sections together. For example, the following is considered a block of code:

```
{
  Store   = "Grocery";
  Item[0] = "spinach";
  Item[1] = "tofu";
  Item[3] = "rice";
}
```

When the program runs and gets to this block of code, it will begin execution at the beginning { and will not continue execution elsewhere until leaving the final }. The opening {, the closing }, and all code in between is considered a *block*. The curly braces must be paired one with one another or Java will not know where a block begins and ends.

10.8.2 Conditional Expressions

Conditional expressions will generally execute one of several sections of code on the basis of a conditional test. This code can be as simple as a single statement, but more complex sections of code will be made up of many statements. Conditional expressions are used to make decisions in a program. They are used to evaluate whether a condition is true or false and will branch to different sections of code on the basis of the answer.

If statement

The simplest, but most important, conditional expression is the if statement. An if statement makes up a conditional expression of the form

```
if (expression) statement;
```

or

```
if (expression)
{
  statement(s);
}
```

If the expression in the parentheses evaluates to the boolean true, *statement* is executed. If the expression evaluates to false, *statement* is skipped and execution continues at the statement following the if statement. The following code fragment shows how this works:

```
int Number;                          // declare variable
Number = System.io.read();           // get character from keyboard
if ( (Number % 2) == 0 )
  System.out.println("even");        // test if number is even and
                                     // print "even" if it is
```

Note that System.io.read will retrieve only one character at a time, so this program actually will handle only single-digit numbers. Only the first character will be tested if you enter a multiple-

digit number.

In this example, the program reads a number from the keyboard. It is then tested to determine if it is even. If it is, the System.out.println statement is executed and the program terminates or continues to the next statement block.

Here is how the previous example can be extended:

```
int Number;                          // declare variable
Number =System.io.read();            // get number from keyboard
if ( (Number % 2) == 0 )             // test if number is even
{                                    // begin block
  System.out.println("even");        // print message to screen
}                                    // end block
```

There is an optional companion to the if statement that can extend its usefulness: the else statement. The statement following the if expression is executed only if the expression evaluates to true. The statement following the else is executed only if the if expression evaluates to false:

```
int Number;                          // declare variable
Number = System.io.read();           // get character from keyboard
if ( (Number % 2) == 0 )             // test if number is even
  {                                  // begin if block
    System.out.println("even");      // print message to screen
  }                                  // end if block
else                                  // else if number not even
{                                    // begin else block
  System.out.println("odd");         // print message to screen
}                                    // end else block
```

You can also nest if statements within each other if you need a more complex multiway branch:

```
char KeyboardChar;                   // declare variables
int Number;
System.out.print("Enter number> ");     // print prompt to screen
Number = System.in.read();           // get character from keyboard
if ( (Number % 2) == 0 )             // test if number is even
{   // begin if block
  if ( Number < 5 )
  {   // begin nested if block
    System.out.println("even & < 5");   // print message to screen
  }   // end nested if block
  else   // else if number not < 0
  {   // begin nested if block
    System.out.println("even & >= 5");  // print message to screen
```

```
  }    // end nested if block
}    // end if block
else    // else if number not even
{    // begin else block
  if ( Number < 5 )
  {    // begin nested if block
    System.out.println("odd & < 5");    // print message to screen
  }    // end nested if block
  else    // else if number not < 0
  {    // begin nested if block
    System.out.println("odd & >= 5");    // print message to screen
  }    // end nested if block
}    // end else block
```

if statements are powerful and are the underpinnings for much of programming. The if statement is a fundamental control structure, because almost anything can be tested with one.

switch

A variation on the if statement is the switch statement, which performs a multiway branch instead of a simple binary branch. switch statements are of the form

```
switch (expression)
{
  case value:
    statement(s);
    break;
  case value:
    statement(s);
    break;
      .
      .
      .
  default:
    statement(s);
    break;
}
```

The keyword switch begins the switch construct. The parentheses contain values that must evaluate to a byte, a char, a short, or an int. Next is a required {, followed by any number of constructs that begin with the keyword case and end with break, with any number of statements between. Next there is an optional section that begins with the keyword default and ends with break. Finally, a required } completes the case statement. This sounds complicated in description, but is obvious in usage, as demonstrated in the next example.

The case keywords are each followed by a *value*. This must also be of type byte, char, short, or int. The case statement itself works by evaluating the expression and then scanning down the case statements until an exact match is found. At this point the corresponding group of statements between the case and break will be executed. When the break is encountered, execution will resume at the first statement following the switch construct. If no matches are found and a default is present, the group of statements associated with default will be executed. If no default is present, execution will fall through the entire switch construct and do nothing, with execution again continuing after the end of the construct.

A case statement does not have to have a break associated with it. If break is not present, program execution falls through to the next case statement. It keeps executing that group of statements until a break is encountered, so be sure to place appropriate breaks if this is not the intended action.

Here is an example:

```
static void ParseChar (char KeyboardChar)
{
switch (KeyboardChar) {
    case 'l':
      System.out.println("left");
       break;
    case 'r':
      System.out.println("right");
       break;
    case 'q':      //note no break here, falls through
    case '\n':
       break;
    case 'h':      //note no break here either
    default
      System.out.println("Syntax: (l)eft, (r)ight, (q)uit");
      System.out.println("Please enter a valid character");
       KeyboardChar = ' ';
       break;
  }
}
```

In this example *expression* is a char type, and each case contains a corresponding char value.

case statements cannot evaluate strings or objects, as is true with C and C++. Only items whose values can be evaluated as integers can be used in case statements.

10.9 Looping Expressions

Looping expressions generally continue to loop through a section of code until a certain condi-

tion is met. Some looping expressions check the condition before executing the code. Other looping expressions check the condition after executing the code.

10.9.1 while

The while loop is a construct that repeatedly executes a block of code as long as a boolean condition remains true. The initial while expression is evaluated first. It is possible for the statements making up a while loop never to execute if the initial expression evaluates to false. while loops are of the form

```
while ( expression ) statement;
    or
while ( expression )
{
  statement(s);
}
```

The keyword while begins the while construct. The parentheses contain an expression, which must evaluate to a boolean. This is followed by a statement or a block.

When a while loop is encountered, the expression is evaluated first. If it evaluates to true, the statement or block following the while statement, known as the *body* of the while loop, is executed. When the end of the body is reached, the expression is evaluated again. If it is false, execution will continue with the next statement following the while loop. If it is true, the body of the while loop will be executed again. The body will continue to be executed until the expression evaluates to false.

Here is an example:

```
char KeyboardChar =(char)System.in.read();
while (KeyboardChar != 'q')
{
  ProcessChar(KeyboardChar);
  KeyBoardChar = (char)System.in.read();
}
```

This expression tests whether KeyboardChar is equal to the character q. If it is not, the body of the while loop will be executed, which will process the character and then read in another. This will continue until the character q is read in. Notice in this example that KeyboardChar has been initialized by reading a character before the loop is executed. If it was not explicitly set, the while loop might never be entered, depending on what value KeyboardChar had. This is another common programming error. Even if KeyboardChar were initialized to something, it would be processed before the user had a chance to type in a character.

A while loop continues to loop until the expression evaluates to false. It is possible for a while loop to execute forever if the expression never evaluates to false; this is known as an *infinite loop*. A common cause of infinite loops is that the programmer forgot to put a statement that changes part of the expression in the body of the loop. If the expression never changes, it will always evaluate the same and an infinite loop occurs.

10.9.2 do-while

The do loop enables your code to repeatedly execute a block of code until a boolean expression evaluates to false. It is almost identical to the while loop, except the expression is evaluated at the bottom of the loop rather than the top. This means that the contents of the loop will always be executed at least once. do loops are of the form

do *statement*; while (*expression*);

or

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

The keyword do begins the do construct. This is followed by a statement or a block. Next is the keyword while, followed by the parentheses containing an expression that must evaluate to a boolean.

When a do loop is encountered, the statement or block following the do keyword is executed. When the do loop body completes, expression is evaluated. If it is false, execution will continue with the next statement following the do loop. If it is true, the body of the do loop will be executed again. The body will continue to be executed until the expression evaluates to false.

Here is an example:

```
do
{
  KeyBoardChar = (char)System.in.read();
  ProcessChar(KeyboardChar);
} while (KeyboardChar != 'q')
```

In this example, the body of the while loop is executed, which reads in a character and then processes it. The expression is then evaluated to determine if KeyboardChar is equal to the character q. If it is, execution will continue with the first statement after the do loop. If it is not, the do loop body will be executed again. This will continue until the character q is read in.

Compare this version with the while loop version shown previously. It does not need an initialization of KeyboardChar because the variable will be read in at the beginning of the loop. This is the most common reason for choosing a do over a while loop.

Like with the while loop, it is possible to create an infinite loop by forgetting to put in the body of the loop a statement that changes part of the expression.

10.9.3 for

The for loop enables code to execute repeatedly until a boolean expression evaluates to false. It is similar to a while loop but is more specialized. As in a while loop, the expression is evaluated at the top of the loop. However, it provides a more explicit means for initializing a loop variable and modifying it at the end of the loop. for loops are of the form

for (*initialization*; *expression*; *modification*) *statement*;

or

```
for (initialization; expression; modification)
{
```

```
  statement(s);
}
```

The keyword for begins the for construct. The parentheses contain an *initialization*, an *expression*, and a *modification*. The *initialization* can be a statement of any kind, but typically its purpose is to initialize part of the expression. Initialization is followed by a semicolon (;), followed by an expression. Like the while and do loops, the expression must evaluate to a boolean. This is followed by another semicolon and then *modification*. *modification* also can be any statement, but again is typically used to modify part of the expression. Finally, this is followed by a statement or a block.

When a for loop is encountered, *initialization* is first executed, and then the expression. If it evaluates to true, the statement or block following the while statement is executed. This statement or block is known as the *body* of the for loop. When the end of the body is reached, *modification* is executed. The expression is then evaluated again. If it is false, execution continues with the next statement following the for loop. If it is true, the body of the for loop is executed again. The body continues to be executed until the expression evaluates to false.

Here is an example:

```
for (char KeyboardChar=ProcessChar(KeyboardChar); KeyboardChar != 'q';
KeyBoardChar=(char)System.in.read())
{
  ProcessChar(KeyboardChar);
}
```

In this example, KeyboardChar is initialized by reading in a character. expression is then evaluated to determine if KeyboardChar is equal to the character q. If so, execution will continue with the first statement after the for loop. If not, the body of the for loop will be executed, which will process KeyboardChar. Next, another character will be read in. This will continue until the character q is read in.

You can use for loops to move through a range of numbers. This is used in conjunction with arrays or other indexes. Here's an example:

```
int Array;
for (I=0; I < ArraySize; I++)
{
  if (Array[i] < 0)
  {
System.out.println("ERROR: negative number encountered, index = " + I);
  }
  else
  {
    ProcessArray(Array[i]);
  }
}
```

This for loop will initialize I to a value of zero and then step through Array, checking for negative numbers before processing an entry. This is a compact, easily assimilated method of writing code.

Like with the while and do loops, it is possible to create an infinite loop by forgetting to put a statement that changes part of the expression in the body of the loop.

10.9.4 break

The break construct can be used to break out of the middle of a for, do, or while loop. (This chapter has already discussed how to use it to break out of a switch statement.) When a break statement is encountered, execution of the current loop immediately stops and resumes at the first statement following the current loop.

Here is how the previous for loop example could be extended:

```
int ix;
for (ix=0; ix < ArraySize; ix++)
{
  if (Array[ix] < 0)
  {
    System.out.println("ERROR: negative number encountered, index = " + ix);
    break;
  }
  ProcessArray(Array[ix]);
}
```

Again, this code will loop through Array looking for negative entries. However, by including the break statement, execution of this for loop will stop at the first negative entry. In this example, a negative entry might be considered so severe that no other processing should be done. Also notice that no else statement is needed because if an error occurs, execution will jump to the end of the loop, skipping the entry-processing code.

10.9.5 continue

The continue construct can be used to short-circuit parts of a for, do, or while loop. When a continue statement is encountered, execution of the current loop immediately resumes at the top, skipping all other code between it and the end of the loop.

The following for loop uses the continue construct:

```
int ix;
for (ix=0; ix < ArraySize; ix++)
{
 if (Array[ix] < 0)
  {
    System.out.println("ERROR: negative number encountered, index = " + ix);
    continue;
  }
```

```
  ProcessArray(Array[ix]);
}
```

Again, this code will loop through Array looking for negative entries. However, the inclusion of the continue statement means that execution of this for loop will not continue in the body of the loop if a negative entry is encountered. In this example, a negative entry might be considered illegal and should not be processed. However, it is not so severe that it stops processing other entries. Again, notice that no else statement is needed because if an error occurs, execution will continue at the top of the loop, skipping the entry-processing code.

10.9.6 labeled Loops

If break and continue only take you to the end or beginning of the current loop, what do you do if you have nested loops and need to get out of more than just the current one? Java provides an extended version of break and continue for just this purpose. By adding a label to a loop and referencing it in a break or continue statement, you can make execution continue at the end or beginning of the loop of your choice.

Here's an example:

```
err:
for (ix=0; ix < ArraySize; ix++)
{
  for (j=0; j < ArraySize; j++)
  {
    if (Array[ix][j] < 0)
    {
      System.out.println("ERROR: negative number encountered, index = " + ix + "," + j);
       break err;
    }
    ProcessArray(Array[ix][j]);
  }
}
```

In this example, Array is extended to two dimensions, and two for loops are used to step through all elements of the array. If a negative entry is encountered, execution will branch to the end of the for loop labeled err, rather than the normal inner one. Without this construct, you would need to set an additional flag variable and test it in the outer loop. The label itself must immediately precede the intended loop; if it is placed anywhere else in the code, a compile-time error will occur.

The capability to jump out of the middle of a loop is handled in C++ and some C implementations with a goto statement. The goto can branch anywhere in the code, which can lead to what is known as *spaghetti code*. Of course, spaghetti code is something we recognize in other people's code, but never our own! The labeled loop concept in Java provides breakout capability while limiting the scope. It is a good compromise.

Self Learning Exercises

1. What are the eight primitive data types supported by the Java programming language?
2. Which operator is used to compare two values, = or == ?
3. Consider the following code snippet.

```
int i = 10;
int n = i++%5;
```

a. What are the values of i and n after the code is executed?

b. What are the final values of i and n if instead of using the postfix increment operator (i++), you use the prefix version (++i))?

4. Consider the following code snippet.

```
if (aNumber >= 0)
if (aNumber == 0)
System.out.println("first string");
else System.out.println("second string");
System.out.println("third string");
```

What output do you think the code will produce if aNumber is 3 ?

10.10 Class Fundamental

Classes are the major building block of an object-oriented structure. Classes are what make objects possible, and without objects, object-oriented programming would not make sense. There are several major advantages to using objects. They enable you to encapsulate data, keeping all information and actions about a particular item separate from the rest of your code. They allow you to build class hierarchies, which enable you to build up more and more complex structures from simpler ones.

Classes are the essential building block in any Java applet or application. Classes are used to create objects. When we create an instance of a class, we create an Object. we can include all the code for that object with in the class. In accordance with the object-oriented paradigm, we can later choose to build upon that class to build new programs or enhance our current program.

10.10.1 Declaring a Class and creating Object

```
class Car {
 String licensePlate;       // e.g. "New York 543 A23"
 double speed;  // in kilometers per hour
 double maxSpeed;    // in kilometers per hour
}
```

To instantiate an object in Java, use the keyword new followed by a call to the class's constructor. Here's how you'd create a new Car variable called c:

```
 Car c;
 c = new Car();
```

The first word, Car, declares the type of the variable c. Classes are types and variables of a class type need to be declared just like variables that are ints or doubles.

The equals sign is the assignment operator and new is the construction operator.

Finally notice the Car() method. The parentheses tell you this is a method and not a data type like the Car on the left hand side of the assignment. This is a constructor, a method that creates a new instance of a class. You'll learn more about constructors shortly. However if you do nothing, then the compiler inserts a default constructor that takes no arguments.

This is often condensed into one line like this:

```
Car c = new Car();
```

10.10.2 Methods

Data types aren't much use unless you can do things with them. For this purpose classes have methods. Fields say what a class is. Methods say what a class does. The fields and methods of a class are collectively referred to as the members of the class.

The classes you've encountered up till now have mostly had a single method, main(). However, in general classes can have many different methods that do many different things. For instance the Car class might have a method to make the car go as fast as it can. For example,

```
class Car {
 String licensePlate = "";   // e.g. "New York 543 A23"
 double speed       = 0.0;   // in kilometers per hour
 double maxSpeed    = 123.45; // in kilometers per hour
  // accelerate to maximum speed
 // put the pedal to the metal
 void floorIt() {
  this.speed = this.maxSpeed;
 }
 }
```

The fields are the same as before, but now there's also a method called floorIt(). It begins with the Java keyword void which is the return type of the method. Every method must have a return type which will either be void or some data type like int, byte, float, or String. The return type says what kind of the value will be sent back to the calling method when all calculations inside the method are finished. If the return type is int, for example, you can use the method anywhere you use an int constant. If the return type is void then no value will be returned.

floorIt is the name of this method. The name is followed by two empty parentheses. Any arguments passed to the method would be passed between the parentheses, but this method has no arguments. Finally an opening brace ({) begins the body of the method.

There is one statement inside the method

this. speed = this.maxSpeed;

Notice that within the Car class the field names are prefixed with the keyword this to indicate that I'm referring to fields in the current object.

Finally the floorIt() method is closed with a } and the class is closed with another }.

10.10.3 Invoking Methods

```
class Car {
 String licensePlate = "";    // e.g. "New York 543 A23"
 double speed       = 0.0;    // in kilometers per hour
 double maxSpeed    = 123.45; // in kilometers per hour
  // accelerate to maximum speed
 // put the pedal to the metal
 void floorIt() {
  this.speed = this.maxSpeed;
 }
}
```

Outside the Car class, you call the floorIt() method just like you reference fields, using the name of the object you want to accelerate to maximum and the . separator as demonstrated below

```
class CarTest3 {
 public static void main(String args[]) {
   Car c = new Car();
  c.licensePlate = "New York A45 636";
   c.maxSpeed = 123.45;
  System.out.println(c.licensePlate + " is moving at " + c.speed +
   " kilometers per hour.");
  c.floorIt();
  System.out.println(c.licensePlate + " is moving at " + c.speed +
   " kilometers per hour.");
 }
  }
```

The output is:

New York A45 636 is moving at 0.0 kilometers per hour.

New York A45 636 is moving at 123.45 kilometers per hour.

The floorIt() method is completely enclosed within the Car class. Every method in a Java program must belong to a class. Unlike C++ programs, Java programs cannot have a method hanging around in global space that does everything you forgot to do inside your classes.

10.11 Constructors

A constructor creates a new instance of the class. It initializes all the variables and does any work necessary to prepare the class to be used. In the line

Car c = new Car();

Car() is the constructor. A constructor has the same name as the class.

If no constructor exists Java provides a generic one that takes no arguments (a noargs construc-

tor), but it's better to write your own. You make a constructor by writing a method that has the same name as the class. Thus the Car constructor is called Car().

Constructors do not have return types. They do return an instance of their own class, but this is implicit, not explicit.

The following method is a constructor that initializes license plate to an empty string, speed to zero, and maximum speed to 120.0.

```
Car() {
  this.licensePlate = "";
   this.speed  = 0.0;
  this.maxSpeed = 120.0;
 }
```

Better yet, you can create a constructor that accepts three arguments and use those to initialize the fields as below.

```
Car(String licensePlate, double speed, double maxSpeed) {
  this.licensePlate = licensePlate;
   this.speed  = speed;
  if (maxSpeed > 0) this.maxSpeed = maxSpeed;
  else this.maxSpeed = 0.0;
  if (speed > this.maxSpeed) this.speed = this.maxSpeed;
  if (speed < 0) this.speed = 0.0;
  else this.speed = speed;
 }
```

Or perhaps you always want the initial speed to be zero, but require the maximum speed and license plate to be specified:

```
Car(String licensePlate, double maxSpeed) {
  this.licensePlate = licensePlate;
   this.speed  = 0.0;
  if (maxSpeed > 0) this.maxSpeed = maxSpeed;
  else this.maxSpeed = 0.0;
 }
```

Here's the complete class:

```
class Car {
 String licensePlate; // e.g. "New York A456 324"
 double speed;       // kilometers per hour
 double maxSpeed;    // kilometers per hour
  Car(String licensePlate, double maxSpeed) {
  this.licensePlate = licensePlate;
   this.speed  = 0.0;
```

```
  if (maxSpeed > 0) this.maxSpeed = maxSpeed;
  else this.maxSpeed = 0.0;

 }
 // getter (accessor) methods
 String getLicensePlate() {
  return this.licensePlate;
 }
 double getMaxSpeed() {
  return this.maxSpeed;
 }
 double getSpeed() {
  return this.speed;
 }
 // accelerate to maximum speed
 // put the pedal to the metal
 void floorIt() {
  this.speed = this.maxSpeed;
 }
 void accelerate(double deltaV) {
   this.speed = this.speed + deltaV;
   if (this.speed > this.maxSpeed) {
     this.speed = this.maxSpeed;
     }
   if (this.speed <  0.0) {
      this.speed = 0.0;
     }
   }
}
```

Notice that I've taken out several things:

the initialization of the fields

the setter methods

10.12 Garbage Collection

As we know that which chunks of memory are in use and which are not is a major headache in c++. Many c++ classes use destructors to free up chunks of memory that are no longer needed, but this does not always work. If you delete objects that are still in use, the program will crash. If

you forget to delete objects that are no longer in use, the program will crash when your system runs out of memory. If you point to the wrong area of memory, you can overwrite essential data and crash the program.

Java avoids memory-related problems by automating garbage collection. To do this, the java runtime environment tracks which chunks of memory are in use and which aren't. When a chunk of memory is no longer needed, the system automatically clears it. When a chunk of memory is needed, the system automatically allocates it.

Java provides an easy way to clean up processes gracefully using a finalize method. The finalize method can be used like a C++ destructor to do final cleanup on an object before garbage collection occurs.

10.13 Summary

This chapter covers the most basic parts of the Java programming language. It gives many examples of ways to use tokens, literals, data types, expressions, declarations, and control flow. Together these form the fundamentals of any program you write or application you develop in Java. This chapter is also a good reference for correct syntax. You will be well on your way to developing powerful applications in Java when you combine these fundamentals with the information covered in the next chapter.

10.14 Glossary

applet : A component that typically executes in a Web browser, but can execute in a variety of other applications or devices that support the applet programming model.

bytecode :Machine-independent code generated by the Java compiler and executed by the Java interpreter.

class :In the Java programming language, a type that defines the implementation of a particular kind of object. A class definition defines instance and class variables and methods, as well as specifying the interfaces the class implements and the immediate superclass of the class. If the superclass is not explicitly specified, the superclass will implicitly be Object.

distributed :Running in more than one address space.

10.15 Further Readings

1. Mastering Java2, J2SE1.4 (Paperback) John Zukowski
2. Java: The Complete Reference ,Herbert Schildt Computers - 2005
3. Java 2: The Complete Reference, Herbert Schildt - Computers - 2002
4. Java How to Program, 7/E (Harvey & Paul) Deitel & Associates Publisher: Prentice Hall ,2007

10.16 Answers to self learning exercises

1. byte, short, int, long, float, double, boolean, char
2. The == operator is used for comparison, and = is used for assignment
3. a. i is 11, and n is 0.
   b. i is 11, and n is 1.
4. second string
   third string

   3 is greater than or equal to 0, so execution progresses to the second if statement. The second if statement's test fails because 3 is not equal to 0. Thus, the else clause executes

(since it's attached to the second if statement). Thus, second string is displayed. The final println is completely outside of any if statement, so it always gets executed, and thus third string is always displayed.

10.17 Unit End Questions

1. Explain the importance of JAVA.
2. List all the features of java and explain them.
3. Write short notes on :
   a. Tokens
   b. Identifiers
   c. Keywords
   d. Variables
   e. Literals
   f. Arrays
   g. Operators
4. What do you mean by control structure, explain all control structure available in java.
5. What is class, how it is declared. Explain by example.
6. What is object, how it is created.
7. What is applet? Write steps for creating the applet.
8. What do you mean by applet life cycle,
9. What are the different methods of applet, explain them.
10. What is garbage collection, why it is useful in java.
11. What is constructor? Explain its role.

UNIT - XI

Overloading and inheritance

Structure of the Unit

11.0 Objective

After going through this unit you will be able to :

- Describe Method overloading ,passing object as parameter, constructor etc
- Describe usage of final and static , private, public and protected, inheritance
- Describe sub and super class, method overriding , dynamic method
- Describe dispatch using abstract class, using final to prevent overriding.

11.1 Introduction

Overloading is when the same method or operator can be used on many different types of data. For instance the + sign is used to add integers as well as real numbers. The plus sign behaves differently depending on the type of its arguments. Therefore the plus sign is inherently overloaded.

Methods can be overloaded as well. System.out.println() can print a double, a float, an int, a long, or a String. You don't do anything different depending on the type of number you want the value of. Overloading takes care of it.

Programmer-defined classes can overload methods as well. To do this simply write two methods with the same name but different argument lists. For Example there are three methods for the class car, one that took three arguments and one that took two arguments, and one that took no arguments. You can use all of these in a single class, though here I only use two because there really aren't any good default values for licensePlate and maxSpeed. On the other hand, 0 is a perfectly reasonable default value for speed.

```
public class Car {
 private String licensePlate; // e.g. "New York A456 324"
 private double speed;       // kilometers per hour
 private double maxSpeed;    // kilometers per hour
 // constructors
 public Car(String licensePlate, double maxSpeed) {
  this.licensePlate = licensePlate;
  this.speed  = 0.0;
  if (maxSpeed >= 0.0) {
   this.maxSpeed = maxSpeed;
   }
  else {
    maxSpeed = 0.0;
   }
 }
 public Car(String licensePlate, double speed, double maxSpeed) {
  this.licensePlate = licensePlate;
  if (maxSpeed >= 0.0) {
   this.maxSpeed = maxSpeed;
```

```
    }
    else {
      maxSpeed = 0.0;
    }
    if (speed < 0.0) {
      speed = 0.0;
    }
    if (speed <= maxSpeed) {
      this.speed = speed;
    }
    else {
      this.speed = maxSpeed;
    }
  }
  // other methods...
}
```

Normally a single identifier refers to exactly one method or constructor. When as above, one identifier refers to more than one method or constructor, the method is said to be *overloaded*. You could argue that this should be called identifier overloading rather than method overloading since it's the identifier that refers to more than one method, not the method that refers to more than one identifier. However in common usage this is called method overloading.

Which method an identifier refers to depends on the signature. The signature is the number, type, and order of the arguments passed to a method. The signature of the first constructor in the above program is Car(String, double). The signature of the second method is Car(String, double, double). Thus the first version of the Car() constructor is called when there is one String argument followed by one double argument and the second version is used when there is one String argument followed by two double arguments.

If there are no arguments to the constructor, or two or three arguments that aren't the right type in the right order, then the compiler generates an error because it doesn't have a method whose signature matches the requested method call. For example

Error: Method Car(double) not found in class Car.

Car.java line 17

Overloading constructors

It is often the case that overloaded methods are essentially the same except that one supplies default values for some of the arguments. In this case, your code will be easier to read and maintain (though perhaps *marginally* slower) if you put all your logic in the method that takes the most arguments, and simply invoke that method from all its overloaded variants that merely fill in appropriate default values.

This technique should also be used when one method needs to convert from one type to another. For instance one variant can convert a String to an int, then invoke the variant that takes the int as an argument.

This straight-forward for regular methods, but doesn't quite work for constructors because you can't simply write a method like this:

```
public Car(String licensePlate, double maxSpeed) {
  Car(licensePlate, 0.0, maxSpeed);
 }
```

Instead, to invoke another constructor in the same class from a constructor you use the keyword this like so:

```
public Car(String licensePlate, double maxSpeed) {
  this(licensePlate, 0.0, maxSpeed);
 }
```

Must this be the first line of the constructor?

For example,

```
public class Car {
 private String licensePlate; // e.g. "New York A456 324"
 private double speed;       // kilometers per hour
 private double maxSpeed;    // kilometers per hour
 // constructors
 public Car(String licensePlate, double maxSpeed) {
  this(licensePlate, 0.0, maxSpeed);
 }
 public Car(String licensePlate, double speed, double maxSpeed) {
  this.licensePlate = licensePlate;
  if (maxSpeed >= 0.0) {
   this.maxSpeed = maxSpeed;
   }
   else {
    maxSpeed = 0.0;
   }
   if (speed < 0.0) {
    speed = 0.0;
   }
   if (speed <= maxSpeed) {
    this.speed = speed;
   }
   else {
    this.speed = maxSpeed;
   }
```

```
  }
 // other methods...
}
```

This approach saves several lines of code. In also means that if you later need to change the constraints or other aspects of construction of cars, you only need to modify one method rather than two. This is not only easier; it gives bugs fewer opportunities to be introduced either through inconsistent modification of multiple methods or by changing one method but not others.

11.2 References to Objects

As you work with objects, one important thing going on behind the scenes is the use of references to those objects. When you assign objects to variables, or pass objects as arguments to methods, you are passing references to those objects, not the objects themselves or copies of those objects.

which shows a simple example of how references work.

```
import java.awt.Point;
class ReferencesTest {
 public static void main (String args[]) {
      Point pt1, pt2;
       pt1 = new Point(100, 100);
        pt2 = pt1;

       pt1.x = 200;
       pt1.y = 200;
     System.out.println("Point1: " + pt1.x + ", " + pt1.y);
     System.out.println("Point2: " + pt2.x + ", " + pt2.y);
   }
}
```

11.3 Usage of static and final

The Static

The keyword static makes the declaration belong to the class as a whole. A static field is shared by all instances of the class, instead of each instance having its own version of the field. A static method does not have a "this" object. A static method can operate on someone else's objects, but not via an implicit or explicit *this*.

The method where execution starts, main(), is a static method. The purpose of main() is to be an entry point to your code, not to track the state of one individual object. Static "per-class" declarations are different from all the "per-object" data you have seen to date.

You can apply the modifier static to four things in Java:

- **Data.** This is a field that belongs to the class, not a field that is stored in each individual object.
- **Methods.** These are methods that belong to the class, not individual objects.

- **Blocks**. These are blocks within a class that are executed only once, usually for some initialization. They are like instance initializers, but execute once per class, not once per object.
- **Classes.** These are classes that are nested in another class. Static classes were introduced with JDK

The final keyword

The final keyword is used in several different contexts as a modifier meaning that what it modifies cannot be changed in some sense.

final classes

You will notice that a number of the classes in Java library are declared final, e.g.

public final class String

This means this class will not be subclassed, and informs the compiler that it can perform certain optimizations it otherwise could not. It also provides some benefit in regard to security and thread safety.

The compiler will not let you subclass any class that is declared final. You probably won't want or need to declare your own classes final though.

final methods

You can also declare that methods are final. A method that is declared final cannot be overridden in a subclass. The syntax is simple, just put the keyword final after the access specifier and before the return type like this:

public final String convertCurrency()

final fields

You may also declare fields to be final. This is not the same thing as declaring a method or class to be final. When a field is declared final, it is a constant which will not and cannot change. It can be set once (for instance when the object is constructed, but it cannot be changed after that.) Attempts to change it will generate either a compile-time error or an exception (depending on how sneaky the attempt is).

Fields that are both final, static, and public are effectively named constants. For instance a physics program might define Physics.c, the speed of light as

```
public class Physics {
 public static final double c = 2.998E8;
}
```

In the SlowCar class, the speedLimit field is likely to be both final and static though it's private.

```
public class SlowCar extends Car {
 private final static double speedLimit = 112.65408; // kph == 70 mph
 public SlowCar(String licensePlate, double speed, double maxSpeed,
  String make, String model, int year, int numberOfPassengers, int numDoors) {
   super(licensePlate,
    (speed < speedLimit) ? speed : speedLimit,
```

```
  maxSpeed, make, model, year, numberOfPassengers, numDoors);
 }
 public void accelerate(double deltaV) {
  double speed = this.speed + deltaV;
  if (speed > this.maxSpeed) {
   speed = this.maxSpeed;
   }
  if (speed > speedLimit) {
   speed = speedLimit;
   }
  if (speed < 0.0) {
    speed = 0.0;
   }
  this.speed = speed;
   }
 }
```

final arguments

Finally, you can declare that method arguments are final. This means that the method will not directly change them. Since all arguments are passed by value, this isn't absolutely required, but it's occasionally helpful.

What can be declared final in the Car and MotorVehicle classes?

11.4 Access control of methods

Access to variables and methods in Java classes is accomplished through access modifiers. *Access modifiers* define varying levels of access between class members and the outside world (other objects). Access modifiers are declared immediately before the type of a member variable or the return type of a method. There are four access modifiers: the default access modifier, public, protected, and private.

Access modifiers affect the visibility not only of class members, but also of classes themselves. However, class visibility is tightly linked with packages, which are covered later in this chapter.

The Default Access Modifier

The default access modifier specifies that only classes in the same package can have access to a class's variables and methods. Class members with default access have a visibility limited to other classes within the same package. There is no actual keyword for declaring the default access modifier; it is applied by default in the absence of an access modifier. For example, the Alien class members all had default access because no access modifiers were specified. Examples of a default access member variable and method follow:

```
long length;
void getLength() {
 return length;
```

```
}
```

Notice that neither the member variable nor the method supplies an access modifier, so each takes on the default access modifier implicitly.

The public Access Modifier

The public access modifier specifies that class variables and methods are accessible to anyone, both inside and outside the class. This means that public class members have global visibility and can be accessed by any other object. Some examples of public member variables follow:

```
public int count;
public boolean isActive;
```

The protected Access Modifier

The protected access modifier specifies that class members are accessible only to methods in that class and subclasses of that class. This means that protected class members have visibility limited to subclasses. Examples of a protected variable and a protected method follow:

```
protected char middleInitial;
protected char getMiddleInitial() {
 return middleInitial;
}
```

The private Access Modifier

The private access modifier is the most restrictive; it specifies that class members are accessible only by the class in which they are defined. This means that no other class has access to private class members, even subclasses. Some examples of private member variables follow:

```
private String firstName;
private double howBigIsIt;
```

11.5 Inner Classes

An *inner class* is a class whose body is defined inside another class, referred to as the *top-level class*. For example:

```
public class Queue {
 Element back = null;
 public void add(Object o) {
Element e = new Element();
   e.data = o;
   e.next = back;
   back = e;
 }
 public Object remove() {
  if (back == null) return null;
  Element e = back;
  while (e.next != null) e = e.next;
   Object o = e.data;
```

```
  Element f = back;
  while (f.next != e) f = f.next;
  f.next = null;
  return o;
 }
 public boolean isEmpty() {
  return back == null;
 }
 // Here's the inner class
 class Element {
  Object data = null;
  Element next = null;
    }
}
```

Inner classes may also contain methods. They may not contain static members.

Inner classes in a class scope can be public, private, protected, final, abstract.

Inner classes can also be used inside methods, loops, and other blocks of code surrounded by braces ({}). Such a class is not a member, and therefore cannot be declared public, private, protected, or static.

The inner class has access to all the methods and fields of the top-level class, even the private ones.

The inner class's short name may not be used outside its scope. If you absolutely must use it, you can use the fully qualified name instead. (For example, Queue$Element) However, if you need to do this you should almost certainly have made it a top-level class instead or at least an inner class within a broader scope.

11.6 Command Line Arguments

If we need to pass the information into the program when we run it this is done by passing command line arguments to the main(). A command line argument is the information that directly follows the program's name on the command line when it is executed.

```
class PrintArgs {
 public static void main (String args[]) {
  for (int i = 0; i < args.length; i++) {
   System.out.println(args[i]);
   }
  }
 }
```

The name of the class is not included in the argument list.

Command line arguments are passed in an array of Strings. The first array component is the zeroth.

For example, consider this invocation:

$ java printArgs Hello There

args[0] is the string "Hello". args[1] is the string "There". args.length is 2.

All command line arguments are passed as String values, never as numbers. Later you'll learn how to convert Strings to numbers.

11.7 Inheritance Basic

Inheritance is the mechanism by which this is achieved. An object can inherit the variables and methods of another object. It can keep those it wants, and replace those it doesn't want.

For example, let us also expand the Car class so that a car also has a make, a model, a year, a number of passengers it can carry, four wheels, either two or four doors. That class might look like this:

```
public class Car {
 private String licensePlate;     // e.g. "New York A456 324"
 private double speed;           // kilometers per hour
 private double maxSpeed;        // kilometers per hour
 private String make;            // e.g. "Ford"
 private String model;          // e.g. "Taurus"
 private int    year;           // e.g. 1997, 1998, 1999, 2000, 2001, etc.
 private int   numberPassengers; // e.g. 4
 private int   numberWheels = 4; // all cars have four wheels
 private int   numberDoors;      // e.g. 4
 // constructors
 public Car(String licensePlate, double maxSpeed,
  String make, String model, int year, int numberOfPassengers,
  int numberOfDoors) {
   this(licensePlate, 0.0, maxSpeed, make, model, year,
    numberOfPassengers, numberOfDoors);
  }
 public Car(String licensePlate, double speed, double maxSpeed,
  String make, String model, int year, int numberOfPassengers) {
   this(licensePlate, speed, maxSpeed, make, model, year,
    numberOfPassengers, 4);
  }
 public Car(String licensePlate, double speed, double maxSpeed,
  String make, String model, int year, int numberOfPassengers,
  int numberOfDoors) {
   // I could add some more constraints like the
   // number of doors being positive but I won't
```

```
  // so that this example doesn't get too big.
  this.licensePlate = licensePlate;
  this.make = make;
  this.model = model;
  this.year = year;
  this.numberPassengers = numberOfPassengers;
  this.numberDoors = numberOfDoors;
  if (maxSpeed >= 0.0) {
   this.maxSpeed = maxSpeed;
  }
  else {
    maxSpeed = 0.0;
  }
  if (speed < 0.0) {
     speed = 0.0;
  }
  if (speed <= maxSpeed) {
    this.speed = speed;
  }
  else {
    this.speed = maxSpeed;
  }
  }
// getter (accessor) methods
public String getLicensePlate() {
 return this.licensePlate;
}
public String getMake() {
 return this.make;
}
public String getModel() {
 return this.model;
}
public int getYear() {
 return this.year;
}
public int getNumberOfPassengers() {
```

```
  return this.numberPassengers;
 }
 public int getNumberOfWheels() {
  return this.numberWheels;
 }
 public int getNumberOfDoors() {
  return this.numberDoors;
 }
 public double getMaxSpeed() {
  return this.maxSpeed;
 }
 public double getSpeed() {
  return this.speed;
 }
 // setter method for the license plate property
 public void setLicensePlate(String licensePlate) {
  this.licensePlate = licensePlate;
 }
 // accelerate to maximum speed
 // put the pedal to the metal
 public void floorIt() {
  this.speed = this.maxSpeed;
 }
 public void accelerate(double deltaV) {
   this.speed = this.speed + deltaV;
   if (this.speed > this.maxSpeed) {
     this.speed = this.maxSpeed;
    }
   if (this.speed <  0.0) {
     this.speed = 0.0;
   }          }
 }
```

Obviously this doesn't exhaust everything there is to say about a car. Which properties you choose to include in your class depends on your application.

Now suppose you also need a class for a motorcycle. A motorcycle also has a make, a model, a year, a speed, a maximum speed, a weight, a price, a number of passengers it can carry, two wheels and many other properties you can represent with fields. A class that represents a motorcycle might look like this:

```
public class Motorcycle {
 private String licensePlate;              // e.g. "New York A456 324"
 private double speed;                     // kilometers per hour
 private double maxSpeed;                  // kilometers per hour
 private String make;                      // e.g. "Harley-Davidson"
 private String model;                     // e.g. "panhead"
 private int   year;                       // e.g. 1997, 1998, 1999, 2000, 2001, etc.
 private int   numberPassengers;           // e.g. 4
 private int   numberWheels = 2;           // all motorcycles have two wheels
  // constructors
 public Motorcycle(String licensePlate, double maxSpeed,
 String make, String model, int year, int numberOfPassengers) {
  this(licensePlate, maxSpeed, make, model, year, numberOfPassengers);
 }
 public Motorcycle(String licensePlate, double speed, double maxSpeed,
 String make, String model, int year, int numberOfPassengers) {
  this(licensePlate, speed, maxSpeed, make, model, year,
   numberOfPassengers);
 }
 public Motorcycle(String licensePlate, double speed, double maxSpeed,
 String make, String model, int year, int numberOfPassengers) {
  // I could add some more constraints like the
  // number of doors being positive but I won't
  // so that this example doesn't get too big.
  this.licensePlate = licensePlate;
  this.make = make;
  this.model = model;
  this.year = year;
  this.numberPassengers = numberOfPassengers;
  if (maxSpeed >= 0.0) {
   this.maxSpeed = maxSpeed;
  }
  else {
    maxSpeed = 0.0;
  }
  if (speed < 0.0) {
```

```
    speed = 0.0;
   }
  if (speed <= maxSpeed) {
    this.speed = speed;
   }
  else {
    this.speed = maxSpeed;
   }
 }

// getter (accessor) methods
public String getLicensePlate() {
 return this.licensePlate;
 }
public String getMake() {
 return this.make;
 }
public String getModel() {
 return this.model;
 }
public int getYear() {
 return this.year;
 }
public int getNumberOfPassengers() {
 return this.numberPassengers;
 }
public int getNumberOfPassengers() {
 return this.numberWheels;
 }
public double getMaxSpeed() {
 return this.maxSpeed;
 }
public double getSpeed() {
 return this.speed;
 }
// setter method for the license plate property
public void setLicensePlate(String licensePlate) {
```

```
  this.licensePlate = licensePlate;
 }
 // accelerate to maximum speed
 // put the pedal to the metal
 public void floorIt() {
  this.speed = this.maxSpeed;
 }
 public void accelerate(double deltaV) {
   this.speed = this.speed + deltaV;
   if (this.speed > this.maxSpeed) {
    this.speed = this.maxSpeed;
    }
   if (this.speed < 0.0) {
     this.speed = 0.0;
    }
 }
}
```

There's a lot of overlap between the class definitions for Car and Motorcycle. In fact the only things that are different are the constructors and a few of the fields. Inheritance takes advantage of the overlap.

In this example you begin by defining a more general MotorVehicle class.

```
public class MotorVehicle {
 private String licensePlate;     // e.g. "New York A456 324"
 private double speed;           // kilometers per hour
 private double maxSpeed;        // kilometers per hour
 private String make;           // e.g. "Harley-Davidson", "Ford"
 private String model;          // e.g. "Fatboy", "Taurus"
 private int    year;           // e.g. 1998, 1999, 2000, 2001, etc.
 private int   numberPassengers; // e.g. 4
 // constructors
 public MotorVehicle(String licensePlate, double maxSpeed,
  String make, String model, int year, int numberOfPassengers) {
  this(licensePlate, 0.0, maxSpeed, make, model, year, numberOfPassengers);
 }
 public MotorVehicle(String licensePlate, double speed, double maxSpeed,
  String make, String model, int year, int numberOfPassengers) {
  // I could add some more constraints like the
```

```
  // number of doors being positive but I won't
  // so that this example doesn't get too big.
  this.licensePlate = licensePlate;
  this.make = make;
  this.model = model;
  this.year = year;
  this.numberPassengers = numberOfPassengers;
  if (maxSpeed >= 0.0) {
   this.maxSpeed = maxSpeed;
  }
  else {
    maxSpeed = 0.0;
  }
  if (speed < 0.0) {
     speed = 0.0;
  }
  if (speed <= maxSpeed) {
    this.speed = speed;
  }
  else {
    this.speed = maxSpeed;
  }
}
// getter (accessor) methods
public String getLicensePlate() {
 return this.licensePlate;
}
public String getMake() {
 return this.make;
}
public String getModel() {
 return this.model;
}
public int getYear() {
 return this.year;
}
```

```
 public int getNumberOfPassengers() {
  return this.numberPassengers;
 }
 public int getNumberOfPassengers() {
  return this.numberWheels;
 }
 public double getMaxSpeed() {
  return this.maxSpeed;
 }

 public double getSpeed() {
  return this.speed;
 }
 // setter method for the license plate property
 protected void setLicensePlate(String licensePlate) {
  this.licensePlate = licensePlate;
 }
 // accelerate to maximum speed
 // put the pedal to the metal
 public void floorIt() {
  this.speed = this.maxSpeed;
 }
 public void accelerate(double deltaV) {
   this.speed = this.speed + deltaV;
   if (this.speed > this.maxSpeed) {
    this.speed = this.maxSpeed;
    }
   if (this.speed < 0.0) {
     this.speed = 0.0;
    }
  }
}
```

The MotorVehicle class has all the characteristics shared by motorcycles and cars, but it leaves the number of wheels unspecified, and it doesn't have a numberDoors field since not all motor vehicles have doors. It also makes the fields and the setLicensePlate() method protected instead of private and public.

The Motorcycle subclass

The MotorVehicle class has all the characteristics shared by motorcycles and cars, but it leaves

the number of wheels unspecified, and it doesn't have a numberDoors field since not all motor vehicles have doors.

Next you define two subclasses of MotorVehicle, one for cars and one for motorcycles. To do this you use the keyword extends.

```
public class Motorcycle extends MotorVehicle {
 private int numberWheels = 2;
 // constructors
 public Motorcycle(String licensePlate, double maxSpeed,
  String make, String model, int year, int numberOfPassengers) {
   this(licensePlate, 0.0, maxSpeed, make, model, year, numberOfPassengers);
  }
 public Motorcycle(String licensePlate, double speed, double maxSpeed,
  String make, String model, int year, int numberOfPassengers) {
   // invoke superclass constructor
   super(licensePlate, speed, maxSpeed, make, model, year,
    numberOfPassengers);
  }
 public int getNumberOfWheels() {
   return this.numberWheels;
  }
```

The Car subclass

```
public class Car extends MotorVehicle {
 private int numberWheels = 4;
 private int numberDoors;
 // constructors
 public Car(String licensePlate, double maxSpeed,
  String make, String model, int year, int numberOfPassengers,
  int numberOfDoors) {
   this(licensePlate, 0.0, maxSpeed, make, model, year, numberOfPassengers,
    numberOfDoors);
  }
 public Car(String licensePlate, double speed, double maxSpeed,
  String make, String model, int year, int numberOfPassengers) {
   this(licensePlate, speed, maxSpeed, make, model, year,
    numberOfPassengers, 4);
  }
 public Car(String licensePlate, double speed, double maxSpeed,
```

```
 String make, String model, int year, int numberOfPassengers,
 int numberOfDoors) {
  super(licensePlate, speed, maxSpeed, make, model,
   year, numberOfPassengers);
  this.numberDoors = numberOfDoors;
 }
public int getNumberOfWheels() {
 return this.numberWheels;
 }

public int getNumberOfDoors() {
 return this.numberDoors;
 }
  }
```

It may look like these classes aren't as complete as the earlier ones, but that's incorrect. Car and Motorcycle each inherit the members of their superclass, MotorVehicle. Since a MotorVehicle has a make, a model, a year, a speed, a maximum speed, a number of passengers, cars and motorcycles also have makes, models, years, speeds, maximum speeds, and numbers of passengers. They also have all the public methods the superclass has. They do not have the same constructors, though they can invoke the superclass constructor through the super keyword, much as a constructor in the same class can be invoked with the this keyword.

11.8 Using superclass variable and constructor

The Car-Motorcycle-MotorVehicle example showed single-level inheritance. There's nothing to stop you from going further. You can define subclasses of cars for compacts, station wagons, sports coupes and more. For example, this class defines a compact as a car with two doors:

```
public class Compact extends Car {
 // constructors
 public Compact(String licensePlate, double maxSpeed,
 String make, String model, int year, int numberOfPassengers) {
  this(licensePlate, 0.0, maxSpeed, make, model, year, numberOfPassengers);
 }
 public Compact(String licensePlate, double speed, double maxSpeed,
 String make, String model, int year, int numberOfPassengers) {
  super(licensePlate, speed, maxSpeed, make, model,
   year, numberOfPassengers, 2);
 }
  }
```

Compact not only inherits from its immediate superclass, Car, but also from Car's superclass, MotorVehicle. Thus the Compact class also has a make, a model, a year and so on. There's no

limit to this chain of inheritance, though getting more than four or five classes deep makes code excessively complex.

Multiple Inheritance

Some object oriented languages, notably C++, allow a class to inherit from more than one unrelated class. This is called multiple inheritance and is different from the multi-level inheritance in this section. Most of the things that can be accomplished via multiple inheritance in C++ can be handled by interfaces in Java.

Constructors

A constructor creates a new instance of the class. It initializes all the variables and does any work necessary to prepare the class to be used. In the line

Car c = new Car();

Car() is the constructor. A constructor has the same name as the class.

If no constructor exists Java provides a generic one that takes no arguments (a *noargs constructor*), but it's better to write your own. You make a constructor by writing a method that has the same name as the class. Thus the Car constructor is called Car().

Constructors do not have return types. They do return an instance of their own class, but this is implicit, not explicit.

The following method is a constructor that initializes license plate to an empty string, speed to zero, and maximum speed to 120.0.

```
Car() {
  this.licensePlate = "";
   this.speed  = 0.0;
  this.maxSpeed = 120.0;
 }
```

Better yet, you can create a constructor that accepts three arguments and use those to initialize the fields as below.

```
Car(String licensePlate, double speed, double maxSpeed) {
  this.licensePlate = licensePlate;
   this.speed  = speed;
  if (maxSpeed > 0) this.maxSpeed = maxSpeed;
  else this.maxSpeed = 0.0;
  if (speed > this.maxSpeed) this.speed = this.maxSpeed;
   if (speed < 0) this.speed = 0.0;
  else this.speed = speed;
 }
```

Or perhaps you always want the initial speed to be zero, but require the maximum speed and license plate to be specified:

```
Car(String licensePlate, double maxSpeed) {
  this.licensePlate = licensePlate;
   this.speed  = 0.0;
```

```
  if (maxSpeed > 0) this.maxSpeed = maxSpeed;
  else this.maxSpeed = 0.0;
```

Here's the complete class:

```
class Car {
 String licensePlate; // e.g. "New York A456 324"
 double speed;       // kilometers per hour
 double maxSpeed;    // kilometers per hour
 Car(String licensePlate, double maxSpeed) {
  this.licensePlate = licensePlate;
   this.speed  = 0.0;
  if (maxSpeed > 0) this.maxSpeed = maxSpeed;
  else this.maxSpeed = 0.0;
 }
 // getter (accessor) methods
 String getLicensePlate() {
  return this.licensePlate;
 }
 double getMaxSpeed() {
  return this.maxSpeed;
 }
 double getSpeed() {
  return this.speed;
 }
 // accelerate to maximum speed
 // put the pedal to the metal
 void floorIt() {
  this.speed = this.maxSpeed;
 }
  void accelerate(double deltaV) {
   this.speed = this.speed + deltaV;
   if (this.speed > this.maxSpeed) {
     this.speed = this.maxSpeed;
    }
   if (this.speed <  0.0) {
     this.speed = 0.0;
    }
}}
```

Notice that I've taken out several things:

- the initialization of the fields
- the setter methods

Using Constructors

The next program uses the constructor to initialize a car rather than setting the fields directly.

```
class CarTest7 {
 public static void main(String args[]) {
   Car c = new Car("New York A45 636", 123.45);
  System.out.println(c.getLicensePlate() + " is moving at " + c.getSpeed() +
   " kilometers per hour.");

  for (int i = 0; i < 15; i++) {
    c.accelerate(10.0);
   System.out.println(c.getLicensePlate() + " is moving at " + c.getSpeed()
    + " kilometers per hour.");
   }
 }

}
```

You no longer need to know about the fields licensePlate, speed and maxSpeed. All you need to know is how to construct a new car and how to print it.

You may ask whether the setLicensePlate() method is still needed since it's now set in a constructor. The general answer to this question depends on the use to which the Car class is to be put. The specific question is whether a car's license plate may need to be changed after the Car object is created.

Some classes may not change after they're created; or, if they do change, they'll represent a different object. The most common such class is String. You cannot change a string's data. You can only create a new String object. Such objects are called *immutable*.

Self Learning Exercises

1. What is the difference between superclass & subclass?
2. Which keyword is used to inherit a class?
3. Subclasses methods can access superclass members/ attributes at all times? True/False
4. When can subclasses not access superclass members?
5. Which class does begin Java class hierarchy?
6. Object class is a superclass of all other classes? True/False
7. Java supports multiple inheritance? True/False

11.9 Dynamic Method Dispatch

Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved

at run time , rather then compile time. Dynamic method is important because this is how the java implements run time polymorphism.

11.10 Method Overriding

Suppose that one day you've just finished your Car class. It's been plugged into your traffic simulation which is chugging along merrily simulating traffic. Then your pointy haired boss rolls in the door, and tells you that he needs the Car class to not accelerate past the 70 miles per hour (pointy haired bosses rarely understand the metric system) even if the car's a Ferrari with a maximum speed in excess of 200 miles per hour.

What are you going to do? Your first reaction may be to change the class that you already wrote so that it limits the speed of all the cars. However you're using that class elsewhere and things will break if you change it.

You could create a completely new class in a different file, either by starting from scratch or by copying and pasting. This would work, but it would mean that if you found a bug in the Car class now you'd have to fix it in two files. And if you wanted to add new methods to the Car class, you'd have to add them in two files. Still this is the best you could do if you were writing in C or some other traditional language.

Overriding Methods: The Solution

The object oriented solution to this problem is to define a new class, call it SlowCar, which inherits from Car and imposes the additional constraint that a car may not go faster than 70 mph (112.65 kph).

To do this you'll need to adjust the two places that speed can be changed, the constructor and the accelerate() method. The constructor has a different name because all constructors are named after their classes but the accelerate() method must be *overridden*. This means the subclass has a method with the same signature as the method in the superclass.

```
public class SlowCar extends Car {
 private static final double speedLimit = 112.65408; // kph == 70 mph
 public SlowCar(String licensePlate, double speed, double maxSpeed,
  String make, String model, int year, int numberOfPassengers, int numDoors) {
   super(licensePlate, 0, maxSpeed, make, model, year,
    numberOfPassengers, numDoors);
   this.accelerate(speed);
     }
 public void accelerate(double deltaV) {
    double speed = this.getSpeed() + deltaV;
    if (speed > speedLimit) {
     super.accelerate(speedLimit - this.getSpeed());
     }
    else {
     super.accelerate(deltaV);
     }
 }
```

}

The first thing to note about this class is what it doesn't have, getSpeed(), getLicensePlate(), getMaximumSpeed(), setLicensePlate() methods or speed, maxSpeed and numDoors fields. All of these are provided by the superclass Car. Nothing about them has changed so they don't need to be repeated here.

Next look at the accelerate() method. This is different than the accelerate() method in Car. It imposes the additional constraint.

The constructor is a little more complicated. First note that if you're going to use a non-default constructor, that is a constructor with arguments, you do need to write a constructor for the subclass, even if it's just going to do the exact same thing as the matching constructor in the superclass. You cannot simply inherit Car's constructor because that constructor is named Car() and this one must be named SlowCar().

The constructor needs to set the value of name, url, and description. However they're not accessible from the subclass. Instead they are set by calling the superclass's constructor using the keyword super. When super is used as a method in the first non-blank line of a constructor, it stands for the constructor of this class's superclass.

The immediate superclass's constructor *will* be called in the first non-blank line of the subclass's constructor. If you don't call it explicitly, then Java will call it for you with no arguments. It's a compile time error if the immediate superclass doesn't have a constructor with no arguments and you don't call a different constructor in the first line of the subclass's constructor.

The use of the ternary operator in the constructor call is unusual. However, it's necessary to meet the compiler's requirement that the invocation of super be the first line in the subclass constructor. Otherwise this could be written more clearly using only if-else.

11.11 Abstract Classes

Java allows methods and classes to be declared abstract. An abstract method is not actually implemented in the class. It is merely declared there. The body of the method is then implemented in subclasses of that class. An abstract method must be part of an abstract class. You create abstract classes by adding the keyword abstract after the access specifier, e.g.

public abstract class MotorVehicle

Abstract classes cannot be instantiated. It is a compile-time error to try something like

MotorVehicle m = new MotorVehicle();

when MotorVehicle has been declared to be abstract. MotorVehicle is actually a pretty good example of the sort of class that might be abstract. You're unlikely to be interested in a generic motor vehicle. Rather you'll have trucks, motorcycles, cars, go-carts and other subclasses of MotorVehicle, but nothing that is only a MotorVehicle.

An abstract method provides a declaration but no implementation. In other words, it has no method body. Abstract methods can only exist inside abstract classes and interfaces. For example, the MotorVehicle class might have an abstract fuel() method:

public abstract void fuel();

Car would override/implement this method with a fuel() method that filled the gas tank with gasoline. EighteenWheelerTruck might override this method with a fuel() method that filled its gas tank with diesel. ElectricCar would override/implement this method with a fuel() method that plugged into the wall socket.

11.12 Summary

- Overloading is when the same method or operator can be used on many different types of data.
- The keyword static makes the declaration belong to the class as a whole. A static field is shared by all instances of the class, instead of each instance having its own version of the field.
- Access modifiers define varying levels of access between class members and the outside world (other objects). Access modifiers are declared immediately before the type of a member variable or the return type of a method. There are four access modifiers: the default access modifier, public, protected, and private.
- An inner class is a class whose body is defined inside another class, referred to as the top-level class.
- If we need to pass the information into the program when we run it this is done by passing command line arguments to the main(). A command line argument is the information that directly follows the program's name on the command line when it is executed.
- Inheritance is the mechanism by which this is achieved. An object can inherit the variables and methods of another object.
- A constructor creates a new instance of the class. It initializes all the variables and does any work necessary to prepare the class to be used.
- Constructors do not have return types. They do return an instance of their own class, but this is implicit, not explicit.
- Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time , rather then compile time. Dynamic method is important because this is how the java implements run time polymorphism.

11.13 Glossary

abstract : A Java keyword used in a class definition to specify that a class is not to be instantiated, but rather inherited by other classes. An abstract class can have abstract methods that are not implemented in the abstract class, but in subclasses.

abstract class : A class that contains one or more *abstract methods*, and therefore can never be instantiated. Abstract classes are defined so that other classes can extend them and make them concrete by implementing the abstract methods.

Constructor: A special type of instance method that creates a new object. In Java, constructors have the same name as their class and have no return value in their declaration.

method : A function defined in a class.

static : A Java keyword used to define a variable as a class variable. Classes maintain one copy of class variables regardless of how many instances exist of that class. static can also be used to define a method as a class method. Class methods are invoked by the class instead of a specific instance, and can only operate on class variables.

11.14 Further Readings

1. Mastering Java2, J2SE1.4 (Paperback) John Zukowski

2. Java: The Complete Reference ,Herbert Schildt Computers - 2005
3. Java 2: The Complete Reference, Herbert Schildt - Computers - 2002
4. Java How to Program, 7/E (Harvey & Paul) Deitel & Associates Publisher: Prentice Hall ,2007

11.15 Answers to self learning exercises

1. A super class is a class that is inherited whereas subclass is a class that does the inheriting.
2. Extends
3. False
4. When superclass is declared as private.
5 .Object class
6. True
7. False

11.16 Unit End Questions

1. What is method overloading.
2. What is constructor, how you will overload the constructor
3. Explain the use of private , public and protected.
4. What is inner class, explain it by example.
5. What is Abstract class
6. What is the use of static and final keywords
7. What do you mean by command line arguments.
8. What is a class?
9. How do classes help you to organize your programs?
10. What are the three parts of a simple, empty class?
11. What two elements do you add to complete the class?
12. How do you create an object from a class?
13. What is inheritance and how does it help you create new classes quickly?
14. What is a subclass and a superclass?
15. How do you create a subclass?
16. How do you override a method inherited from a superclass?

UNIT - XII

Packages and Interfaces

Structure of the Unit

12.0 Objective

After going through this unit you will be able to:

- Define CLASSPATH, java package, importing package.
- Describe access protection in packages, interfaces
- Describe implementing interfaces, applying interfaces, variables in interface
- Describe usage of abstract class etc.

12.1 Packages

Java provides a powerful means of grouping related classes and interfaces together in a single unit: packages. (You learn about interfaces a little later in this chapter.) Put simply, *packages* are groups of related classes and interfaces. Packages provide a convenient mechanism for managing a large group of classes and interfaces, while avoiding potential naming conflicts. The Java API itself is implemented as a group of packages.

As an example, the Alien and Enemy classes could be fitten nicely into an Enemy package-along with any other enemy objects. By placing classes into a package, you also allow them to benefit from the default access modifier, which provides classes in the same package with access to each other's class information.

Declaring a Package

Packages are declared using the package keyword followed by a package name. This must occur as the first statement in a Java source file, excluding comments and white space. Here is an example:

```
package mammals;

class AMammal  {

  ...body of class AMammal}
```

In this example, the package name is mammals. The class AMammal is now considered a part of this package. Including other classes in package mammals is easy: Simply place an identical package line at the top of those source files as well. Because every class is generally placed in its own source file, every source file that contains classes for a particular package must include this line. There can be only one package statement in any source file.

Note that the Java compiler only requires classes that are declared public to be put in a separate source file. Nonpublic classes can be put in the same source file. Although it is good programming practice to put each of these in its own source file, one package statement at the top of a file will apply to all classes declared in that file.

Java also supports the concept of package hierarchy. This is similar to the directory hierarchy found in many operating systems. This is done by specifying multiple names in a package statement, separated by a period. In the following code, class AMammal belongs to the package mammal that is in the animal package hierarchy:

```
package animal.mammal;

class AMammal {

  ...body of class Amammal
```

} This allows grouping of related classes into a package and then grouping related packages into a larger package. To reference a member of another package, the package name is prepended

to the class name. This is an example of a call to method promptForName in class ACat in subpackage mammal in package animal:

animal.mammal.Cat.promptForName();

The analogy to a directory hierarchy is reinforced by the Java interpreter. The Java interpreter requires that the .class files be physically located in a subdirectory with a name that matches the subpackage name, when accessing a member of a subpackage. If the previous example were located on a UNIX system, the class promptForName would be located as follows:

animal/mammal/ACat.class

Of course, the directory-naming conventions will be different for different operating systems. The Java compiler will happily place the .class files into the same directory as the source files. It may be necessary to move the resulting class files into the appropriate directory if the source files are not in the class files.

The class files can also be placed directly into the desired directory by specifying the -d (directory) option on the javac command line. To continue the example, the following code places the resulting output files in the subdirectory animal/mammal/ACat of the current directory:

> javac -d animal/mammal/ACat ACat.java

All classes actually belong to a package even if not explicitly declared. Even though in the examples used throughout most of this book no package name has been declared, the programs can be compiled and run without problems. As usual in Java, what is not explicitly declared automatically gets default values. In this case, there is a default unnamed package to which all such packages belong. The package does not have an explicit name, and it is not possible for other packages to reference an unnamed package. Therefore, no other package is able to reference most of the examples in this book as they are now. It is a good idea to place all nontrivial classes into packages.

12.1.1 Accessing Packages

Recall that you can reference packages by prepending a complete package name to a class. A shortcut-using the import statement-can be used when there are many references to a particular package or the package name is long and unwieldy.

The import statement is used to include a list of packages to be searched for a particular class. The syntax of an import statement follows:

import packagename;

import is a keyword and packagename is the name of the package to be imported. The statement must end with ;. The import statement should appear before any class declarations in a source file. Multiple import statements can also be made. The following is an example of an import statement:

import mammal.animal.Cat;

In this example all the members (for example, variables, methods) of class Cat can now be directly accessed by simply specifying their name without prepending the entire package name.

This shortcut poses both an advantage and a disadvantage. The advantage is that the code is no longer cluttered with long names and is easier to type. The disadvantage is that it is more difficult to determine from which package a particular member came. This is especially true when a large number of packages are imported.

import statements can also include the wildcard character *. The asterisk specifies that all classes

located in a hierarchy be imported, rather than just a single class. For example, the following code imports all classes that are in the mammal.animal subpackage:

import mammal.animal.*;

This is a handy way to bring all classes from a particular package.

The import statement has been used quite heavily in the examples in this book. It has typically been used to bring in various parts of the Java API. By default the java.lang.* set of classes is always imported. The other Java class libraries must be explicitly imported. For example, the following code brings in all the windowing toolkit graphic and image classes.

import java.awt.Graphics;

import java.awt.Image;

12.1.2 Package-Naming Conventions

Packages can be named anything that follows the standard Java naming scheme. By convention, however, packages begin with lowercase letters to make it simpler to distinguish package names from class names when looking at an explicit reference to a class. This is why class names are, by convention, begun with an uppercase letter. For example, when using the following convention, it is immediately obvious that mammal and animal are package names and Cat is a class name. Anything following the class name is a member of that class:

mammal.animal.Cat.promptForName();

Java follows this convention for the Java internals and API. The System.out.println() method that has been used follows this convention. The package name is not explicitly declared because java.lang.* is always imported implicitly. System is the class name from package java.lang.*, and it is capitalized. The full name of the method is

java.lang.System.out.println();

Every package name must be unique to make the best use of packages. Naming conflicts that will cause runtime errors will occur if duplicate package names are present. The class files may step on each other in the class directory hierarchy if there is duplication.

It is not difficult to keep package names unique if a single individual or small group does the programming. Large-group projects must iron out package-name conventions early in the project to avoid chaos.

No single organization has control over the World Wide Web, and many Java applications will be implemented over the Web. Remember also that Web servers are likely to include Java applets from multiple sources. It seems impossible to avoid duplicate package names.

Sun recognized this problem late in the development stage but before officially releasing Java. It developed a convention that ensures package-name uniqueness using a variation on domain names, which are guaranteed to be unique: to use the domain name in a reverse manner. Domain mycompany.com would prefix all package names with com.mycompany. Educational institute city.state.edu would prefix package names with edu.state.city. This neatly solves the naming problem and generates a very nice tree structure for all Java class libraries.

12.2 CLASSPATH : Environment Variable

The Java interpreter must find all the referenced class libraries when running a Java application.

By default, Java looks in the Java install tree for the libraries. It is usually better when developing code to put your own class libraries someplace else. An environment variable named CLASSPATH can be used to tell Java where these libraries are located. CLASSPATH contains a list of directories to search for Java class library trees. The syntax of the list will vary according to the operating system being used. On UNIX systems, CLASSPATH contains a colon-separated list of directory names. Under Windows, the list is separated by ;. The following is a CLASSPATH statement for a UNIX system:

CLASSPATH=/grps/IT/Java/classes:/opt/apps/Java

This tells the Java interpreter to look in the /grps/IT/Java/classes and /opt/apps/Java directories for class libraries.

12.3 Defining an Interface

Interfaces are Java's way of cutting down on the complexity of a project. Single inheritance makes it easy to find out method or class origins. However, this may be somewhat limiting. C++ permits multiple inheritance, which can be a breeding ground of needless complexity but does allow for ultimate flexibility. Java is less flexible but also less complex. It uses single inheritance for simplicity's sake but uses interfaces to bring in functionality from other classes.

A review of methods is in order before continuing. Methods are similar to functions in other languages. A method is a unit of code that is called and returns a value. Methods perform work on the variables and contain executable code. They are always internal to a class and are associated with an object.

The concept of interfaces is one of the main differences in project design between traditional C and Java application development. The C and other procedural programming language systems' development life cycle often begins with the definition of the application function names and their arguments as empty "black boxes." In other words, the programmers know the necessary argument parameters when programming code that calls these functions without knowing how they are implemented in the function. Thus they can develop code without first fully fleshing out all the functions. In C, this could be done by defining a function prototype for all the functions and then implementing them as resources and schedules permit. How is this accomplished in Java? Through interfaces.

An interface only defines a method's name, return type, and arguments. It does not include executable code or point to a particular method. Think of an interface as a template of structure, not usage.

Interfaces are used to define the structure of a set of methods that will be implemented by classes yet to be designed and coded. In other words, the calling arguments and the return value must conform to the definition in the interface. The compiler checks this conformity. However, the code internal to one method defined by the interface may achieve the intended result in a wildly different way than a method in another class.

The concept of using interfaces is a variation on inheritance used heavily in Java. The chief benefit of interfaces is that many different classes can all implement the same interface. This guarantees that all such classes will implement a few common methods. It also ensures that all the classes will implement these common methods using the same return type, name, and arguments.

Let's get theoretical. Say that a method is defined in the interface as boolean. The only argument is an input string. In the comments section, the method is said to test something to see if it is black and return true or false. Methods using this interface could be written to test cats, pavement, teeth, screen color-just about anything. The only thing these methods have in common is that they

have the same definition. Any programmer seeing the method name knows the purpose of the method and the calling arguments.

Java, of course, does not require all classes that implement a method of a certain name to use the interface for argument verification. No language can make up for poor project management. It does, however, provide the structure for use.

An **interface** is a collection of constants and method declarations. The method declarations do not include an implementation (there is no method body.)

A child class that extends a parent class can also implement an interface to gain some additional behavior. Here is what an interface definition looks like:

```
interface InterfaceName
{
 constant definitions
 method declarations (without implementations.)
}
```

A method declaration is simply an access modifier, return type, and method signature followed by a semicolon.

This looks somewhat like a class definition. But no objects can be constructed from it. Objects can be constructed from a class that **implements** an interface. A class definition implements an interface like this:

```
class SomeClass extends SomeParent implements interfaceName
{
}
```

A class always extends just one parent but may implement several interfaces.

Example interface

Here is an example interface definition:

```
interface MyInterface
{
 public final int aConstant = 32;      // a constant
 public final double pi = 3.14159;     // a constant
 public void methodA( int x );        // a method declaration
 double methodB();                   // a method declaration
}
```

The constants don't have to be separated from the methods, but doing so makes the interface easier to read. A method in an interface cannot be made private. A method in an intervace is public by default. In the example methodB is public even though it does not say so.

- A class that *implements* an interface must implement each method in the interface.
- Each method must be public (this happens by default).
- Constants from the interface can be used as if they had been defined in the class.
- Constants should not be redefined in the class.

12.3.1 Implementing an Interface

A class definition must always extend one parent, but it can *implement* zero or more interfaces:

```
class SomeClass extends Parent implements SomeInterface
{
  ordinary class definition body
}
```

The body of the class definition is the same as always. However, since it implements an interface the body must have a definition of each of the methods from the interface. The class definition can use access modifiers as usual. Here is a class definition that implements three interfaces:

```
public class BigClass extends Parent
  implements  InterfaceA, InterfaceB, InterfaceC
{
  ordinary class definition body
}
```

Now BigClass must provide a method definition for every method in each of the interfaces. Any number of classes can implement the same interfaces. Here is another class definition:

```
public class SmallClass implements InterfaceA
{
  ordinary class definition body
}
```

Example Problem

Let us create a database program for a store. The store sells:

- Goods, each of which has the attributes:
  - description
  - price
- The goods are either:
  - Food—with an attribute "calories"
  - Toy—with an attribute "minimumAge", or
  - Book—with an attribute "author."

Of these goods, toys and books are taxable, but food is not. There are many other things that are taxable, such as services or entertainment so we want to have the concept "taxable" as a separate concept, not part of the concept of Goods.

Here is what the concept Taxable looks like:

- A Taxable item,
  - has a taxRate of 6 percent,
  - has a calculateTax() method.

When implemented in Java, these concepts will appear as classes and an interface.

The children classes **extend** their parent. This is shown with a solid arrow pointing at the parent.

The solid arrows show *inheritance*. The three children inherit the display() method.

Two of the classes **implement** the interface. This is shown with a dotted arrow pointing at the interface.

The dotted arrow show what a class *must implement*. The Toy and Book classes must implement the calculateTax() method.

Starting the Program

Here is a class definition for Goods:

```
class Goods
{
 String description;
 double price;
 Goods( String des, double pr )
 {
  description = des;
   price       = pr;
 }
 void display()
 {
  System.out.println( "item: " + description +
      " price: " + price );
 }
}
```

Here is a skeleton of the class Food. Recall that it adds the variable calories.

Adding the Interface

The child class Food extends the parent class. It uses super to use the parent's constructor and the parent's display method.

```
class Food extends Goods
{
 double calories;
 Food( String des, double pr, double cal)
 {
   super( des, pr );
   calories = cal ;  }
 void display()
 {
  super.display( );
  System.out.println( "calories: " + calories );  }}
```

Here is Taxable:

- A Taxable item,
  - has a taxRate of 6 percent, which should be a double constant.
  - and a calculateTax() method. which should return a double value.

The Taxable interface looks like this:

```
interface Taxable
{
 final double _____ = _______ ;
 double _____________() ;
}
```

The final says that what follows is a constant, not a variable (variables are not allowed in interfaces.) In fact, the final can be omitted since the identifier that follows will automatically be a constant. The " = value " cannot be omitted.

The method declaration (in the second line) is public by default.

Adding another Class

Since taxRate is a constant, it must be set to a value, here, 6 percent. Here is a partial definition of Toy. Recall that it:

- Has a parent Goods.
- Adds a variable minimumAge.
- Is taxable.

```
class Toy extends Goods _____________ _____________
{
 int minimumAge;
 Toy( String des, double pr, int min)
 {
   super( des, pr );
  minimumAge = min ;
 }
 void display()
 {
  super.display() ;
  System.out.println( "minimum age: " + minimumAge );
 }
 public double _____________()  // implementing the interface
 {
  return price * _____________ ;
```

```
  }
}
```

Adding the Remaining Class

The constant taxRate is used in the calculateTax() method just as if it had been defined in the Toy class. Also, it can be used in methods of the class other than those listed in the interface.

```
class Toy extends Goods implements Taxable
{
 int minimumAge;
 Toy( String des, double pr, int min)
 {
   super( des, pr );
  minimumAge = min ;
 }
 void display()
 {
  super.display() ;
  System.out.println( "minimum age: " + minimumAge );
 }
 public double calculateTax()
 {
  return price * taxRate ;
 }
}
```

The calculateTax() method must be made public.

12.3.2 Using an interface as a data type

An interface can be used as a data type for a reference variable. Since Toy and Book implement Taxable, they can both be used with a reference variable of type Taxable:

```
 public static void main ( String[] args )
 {
  Taxable item1 = new Book ( "Emma", 24.95, "Austin" );
  Taxable item2 = new Toy  ( "Leggos", 54.45, 8 );

  System.out.println( "Tax on item 1 "+ item1.calculateTax() );
  System.out.println( "Tax on item 2 "+ item2.calculateTax() );
 }
```

The compiler has been told in the interface that all Taxable objects will have a calculateTax()

method, so that method can be used with the variables.

Type Cast

When you use a variable of type Taxable you are asking to use the "taxable" aspect of the object. Many different kinds of objects might be referred to by the variable. (In a larger program there may be Taxable classes that are not Goods.) The compiler can only use the methods it knows that the object must have—those in the interface

However, you can use a type cast to tell the compiler that in a particular statement in the program the variable will refer to an object of a specific class:

```
public static void main ( String[] args )
{
  Taxable item1 = new Book ( "Emma", 24.95, "Austin" );
  System.out.println( "Tax on item 1 "+ item1.calculateTax() );
  ((Book)item1).display();
        }
```

This program is not very sensibly written, since if the variable item1 were of type Book everything would work without the need for a type cast. But in programs with more complicated logic such casts are sometimes needed.

When is a Type Cast Needed?

Now examine the following:

```
public static void main ( String[] args )
{
   Book    book ;
  Taxable tax = new Book ( "Emma", 24.95, "Austin" );
   book = tax;
  book.display();
  System.out.println( "Tax on item 1 "+ book.calculateTax() );
}
```

12.3.3 Implementing multiple interfaces

There are features of interfaces that the example did not show: A class can implement several interfaces:

```
class SomeClass extends SomeParent
   implements InterfaceA, InterfaceB, InterfaceC
{
}
```

Now SomeClass must implement all the methods listed in all the interfaces.

Public Interfaces

However, it is OK if two interfaces ask for the same method. A class that implements both interfaces only needs to provide one complete method definition to satisfy both interfaces.

An interface can be made public. In fact, this is usually what is done. When a class or interface is

public it must be the only public class or interface in the file that contains it. (These notes have been avoiding public classes so that the "copy paste save and run" method can be used with just one file.)

A public interface can be implemented by any class in any file. Many graphical user interface components implement public interfaces. You must use them to work with the GUI features of Java.

Self Learning Exercises

1. What are packages ? what is use of packages ?
2. What do you understand by package access specifier? control mechanism.
3. What is interface? What is use of interface?
4. Is it is necessary to implement all methods in an interface?
5. Which is the default access modifier for an interface method?
6. By default, all program import the java.lang package. True/False
7. User-defined package can also be imported just like the standard packages. True/False

12. 3.4 Extending an Interface

An interface can be an extension of another interface (but **not** an extension of a class):

```
public interface ExciseTaxable extends Taxable
{
 double extraTax = 0.02;
 double calculateExtra();
}
```

A complex hierarchy of interfaces can be constructed using this feature. This is an advanced feature which you will probably not need to use.

12.4 Usage of abstract class

Java allows methods and classes to be declared abstract. An abstract method is not actually implemented in the class. It is merely declared there. The body of the method is then implemented in subclasses of that class. An abstract method must be part of an abstract class. You create abstract classes by adding the keyword abstract after the access specifier, e.g.

```
public abstract class MotorVehicle
```

Abstract classes cannot be instantiated. It is a compile-time error to try something like

```
MotorVehicle m = new MotorVehicle();
```

when MotorVehicle has been declared to be abstract. MotorVehicle is actually a pretty good example of the sort of class that might be abstract. You're unlikely to be interested in a generic motor vehicle. Rather you'll have trucks, motorcycles, cars, go-carts and other subclasses of MotorVehicle, but nothing that is only a MotorVehicle.

An abstract method provides a declaration but no implementation. In other words, it has no method body. Abstract methods can only exist inside abstract classes and interfaces. For example, the MotorVehicle class might have an abstract fuel() method:

public abstract void fuel();

12.5 Summary

- Packages are used for grouping related classes together. They can be used to access related classes as well as to hide the internals of a package from outside programs.
- Packages are declared using the package keyword, which must be located at the beginning of all source files that are to be a part of the same package. Package members can be accessed by pretending the complete package name separated by . to the needed member. The contents of packages can also be accessed by importing the package into a class using the import keyword. Doing so enables the code to specify package members directly without explicitly specifying the complete package name.
- l Package names by convention begin with lowercase letters. This distinguishes them from class names, which by convention begin with uppercase letters. Sun encourages programmers to use a reverse Internet domain name as the top level of a package name to keep package names unique on a global scale.
- Packages can contain sub packages. Java requires that the resulting .class files reside in a directory structure that mirrors the package name hierarchy. By default, the Java interpreter looks for packages where the Java programs were installed. Use the CLASSPATH environment variable to list a set of additional directories to search.
- An interface is a collection of constants and method declarations. The method declarations do not include an implementation (there is no method body.)
- Interfaces are used to define the structure of a set of methods that will be implemented by classes yet to be designed and coded. In other words, the calling arguments and the return value must conform to the definition in the interface

12.6 Glossary

Classpath: An environmental variable which tells the Java virtual machine[1] and Java technology-based applications where to find the class libraries, including user-defined class libraries.

Interface: A Java language keyword that is used to declare a class-like structure. An interface can declare attributes and methods, but it cannot provide method implementations. Any class implementing the interface is required to define the interface's method bodies.

package : A group of *types*. Packages are declared with the package keyword.

12.7 Further Readings

1. Mastering Java2, J2SE1.4 (Paperback) John Zukowski
2. Java: The Complete Reference ,Herbert Schildt Computers - 2005
3. Java 2: The Complete Reference, Herbert Schildt - Computers - 2002
4. Java How to Program, 7/E (Harvey & Paul) Deitel & Associates Publisher: Prentice Hall ,2007

12.8 Answers to Self Learning Exercises

1. The package statement defines a name space in which classes are stored.If you omit the package, the classes are put into the default package. Signature... package pkg;

Use: * It specifies to which package the classes defined in a file belongs to. * Package is both naming and a visibility.

2. public: Anything declared as public can be accessed from anywhere

   private: Anything declared in the private can't be seen outside of its class.

   default: It is visible to subclasses as well as to other classes in the same package

3. It is similar to class which may contain method's signature only but not bodies. Methods declared in interface are abstract methods. We can implement many interfaces on a class which support the multiple inheritance.

4. Yes. All the methods have to be implemented

5. public.

6. True

7. True

12.9 Unit End Questions

1. What is package in java, why it is important.
2. How the package import in java program.
3. What is CLASSPATH explain its use.
4. What is interfaces, why it is used.
5. How will you implement a interface write a code.
6. What is abstract class.
7. Write a code to extend a interface.
8. What is the difference between an Interface and an Abstract class?
9. Which is the default access modifier for an interface method?
10. By default, all program import the java.lang package. True/False
11.. User-defined package can also be imported just like the standard packages. True/False

UNIT - XIII

Exception Handling

Structure of the Unit

13.0 Objective

After completion of this unit, you will learn

- Exception handling basics
- Exception handling through try and catch
- Multiple catches and nested try
- Using throw and throws
- Using built-in exceptions

13.1 Introduction

Object-oriented programming based features of Java have been described in previous units. But one of the important features of Java is robust programming. A program should produce correct output on correct input. But behavior on wrong inputs should be under control. The program should also be able to tolerate intermediate abnormal and unexpected conditions. A program that tolerates abnormal conditions and incorrect inputs is known as robust program. It is possible to control the behavior of program under abnormal conditions and on incorrect inputs in Java through Exception Handling capability. This unit describes robust programming needs, types of exceptions and various constructs of Java needed for exception handling.

13.2 Exception Fundamentals

In traditional programming languages prior to Java, program used to exit on error condition with error code. The error condition could be incorrect input or abnormal program condition during execution. Programmer needs to correct the program manually and then run again. This process is troublesome process and can not be used when program runs under distributed environment. For distributed environment, a robust program continues to run even if it encounters error. However, it may exit on very serious errors. The following subsections describe robust programming needs, exception conditions and mechanism to handle them.

13.2.1 Robust Programming Needs

A reliable program is one, which gives correct output on correct inputs and detects incorrect input conditions. However, a robust program is one that takes care of even unexpected and abnormal conditions. Incorrect input could be a zero value entered in input that is used in denominator i.e. divide by zero condition occurs. An abnormal condition may arise when an effort is made to access an array beyond its size i.e. index value exceeds size of array (*ArrayIndexOutofBoundsException*).

There can be two types of behaviors under this situation

I. Exit program with error condition

II. Report error condition or incorrect input condition and ask for another input or ignore error condition and go ahead

Traditional programming languages used to behave in type – I manner. While behavior of type – II is supported by languages like Java. The key requirement of robust programming is to exhibit expected behavior like type - II on unexpected conditions. The unexpected conditions at runtime are known as exceptions. A good exception handling mechanism is essential for robust programming.

13.2.2 Exception conditions

An exception is an abnormal condition that arises in a code sequence at run-time. In other words, these conditions are observed at run-time and disrupts program running. There can be two types of conditions namely run-time exception and errors. Run-time exceptions are the conditions for which it is possible to handle the condition and continue running if programmer desires, like *divide by zero*. Whereas error conditions can not normally be taken care by programmer e.g. *stack overflow*. This unit deals with run-time exceptions and discusses various ways of handling them. These conditions are described in Java as a number of classes. Table 13.1 shows some commonly used exception conditions.

S.No.	Exception Description	Exception Condition
1.	An input variable value is entered zero or a variable continuously decreasing variable reaches zero that is used in denominator.	Divide by zero
2.	An attempt was made to access an array element beyond the size of array i.e. array index is not within limit	Array index out of bounds
3.	Input entered as string to be used as number. But entered number was not in a format that can be converted into number.	Not in a number format
4.	An attempt was made to store an array element that was not of array type	Improper type array storage
5.	Type casting of a class was to an incompatible type	Invalid cast
6.	One or more actual argument(s) differ in type of formal argument(s)	Illegal argument type
7.	An array index value was specified as negative	Negative array size
8.	A pointer for a class was declared and an attempt was made to access a method or variable of that class	Invalid use of null pointer
9.	Attempt was made to create an object of the class which was not available	Class not found
10.	Attempt was made to access class which was not allowed	Illegal access to a class
11.	Attempt was made to access a non-existing method	Method does not exist
12.	Attempt to create an object of an abstract class or interface	Instantiation not allowed

13.	Attempt to clone an object that does not implement *Cloneable* interface	Clone not supported
14.	One Thread has been interrupted by another thread	Interrupted by thread
15.	An attempt was made to access a non-existing field	Field not found

Table 13.1: Exception Condition Description

13.2.3 Exception handling

Exception handling mechanism describes the behavior of program when exception occurs. There are two ways of handling exceptions : programmer defined exception handling and default exception handling.

Programmer can specify behavior depending on context. Some of the behaviors are given below:

i) Report exception, ignore condition and continue

ii) Report exception and ask user input for further action

iii) Propagate exception to multiple places.

Default exception handler is second type handler that is executed when programmer specifies no behavior for abnormal condition.

Following program illustrates the working of a simple exception handler:

Program 13.1:

Write a program to demonstrate a simple exception handling. The program should try to convert an alphanumeric string to integer and report exception if can not convert.

```
class SimpleException
{
 public static void main(String args[])
  {
   try
     {
     String noStr = "12k";
        int intVal   = Integer.parseInt(noStr);
        System.out.println(" Integer value of String is" + intVal + " .");
     } catch ( NumberFormatException e)
           {
       System.out.println("String is not in format of integer.");
            }
   }
}
```

Output

String is not in format of integer.

Explanation: This program illustrates a simple exception handling concept. A class *SimpleException* contains only *main* method in which some exception is expected to occur. The part of the code, in which exception is expected to occur, is enclosed within ***try*** and ***catch*** keywords. A string type literal *noStr* is declared and initialized with a value "12k", which does not have an equivalent integer value. Another variable *intVal* is declared and initialized with the integer value after conversion of string "12k" into integer. Method ***parseInt*** of class ***Integer*** converts a valid string into equivalent integer and displays its value using *System.out.println* command. However, if string is not in proper format, an exception ***NumberFormatException*** is raised. When this exception occurs, control transfers to catch part of the code and executes it. As the string "12k" can not be converted into an equivalent string, only catch part of the string is executed and output string *String is not in format of integer* is displayed. As above exception handler only reports occurrence of exception, it shows a behavior of first type.

13.3 Exception Classes

Java manages exception handling through a collection of classes through a set of keywords. The classes used for representing exceptions are built-in exception classes. Java provides a number of built-in exception classes in its package ***java.lang***. This package is implicitly imported to all classes. A class ***Throwable*** is at top of the hierarchy. ***Throwable*** class has two subclasses namely ***Exception*** and ***Error***. As discussed earlier that errors can not be handled by programmer, this unit will focus on exception handling related classes. Hierarchy of some exception classes is shown in Figure-13.1. There are more than 60 classes in this hierarchy.

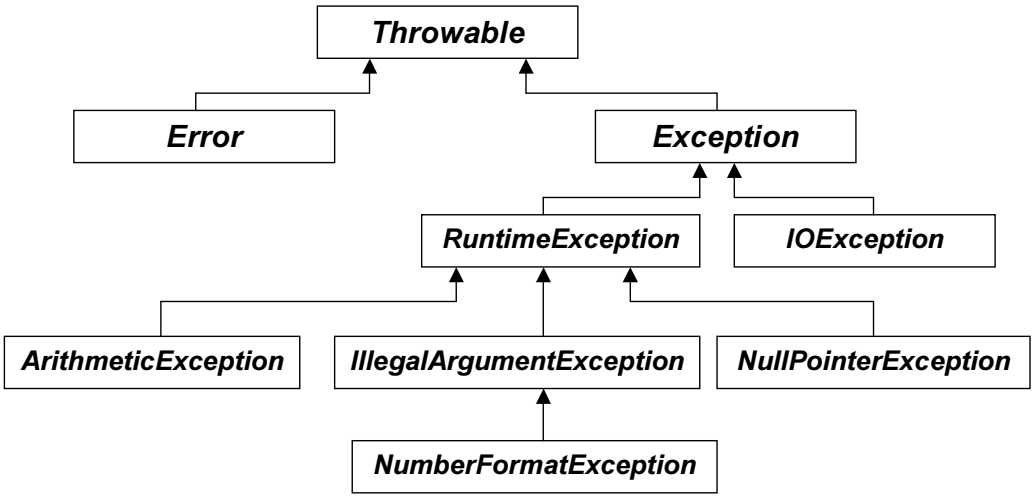


Figure: 13.1 Some classes in Throwable hierarchy.

There can be two types of built-in exception classes namely, checked exceptions and unchecked exceptions. Programmer needs to explicitly mention about handling of checked exceptions in the program where as not necessarily about unchecked exceptions.

13.3.1 Checked Exception

Checked exceptions are either handled by the method that generate them using a pair of keyword ***try*** and ***catch*** or it may be explicitly declared using keyword ***throws***. Hence, compiler

checks these exceptions. Table-13.2 shows some exception conditions and their corresponding checked exception classes.

S.No.	Exception Condition	Checked Exception
1.	Class Not Found	ClassNotFoundException
2.	Clone not supported	CloneNotSupportedException
3.	Illegal access to a class	IllegalAccessException
4.	Instantiation not allowed	InstantiationException
5.	Interrupted by thread	InterruptedException
6.	Field not found	NoSuchFieldException
7.	Method does not exist	NoSuchMethodException

Table 13.2: Checked Exception

13.3.2 UNCHECKED Exception

Unchecked exceptions occur at run time and are not checked by compiler. Programmer is not compelled to write exception handler for them rather it is optional. Table-13.3 shows some common exception conditions and their corresponding unchecked exception classes.

S.No.	Exception Conditions	Unchecked Exception
1.	Divide by zero	ArithmeticException
2.	Array index out of bounds	Array Index Out Of Bounds Exception
3.	Not in a number format	NumberFormatException
4.	Improper type array storage	ArrayStoreException
5.	Invalid cast	ClassCastException
6.	Illegal argument type	IllegalArgumentException
7.	Negative array size	NegativeArraySizeException
8.	Invalid use of null pointer	NullPointerException

Table 13.3: Unchecked Exceptions

13.4 Exception Catching

In previous sections, a number of exceptions have been described. Exception handling is necessary for checked exceptions. For exception handling, first exception is caught and then remedial action is taken on that. These exceptions can be caught using some reserved words of Java. The following subsections explain exception catching.

13.4.1 USING try and catch

A combination of try and catch keywords are used to guard a block of code against the exception and handle the same. The block of code is called try block. Immediately following the try block, include a catch clause that specifies exception type that needs to be caught. The following program illustrates exception handling using try and catch.

Program 13.2:

Write a program in which divide by zero exception is likely to occur. Using try and catch write exception handler for that.

```
class DivZeroException
{
 public static void main(String args[])
  {
    int a, b;
   try
     {
       a = 0;
          b = 61 / a;
          System.out.println(" This may not be printed .");
     } catch ( ArithmeticException e)
            {
         System.out.println("Division by zero exception.");
              }
   }
}
```

Output

Division by zero exception.

Explanation: This program illustrates exception-handling using ***try*** and ***catch*** block. The block generates an exception condition divide by zero that causes exception ***ArithmeticException***. Catch clause displays the exception as shown in output.

13.4.2 Multiple Catches

It is also possible that more than one type of exception may occur within a piece of code. Multiple catch clauses with one try statement are required to take care of that situation. When an exception is thrown, each catch statement is inspected in order and first one whose type matches that of the execution is executed and others are bypassed.

The following program illustrates the use of multiple catch clauses.

Program 13.3:

Write a program in which multiple exceptions are likely to occur. Write exception handler using try and multiple catches.

```
class MultipleCatchClauses
{
 public static void main(String args[])
  {
   try
```

```
    {
     int b = 0;
    System.out.println("b = " + b);
        int a  = 61 / b;
     int c[ ] = { 1 };
         c[7] = 99;
          } catch ( ArrayIndexOutOfBoundsException e)
           {
         System.out.println("Array index out of bounds");
              }
         catch ( ArithmeticException e)
              {
          System.out.println("Division by zero exception");
                }
  }
}
```

Output

b = 0

Division by zero exception.

Explanation: This program illustrates exception handling using multiple ***catch*** clauses. Value of variable b is displayed first and then exception condition divide by zero occurs that causes exception ***ArithmeticException***. This exception is caught by second catch clause that displays the exception as shown in output.

It may also be noted that Exception is a super class of all exceptions. Hence, single catch (Exception e) can catch all types of exceptions.

13.4.3 NESTED try

A try statement can also be within another try statement. This is called nesting of try statements. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch clause for a particular exception, the stack unwound and the next try statement's catch clauses are inspected till a match found or all nested try statements are exhausted. If no catch clause matches, then Java run-time system will handle the exception.

The following program illustrates the use of nested try.

Program 13.4:

Write a program to illustrate nested try statements.

```
class NestedTry
{
 public static void main(String args[])
  {
```

```
    try
      {
       int b = 0;
      System.out.println("b = " + b);
try
{
            int a  = 61 / b;
            int c[ ] = { 1 };
                c[7] = 99;
                } catch ( ArrayIndexOutOfBoundsException e)
                   {
                   System.out.println("Array index out of bounds");
            }
          } catch ( ArithmeticException e)
            {
          System.out.println("Division by zero exception");
              }
    }
 }
```

Output

b = 0

Division by zero exception.

Explanation: This program illustrates exception handling using nested try statement. The value of variable b is displayed first and then exception condition divide by zero occurs in next try statement that causes exception ArithmeticException. The handler for that exception is not found in the catch clause of that try statement. However, the same is found in the catch clause of next try statement. Hence, that catch clause is executed and above output is displays.

13.4.4 USING throw

It is also possible to throw an exception explicitly using throw statement even though it does not occur due to abnormal condition. The flow of execution stops immediately after throw statement and any subsequent statements are not executed. Nearest try block is inspected for catching this exception. If not found then next try statements are inspected like in nested try.

Program 13.5:

Write a program that can throw and handle an exception.

```
class ThrowTest
{
 static void throwTest ()
  {
```

```
    try
      {
       throw new NullPointerExceptio("Test");
       } catch (NullPointerException e)
            {
          System.out.println("Caught in method throwTest");
             throw e;
              }
   }
 public static void main(String args[])
  {
    try
      {
        throwTest();
       } catch ( NullPointerException e)
       {
                System.out.println("Recaught in main");
         }
          }
   }
}
```

Output

Caught in method throwTest

Recaught in main

Explanation: In this program, a class illustrates exception handling using nested try statement. The value of variable b is displayed first and then exception condition divide by zero occurs in next try statement that causes exception ArithmeticException. The handler for that exception is not found in the catch clause of that try statement. However, the same is found in the catch clause of next try statement. Hence, that catch clause is executed and above output is displays.

13.4.5 USING throws

If a method is capable of causing exception that it does not handle, it can specify this behavior so that callers of that method can guard themselves against that exception. This can be done using throws clause in the methods declaration. A throws clause lists the types of exceptions that a method might throw. This is necessary for all checked exceptions.

The following program illustrates throws clause.

Program 13.6:

Write a program in which one method is capable of causing exception but does not handle that.

```
class ThrowsTesting
{
  static void throwsTest () throws IllegaAccessException
    {
    System.out.println("Inside throwtest.");
    throw new IllegalAccessException("Throws Test");
      }
  public static void main(String args[])
  {
    try
      {
       throwsTest( );
       } catch ( IllegalAccessException e)
          {
             System.out.println("Illegal access exception");
                }
          }
    }
}
```

Output

Illegal access exception

Explanation: In this program, class ThrowsTesing contains one method throwsTest() that causes exception IllegalAccessException, but does not handle that. The same is handled by catch clause of calling method.

13.4.6 USING finally

When exceptions are thrown, execution of a method may not take a linear path rather may leave a part of the code or enter into a catch block. Sometimes execution of a part of the code is necessary before exit in any form. For example, if a file is open and because of abnormal exit of code file is not closed. Keyword finally can take care of this contingency situation.

finally creates a block of code that will be executed after a try/catch block execution in case of exception or when no catch statement is available or even if no exception is thrown.

The following program illustrates the use of finally clause.

Program 13.7:

Write a program to illustrate the use of *finally* clause.

```
class FinallyTesting
{
  static void functionOne()
    {
```

```
    try
      {
      System.out.println("Inside functionOne");
          throw new RuntimeException("finally Test");
      } finally
        {
       System.out.println("functionOne's finally");
            }
    }
 static void functionTwo ()
  {
    try
     {
      System.out.println("Inside functionTwo");
      Return;
       } finally
          {
       System.out.println("functionTwo's finally");
            }
    }
 public static void main(String args[ ])
 {
  try
    {
    functionOne( );
     } catch ( Exception e)
       {
          System.out.println("Exception caught");
             }
       }
   functionTwo();
  }
}
```

Output

Inside functionOne

functionOne's finally

Exception caught

Inside functionTwo

functionTwo's finally

Explanation: In this program, class FinallyTesing contains two methods namely functionOne() and functionTwo() in addition to main() method. Both the methods have finally clause, but method functionOne() causes exception. Method function One() outputs Inside functionOne followed by functionOne's finally. Since finally clause of functionOne() again causes exception the same is again caught inside main and displays Exception caught. Method main() again calls functionTwo() which prints Inside functionTwo. Method functionTwo() does not cause any exception, but still finally clause is executed and displays functionTwo's finally.

13.5 Programmer Defined Exception

Most of the common errors are handled by Java's built-in exceptions. However, in some situations it is required to define a new exception. To create a new exception, one needs to define a subclass of ***Exception***. As Exception class is a subclass of ***Throwable*** class, the methods available in ***Throwable*** class can also be used by newly created exception. Programmer can override one or more of these methods if required. Table-13.4 shows a list of some commonly used methods available in ***Throwable*** class.

S.N.	Method	Description
1.	Throwable fillInStackTree()	Returns a Throwable class object containing a completed stack tree. The object can be rethrown
2.	String getLocalizedMessage()	Returns a localized description of the exception
3.	String getMessage()	Returns a description of the exception
4.	void printStackTrace()	Displays the stack trace
5.	String toString()	Returns a String object containing a description of the exception. This method is called by println() when outputting a Throwable class object

Table 13.4: Some commonly used methods defined by Throwable class

The following example declares a new subclass of ***Exception*** and then uses that subclass to signal an abnormal condition in a method. It overrides the *toString()* method and displays the description of the exception using *println()*.

Program 13.8:

Write a program that defines a new exception simply overriding *toString()* method. Also write exception handler that prints different strings on exception and no exception conditions.

```
class NewException extends Exception
{
  private int value;
  NewException (int a)
   {
```

```
    value = a;
   }
 public String toString()
 {
  return "NewException value is " + value;
  }
}
class ExceptionHandling
{
 static void showValue ( int a ) throws NewException
 {
  System.out.println( "Called showValue(" + a + ")");
  if (a > 10)
   throw new NewException(a);
  System.out.println("Normal exit");
  }

public static void main(String args[])
  {
   try
      {
      showValue(2);
      showValue(15);
      } catch ( NewException e)
            {
          System.out.println("Caught  " + e);
              }
   }
 }
```

Output

Called showValue(2)

Normal exit

Called showValue(15)

Caught NewException value is 15

Explanation: In this program, a new exception is defined and tested. Class NewException is created as a subclass of Exception. The class contains one constructor that simply assigns an

integer value passed through argument to internal variable value and overrides toString() method.

Class ExceptionHandling declares two methods namely showValue() and main(). Method showValue() expects that an programmer defined exception NewException can occur during its execution, but it should be handled by calling method. The method takes an integer value in input and throws an exception if integer value is more than 10. To test this exception, main() method calls showValue() method two times. First time input value 2 is passed which does not cause exception and another time with value 15 that causes exception. The same is verified through output.

Self Learning Exercises

1. What is the difference between 'throw' and 'throws' ?And it's application?
2. What is the difference between 'Exception' and 'error' in java?
3. Can we have catch block with out try block? If so when?
4. What will happen to the Exception object after exception handling?

13.6 Summary

- Robust program is one that continues to run even on incorrect input and abnormal condition during program execution. It is possible using exception handler.
- There abnormal condition like stack overflow are called Errors, which can not normally be handled by programmer.
- Programmer can control the flow of the program for some abnormal condition like divide by zero. Such abnormal conditions are called Exception.
- All classes related to abnormal condition inherit class Throwable and nearly 60 classes are derived from this class in hierarchy.
- Whenever exception occurs, program does not exit rather goes to a sequence of code defined for exception condition.
- There are two types of exceptions namely checked and unchecked exceptions. Checked exceptions are checked at compilation time, but unchecked exceptions are not checked at compilation time.
- In a method, where checked exception is expected to occur, code start with a keyword try followed by braces and code within that and then a keyword catch followed by code to be executed when exception occurs.
- If a method can generate checked exception but expects calling method to handle then it has to declare the same in the beginning using throws keyword.
- Unchecked exceptions need not be included within try and catch or throws keywords.
- There can be multiple catch statements for different exceptions.
- For remaining exceptions, there can be finally statements after one or more exceptions.
- It is also possible to throw exception explicitly using throw keyword.
- It is also possible to create programmer-defined exceptions for application specific conditions. This exception can also override methods defined in class Throwable.

13.7 Glossary

Exception :An event during program execution that prevents the program from continuing normally; generally, an error. The Java programming language supports exceptions with the try, catch, and throw keywords.
exception handler :A block of code that reacts to a specific type of *exception*. If the exception is for an error that the program can recover from, the program can resume executing after the exception handler has executed.
throw:A Java language keyword that causes an exception or other Throwable object to be passed up the call stack, redirecting program execution to the first catch block that handles the Throwable.

13.8 Further Readings

1. Java, UPTEC & Excel Books
2. Programming With Java, E. balaguruswamy, TMH
3. Complete Java Book, Madhav Oke, BPB
4. Complete Java 2 Certification Study Guide, BPB
5. Java2: The Complete Reference, Patrick Naughton & Herbert Schildt
6. Java & XML, O' Reilly
7. Who's Afraid of Java? Steve Heller, AP Professional
8. Teach Yourself Java 1.1 Programming in 24 Hours, Rogers Cadenhead, BPB
9. Java Now!, Kris Jamsa, Jamsa Press
10. The Java Language Specification, James Gosling, Bill Joy, Guy Steele, Addison-Wesley

13.9 Answers to self learning exercises

1.Exceptions that are thrown by java runtime systems can be handled by Try and catch blocks. With throw exception we can handle the exceptions thrown by the program itself. If a method is capable of causing an exception that it does not handle, it must specify this behavior so the callers of the method can guard against that exception.

2.Exception and Error are the subclasses of the Throwable class. Exception class is used for exceptional conditions that user program should can subclass to create our own custom exception. Error defines exceptions that are not excepted to be caught by you program. Example is Stack Overflow.

3.No. Try/Catch or Try/finally form a unit.

4.It will go for Garbage Collector. And frees the memory.

13.10 Unit End Questions

1) Distinguish between exception and error. Describe the hierarchy of subclasses of exception related classes.
2) Distinguish between checked and unchecked exceptions. Give three examples of each.
3) Write a program to demonstrate that explains nested try statements.
4) Explain the use of Throw statement in exception handling.

UNIT - XIV

Multithreading

Structure of the Unit

14.0 Objectives

After reading this unit, you should be able to:

- Explain the meaning, purpose and use of Multithreading
- Describe the Thread Life Cycle (TLC)
- Identify the major Thread Priorities
- Creating and executing threads
- Explain the use of Thread Synchronization
- Describe the concepts of Thread Class and Runnable Interface
- Discuss the concepts of Interthread Communication and Deadlock
- Describe the Suspending, Resuming and Stopping the Thread

14.1 Introduction

Well whatever you have learnt about Java in he previous units is only a part of it. We know that Java was designed to meet the real world requirement of creating interactive, networked programs. To accomplish this, Java supports multithreaded programming, which allows us to write programs that do many things simultaneously. The Java runtime system comes with an elegant yet sophisticated solution for mutiprocess synchronization that enables you to construct smoothly running interactive systems.

14.2 An Introduction of Multithreading:

Now, we are much familiar with the modern operating systems such as Windows 95/98 may recognize which users can execute programs simultaneously. This ability is known as multitasking. In system's terminology, *it is also called multithreading.*

Multithreading is a conceptual programming paradigm where a program is divided into two or more subprograms, which can be implemented at the same time another parallel. For example, one subprogram can display an animation on the screen while another may build the next animation to be displayed. This is something similar to dividing a task into subtasks and assigning them to different people for execution independently and simultaneously.

Mostly in the computers, we have only a single processor and therefore, in reality, the processor is doing only one thing at a time. However, the processor switches between the processes so fast that it appears to human beings that all of them are being done simultaneously.

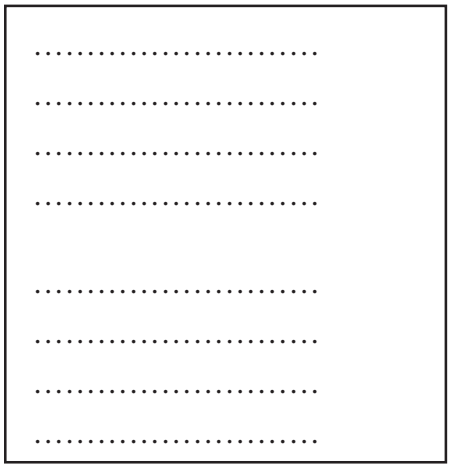
We have seen and discussed that the Java programs contain only a single sequential flow of control. This what happens when we execute a normal program. The program begins, runs through a sequence of execution, and finally ends. At any given point of time, there is only one statement under execution.

A thread is similar to a program that has a single flow of control. It has a beginning, a body, and an end and it execute commands sequentially.

Generally all main programs in our earlier examples can be called single-threaded programs.

Every program will have at least one thread as represented in the Figure 14.1 is given below:

```
Class ABC
{
.....                                    Beginning
```



Single-threaded body

of Execution

End

}

Figure 14.1 : A single threaded program

An important feature of Java is its support for multithreading. That is, Java enables us to use multiple flow of control is developing programs. Each flow of control can be considered as a separate small program or module known as thread which runs in parallel to others as represented in the Figure 14.2 given below:

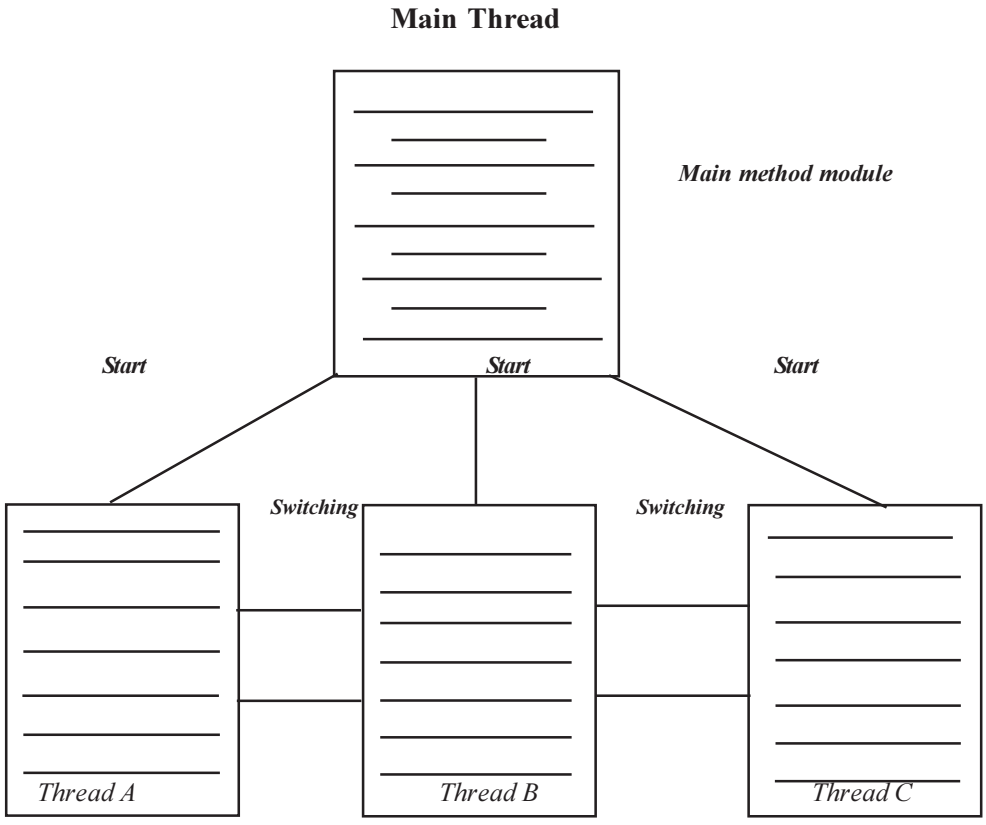


Figure 14.2 A Multithreaded Program

A program that contain multiple flows of control is known as *multithreaded program*. Here we have discussed the diagram14.2 that represents a Java program with four threads, one main and three others. The main thread is actually the main method module, which is designed to create and start the other three threads, such as A, B and C.

Here, Once initiated by the main thread, the threads A, B and C run concurrently and share the resources jointly. It is like people living in joint families and sharing certain resources among all of them. The ability of a language to support multithreading is referred to as concurrency.

Because we know that the threads in Java are subprograms of a main application program and share the same memory space, they are named as *lightweight-threads or lightweight processes.*

It is keep in your mind that threads running in parallell does not really mean that they actually run at the same time. Since all the threads are running on a single processor, the flow of execution is shared between the threads. The Java interpreter handles the switching of control between the threads in such a way that it appears they are running concurrently.

Therefore we can say the multithreading is a powerful programming tool that makes Java uniquely different from its fellow programming languages. Mulithreading is very useful in a number of ways. Mulithreading enables programmers or users to accomplish multiple things at one time. Which can divide a long program into threads and execute them in parallel.

For example: We can send tasks such as printing into the background and continue to perform some other task in the foreground. This approach would considerably improve the speed of our programs.

So that the Threads are extensively used in Java-enabled browsers such as HotJava. These browsers can download a file to the local computer, display a Web page in the window, output another Web page to a printer and so on.When we are working on any application that requires two or more things to be done at the same time is probably a best one for use of threads.

14.3 Thread Life Cycle (TLC)

The Thread Life Cycle consist of the following states:

14.3.1 Newborn State

14.3.2 Runnable State

14.3.3 Running State

14.3.4 Blocked State

14.3.5 Dead State

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in the Figure 14.3 given below:

Here we are discussing the Life Cycle of a Thread with diagrammatic representation that consists five states. These states are following: -

(i) Newborn State

(ii) Runnable State

(iii) Running State

(iv) Blocked State

(v) Dead State

A thread is always in one of these five states. It can move from one state to another via a variety of ways as shown in the Figure 14.3 given below:

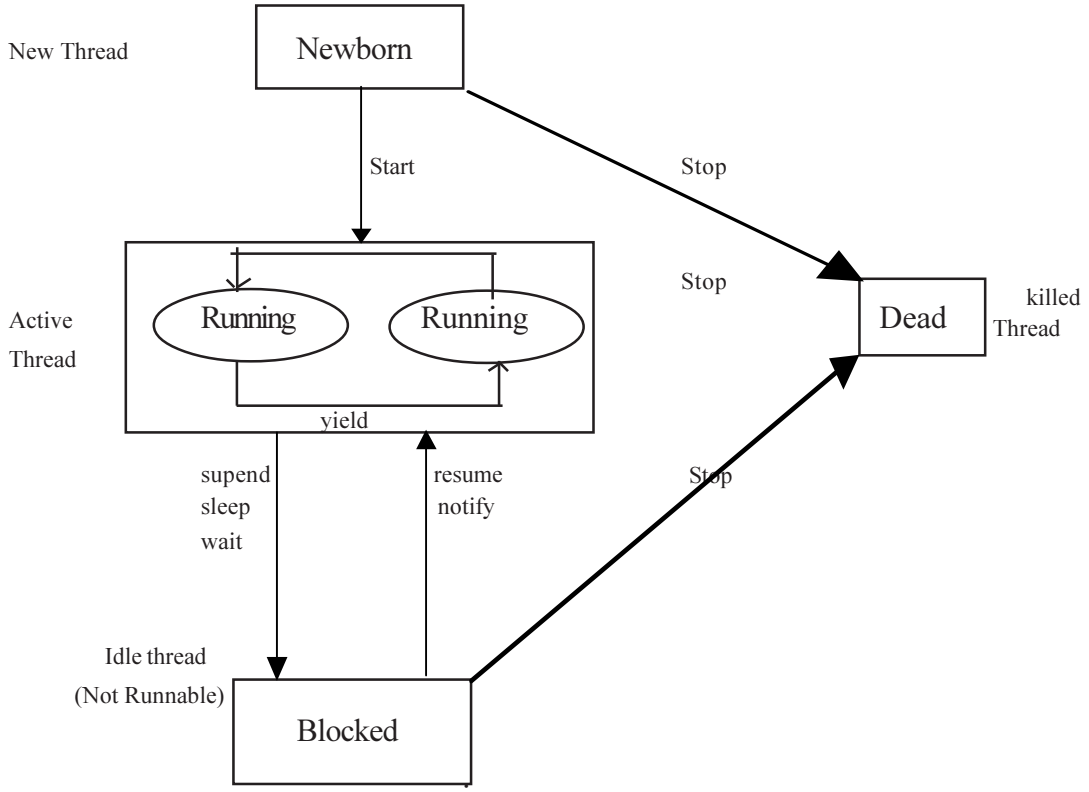


Figure 14.3 A Life Cycle of a Thread

14.3.1 Newborn State:

When we create a thread object, the thread is born and it is said to be in newborn state. The thread is not yet scheduled for running.

At this state, we can do only one of the following things with it:

(i) Schedule it for running using ***start()*** method.

(ii) Kill it using ***stop()*** method.

If the thread has been scheduled then it will move to the runnable state.

That is represented in theFigure14.4 given below:

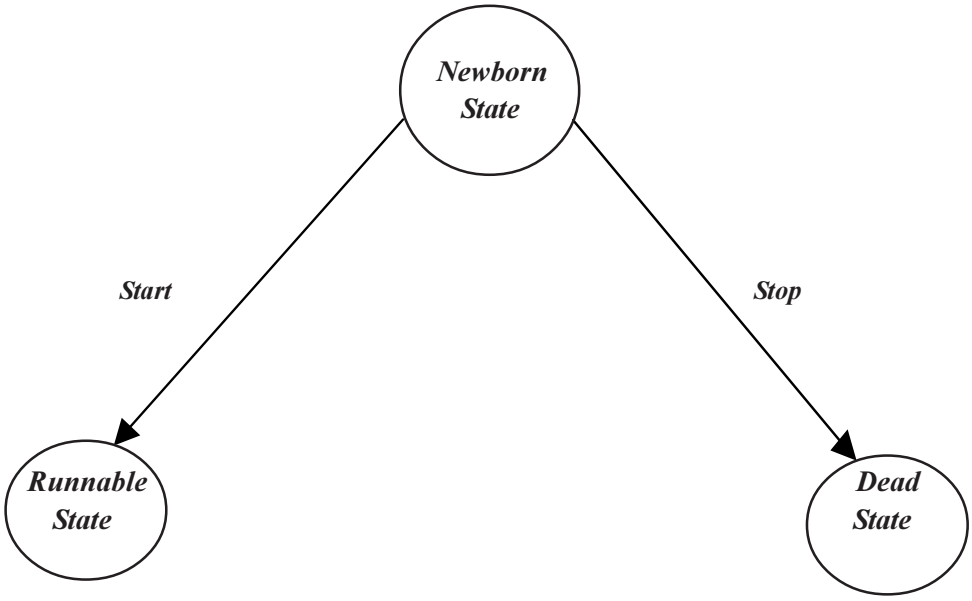


Figure14.4 ***: Scheduling a Newborn Thread***

14.3.2 Runnable State:

The runnable state means that the thread is ready for execution and is waiting for the availability of the processor.

That means the thread has joined the queue of threads which are waiting for execution. If all threads have equal priority, then they are given time slots for execution in round-robin scheduling i.e. first - come first-serve(FCFS) scheduling. The thread leaves control joins the queue at the end and again waits for its turn. *This process of assigning time to threads is called time-slicing.*

Whatever we want a thread to leave control to another thread of equal priority before its turn comes, we can do needful things by using the ***yield()*** method.

The runnable state of the thread is represented in the Figure 14.5 given below:

/* A diagrammatic representation of the runnable state of a thread */

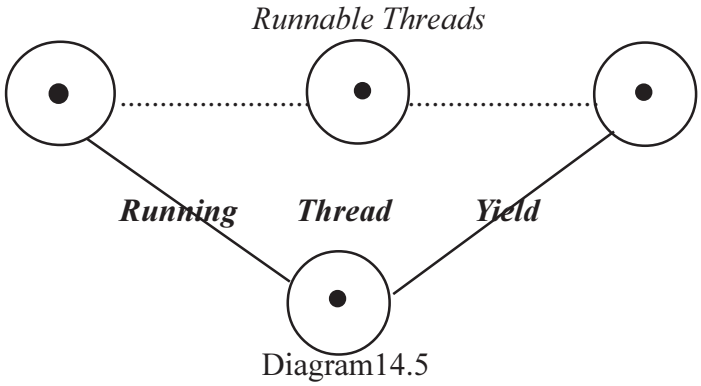


Diagram14.5

14.3.3 Running State:

Running means that the processor has given its time to the thread for its execution. The thread runs until it leaves control on its own or it is preempted by a higher priority thread. A running thread may leave its control in one of the following situations.

(i) It has been suspended using ***suspend()*** method. A suspended thread can be revived by using the ***resume()*** method. This approach is very useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it. This is shown in the diagram 14.6 as follows:

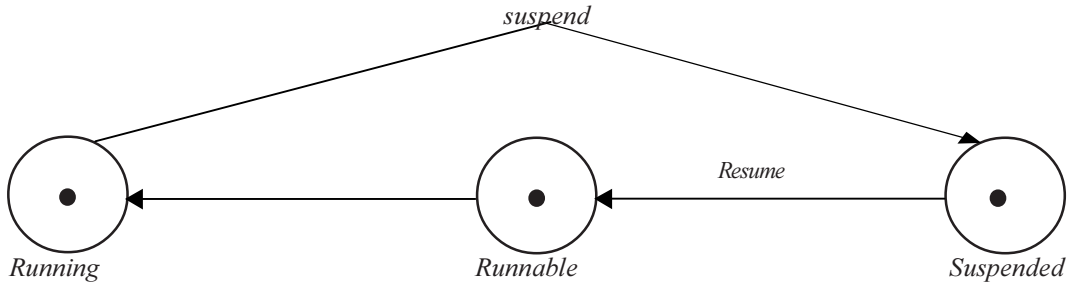
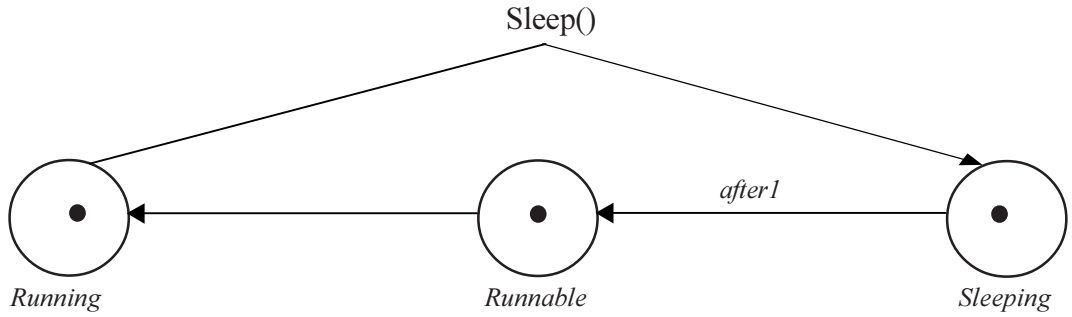


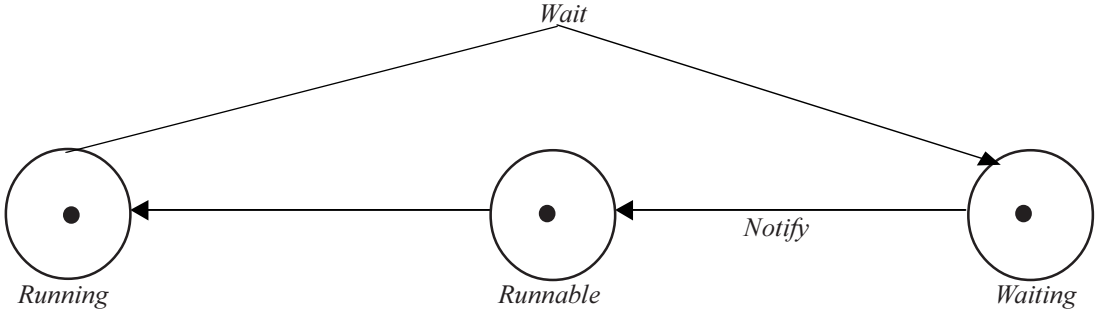
Diagram14.6

(ii) It has been made to sleep. We can put a thread to sleep for a specified time period using the method *sleep (time)* where time is in milliseconds. This means that the thread is out of the queue during this period. The thread re-enters the runnable state as soon as this time period is elapsed. This is shown in the diagram 14.7 as follows:



/* Diagram14.7 */

(iii) It has been told to wait until some event occurs. This is done using the ***wait()*** method. The thread can be run again using the ***notify()*** method. This is shown in the Figure 14.8 as follows:



Self Learning Exercises -I

Before you proceed to the next section, answers the following questions:

Q1. What do you understand by thread?

Q2. What is Multithreading?

Q3. What are the different states of thread life cycle?

Q4. Define the concept of time slicing.

Q5. Explain the following terms:

(i) Light weighted thread

(ii) Multithreaded program

14.4 Thread Priorities

Each thread is assigned a priority in Java, which affects the order in which it is scheduled for running. The threads that we have discussed so far are of the same priority. The thread of the same priority are given equal treatment by the Java scheduler and therefore, they share the processor on a first-come, first-serve(FCFS) basis.

The Thread class in Java supports the following priorities in many ways:

(i) MIN_PRIORITY

(ii) NORM_PRIORITY

(iii) MAX_PRIORITY

Java permits us to set the priority of a thread by using the *setPriority( ) method* as follows:

```
ThreadName.setPriority(intNumber) ;
```

The *intNumber* is an integer value to which the thread's priority is set. The Thread class defines several priority constants :

```
MIN_PRIORITY    = 1
NORM_PRIORITY   = 5
MAX_PRIORITY    = 10
```

The *intNumber* may assume one of these constants or any value between 1 and 10. It is noted that the default setting is NORM_PRIORITY.

Most user-level processes should use NORM_PRIORITY, plus or minus 1. Background tasks such as network I/O and screen repainting should use a value very near to the lower limit. We should be very cautious when trying to use very high priority values. This may defeat the very purpose of using multithreads.

By assigning priorities to threads, we can ensure that they are given the attention they deserve.

For example, we may need to answer an input as quickly as possible. Whenever multiple threads are ready for execution, the Java system chooses the highest priority thread and executes it. For a thread of lower priority to gain control, one of the following things should happen:

(a) It stops running at the end of ***run()***.

(b) It is made to sleep using ***sleep()***.

(c) It is told to wait using ***wait()***.

However, if another thread of a higher priority comes along, the currently running thread will be preempted by the incoming thread thus forcing the current thread to move to the runnable state. Remember that the highest priority thread always preempts any lower priority threads.

Here, we discuss an example program is given below and its output tells us about the effect of assigning higher priority to a thread. It is also noted that although the thread A started first, the higher priority thread B has preempted it and started printing the output first.

Immediately, the thread C that has been assigned the highest priority takes control over the other to threads. The thread A is the last to complete.

An Example Program which demonstrates the Use of priority in threads:

```
class A extends Thread
{
        public void run ( )
        {
                System.out.println ("threadA started") ;
                for (int i = 1 ; i<=4; i + +)
                {
                        System.out.println ("\tFrom Thread A  : i = " + i) ;
                }
                System.out.println ("Exit from A ") ;
        }
}
class B extends Thread
{
        public void run ( )
        {
                System.out.println ("threadB started") ;
                for (int j =1; j<=4; j + +)
                {
                        System.out.println ("\tFrom Thread B  : j = " + j) ;
                }
                System.out.println ("Exit from B ") ;
        }
}
class C extends Thread
{
        public void run ( )
        {
                System.out.println ("threadC started") ;
```

```
            for (int k = 1 ; k<=4; k + +)
            {
                  System.out.println ("\tFrom Thread C  : k = " + k) ;
            }
            System.out.println ("Exit from C ") ;
      }
}
class ThreadPriority
{
      public static void main (String args [ ] )
      {
            A threadA = new A ( ) ;
            B threadB = new B ( ) ;
            C threadC = new C ( ) ;
            threadC. setPriority (Thread. MAX_PRIORITY) ;
            threadB. setPriority (threadA. getPriority ( ) + 1) ;
            threadA. setPriority (Thread. MIN_PRIORITY) ;
            System.out.println ("Start thread A")
            thread A.start ( ) ;
            System. out. println ("Start thread B") ;
            thread B. start ( );
            System. out.println ("Start thread C") ;
            thread C. start ( ) ;
            System. out. println ("End of main thread") ;
      }
}
```

A Sample Output of the above Program would be:

```
      Start thread A
      Start thread B
      Start thread C
      threadB started
                  From Thread   B     :     j     = 1
                  From Thread   B     :     j     = 2
      threadC started
                  From Thread   C     :     k     = 1
                  From Thread   C     :     k     = 2
```

```
                From Thread  C   :   k   = 3
                From Thread  C   :   k   = 4
Exit from C
End of main thread
                From Thread  B   :   j   = 3
                From Thread  B   :   j   = 4
Exit from B
threadA started
                From Thread  A   :   i   = 1
                From Thread  A   :   i   = 2
                From Thread  A   :   i   = 3
                From Thread  A   :   i   = 4
Exit from A
```

14.5 Creation & Execution of a Thread

The Creation of threads in Java is very simple. Threads are implemented in the form of objects which contain a method called ***run()*** method. This method is the heart and soul of any thread. It makes up the entire body of a thread and is the only method in which the thread's behavior can be implemented.

A typical *run()* method appears as :

```
public void run()
{
    .....
    .....      // statements for implementing thread
    }
```

The *run()* method should be invoked by an object of the concerned thread. It is achieved by creating the thread and initiating it with the help of another thread method called *start()*.

A new thread can be created in two ways:-

(i) *By creating a thread class:* Define a class that extends Thread class and override its run() method with the code required by the thread.

(ii) *By converting a class to a thread:* Define a class that implements Runnable interface. The Runnable interface has only one method, run( ), that is to be defined in the method with the code to be executed by the thread.

The approach to be used depends on what the class we are creating requires. If it requires to extend another class, then we have no choice but to implement the Runnable interface, since Java classes cannot have two superclasses.

14.5.1 Extending Thread Class:

Here we can create our class runnable as a thread by extending the class *java.lang.Thread*. It provides us access to all the thread methods directly.

This concept includes the following steps:

Step-I. Declare the class as extending the Thread class.

Step-II. Implement the run( ) method that is responsible for executing the sequence of code that thread will execute.

Step-III. Create a thread object and call the start( ) method to initiate the thread execution.

Declaring the class:

The Thread class can be extended as follows.

```
class Mythread extends Thread
    {
    .....
    .....
    .....
    }
```

Now we have a new type of thread *MyThread.*

Implementing the run( )method:

The *run()* method has been inherited by the class MyThread. We have to override this method inorder to implement the code to be executed by our thread.

The basic implementation of run() method looks like as:

```
public void run ( )
{
    .....
    .....                          // Thread code here
    .....
  }
```

When we start the new thread, Java calls the thread's *run() method,* so it is the *run( )* where all the action takes place.

Starting New Thread:

When we want to create and run an instance of our thread class, then we should write the following:

```
MyThreadaThread = new MyThread ( ) ;
aThread. start ( ) ;                          // invokes run( ) method
```

The first line instantiates a new object of class *MyThread.*

It is noted that this statement just creates the object. The thread that will run this object is not yet running. The thread is in a newborn state.

The second line calls the *start( )* method causing the thread to move into the runnable state. Then, the Java runtime will schedule the thread to run by invoking its *run( )* method. Now, the thread is said to be in the running state.

Here we will discuss an example program which creates threads using the thread class:

An Example Program of Using the Thread Class:

The Example program is given below that demonstrates the proper use of Thread class for creating and running threads in an application.

This program creates three threads A, B, and C for undertaking three different tasks. The main method in the *ThreadTest* class also constitutes another thread which we may call the "main thread".

The main thread dies at the end of its main method. However, before it dies, it creates and starts all the three threads A, B, and C. Please note this statements like:

```
new A( ).start ( ) ;
```

in the main thread. This is just a compact way of starting a thread. This is equivalent to:

```
A threadA = new A ( ) ;
threadA. start ( ) ;
```

Immediately after the thread A is started, there will be two threads running in the program: First is the main thread and second is : the thread A. The *start( )* method returns back to the main thread immediately after invoking the *run( )* method, thus allowing the main thread to start the thread B.

```
// An example program which creates threads using thread class
class A extends thread
{
        public void run ( )
        {
                for  (int i=1; i<=5; i++)
                        {
                                System. out println (" \tFrom Thread A : i = " + i) ;
                        }
                System out. println ("Exit from A ") ;
    }
}
class B extends Thread
{
        public void run ( )
        {
                for (int j=1; j<=5; j++)
                        {
                                System. out. println (" \tFrom Thread B : j = " + j) ;
                                }
                        System. out. println ("Exit from B ")'
        }
}
```

```
class C extends Thread
{
        public void run ( )
        {
            for (int k=1; k<=5; k++)
        {
            System.out.println (" \tFrom Thread C : k = " + k) ;
                }
                System.out.println ("Exit from C " ) ;
        }
}
class ThreadTest
{
        public static void main (String args [ ] )
        {
                new A ( ). start ( ) ;
                new B ( ). start ( ) ;
                new C ( ). start ( ) ;
        }
}
```

A Sample Output of the above program would be:

First run

```
        From Thread   A     :     i     =     1
        From Thread   A     :     i     =     2
        From Thread   B     :     j     =     1
        From Thread   B     :     j     =     2
        From Thread   C     :     k     =     1
        From Thread   C     :     k     =     2
        From Thread   A     :     i     =     3
        From Thread   A     :     i     =     4
        From Thread   B     :     j     =     3
        From Thread   B     :     j     =     4
        From Thread   C     :     k     =     3
        From Thread   C     :     k     =     4
        From Thread   A     :     i     =     5
Exit from A
```

```
        From Thread  B  :  j  =  5
Exit from B
        From Thread  C  :  k  =  5
Exit from C
Second run
        From Thread  A  :  i  =  1
        From Thread  A  :  i  =  2
        From Thread  C  :  k  =  1
        From Thread  C  :  k  =  2
        From Thread  A  :  i  =  3
        From Thread  A  :  i  =  4
        From Thread  B  :  j  =  1
        From Thread  B  :  j  =  2
        From Thread  C  :  k  =  3
        From Thread  C  :  k  =  4
        From Thread  A  :  i  =  5
Exit from A
        From Thread  B  :  k  =  4
        From Thread  B  :  j  =  5
        From Thread  C  :  k  =  5
Exit from C
        From Thread  B  :  j  =  5
Exit from B
```

Similarly, it starts C thread. By the time the main thread has reached the end of its main method, there are a total of four separate threads running in parallel.

We have simply initiated three new threads and started them. We did not hold on to them any further. they are running concurrently on their own.

It is noted that the output from the threads are not specially sequential. They do not follow any specific order. They are running independently of one another and each executes whenever it has a chance. remember, once the threads are started, we cannot decide with certainty the order in which they may execute statements. It is also noted here a second run has a different output sequence.

14.5.2 Implementing the Runnable Interface

We have discussed earlier that *we can create threads in two ways:*

First is ***By using the extended Thread class*** and Second is ***By implementing the Runnable interface***.

We have already learnt in detail how the Thread class is used for creating and running threads.

In this section, we will discuss about the use of the Runnable interface to implement threads.

The Runnable interface declares the *run( ) method* that is required for implementing threads in our programs.

For this purpose, we should use the following steps given below:

Step-I. Declare the class as implementing the Runnable interface.

Step-II. Implement the run ( ) method.

Step-III. Create a thread by defining an object that is instantiated from this "runnable" class as the target of the thread.

Step-IV. Call the thread's start ( ) method to run the thread.

Here we will explain this concept with the help of an example program.

This example program is given below represents the implementation of the above steps.

```
//An Example Program of Using the Runnable Interface:
class X implements Runnable                                    //step1
{
public void run()                                              //step2
{
for(int i=1; i<=10;i++)
{
System.out.println("\t ThreadX : " +i);
}
System.out.println("End of ThreadX ");
}
}
class RunnableTest
{
public static void main(String args[])
{
X runnable=new X();
Thread threadX=new Thread(runnabe);          //step3
threadX.start ();                                    //step4
System.out.println("End of Main  Thread");
}
}
```

The Sample output of the above program is:

```
End of main Thread
        ThreadX     :     1
        ThreadX     :     2
        ThreadX     :     3
```

```
ThreadX     :     4
ThreadX     :     5
ThreadX     :     6
ThreadX     :     7
ThreadX     :     8
ThreadX     :     9
ThreadX     :     10
End of  ThreadX
```

Here in main method, we first create an instance of X and then pass this instance as the initial value of the object thread X (i.e. an object of Thread Class). Whenever, the new thread thread X starts up, its *run()* method calls the *run()* method of the target object supplied to it.

Here, the target object is runnable. If the direct reference to the thread thread X is not required then we can use a shortcut as shown below:

```
new Thread (newX( ) ). start ( ) ;
```

14.6 Thread Synchronization

We know that the threads which use their own data and methods provided inside their run ( ) methods. What happens when they try to use data and methods outside themselves? In this situation, they may compete for the same resources and may lead to serious problems.

For example, one thread may try to read a record from a file while another is still writing to the same file. ***Depending on the situation, we may get strange results. Java enables us to overcome this problem using a technique known as synchronization.***

The keyword ***synchronized*** is well used in Java language for this purpose. The keyword ***synchronized*** helps to solve such problems by keeping a watch on such locations.

For example, the method that will read information from a file and the method that will update the same file may be declared as synchronized. Example is given as:

```
synchronized void update ( )
    {
        .....
        .....                    //code here is synchronized
        .....
    }
```

If we have to declare a method synchronized, Java creates a monitor: and hands it over to the thread that calls the method first time. As long as the thread holds the monitor, no other thread can enter the synchronized section of code. *A monitor is like a key and the thread that holds the key can only open the lock.*

It is also possible to mark a block of code as synchronized as shown below:

```
synchronized (lock - object)
{
```

```
                    .....                    // code here is synchronized
                    .....
    }
```

Whenever a thread has completed its work of using synchronized method (or block of code), it will hand over the monitor to the next thread that is ready to use the same resource.

There is an interesting situation occur when two or more threads are waiting to gain control of a resource. Due to some reason, the condition on which the waiting threads rely on to gain control does not happen. This situation is known as deadlock.

For example, assume that the thread A must access Method1 before it can release Method2, but the thread B cannot release Method1 until it gets hold of Method2. Because these are mutually exclusive conditions, a deadlock occurs.

The given code demonstrates like this:

```
    Thread A
        synchronized method2 ( )
        {
                synchronized method1 ( )
                {
                                .....
                                .....
                }
        }
Thread B
        synchronized method1 ( )
        {
            synchronized method2 ( )

            {
                                .....
                                .....
            }
        }
```

Self Learning Exercises-II

Before you proceed to the next section, answers the following questions:

Q1. What are the different ways of creation of a thread?

Q2. Explain the Thread Priority? Give their name.

Q3. Discuss the concept of Thread Synchronization in brief.

If your answers are correct, then proceed to the next topics:

14.7 Messaging

When we have to divide our program into separate threads, then we need to define how they will communicate with each other. When programming with other languages, we must depend on the operating system to establish communication between threads. Exactly this adds overhead.

By contrast, Java provides a clean, low-cast way for two or more threads to talk to each other, via calls to predefined method that all object have. *Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out.*

14.8 The Thread Class and the Runnable Interface

The multithreading system of Java is built upon the Thread class, its methods, and its companion interface, Runnable.

Thread encapsulates a thread of execution. Because we can't directly refer to the ethereal state of a running thread, we will deal with it through its proxy, the Thread instance that spawned it.

To create a new thread, our program will either extend Thread or implement the Runnable interface.

The Thread class defines various methods which are helpful to manage threads.

Here, some of important are given below:

Method	Meaning
getName	obtain a thread's name.
getPriority	obtain a thread's priority.
isAlive	Determine it a thread is still running.
join	Wait for a thread to terminate.
run	Entry point for the thread.
sleep	Suspend a thread for a period of time.
start	Start a thread by calling its run method

14.9 Interthread Communication

The previous examples unconditionally blocked other threads from asynchronous access to certain methods. This use of the implicit monitors in Java objects is powerful, but you can achieve a more subtle level of control through interprocess communication. As you will see, this is especially easy in Java.

We have discussed in previous unit, multithreading replaces event loop programming by dividing your tasks into discrete and logical units. Threads also provide a secondary benefit: they do away with polling.

Polling is usually implemented by a loop that is used to check some condition repeatedly. Once the condition is true, appropriate action is taken. This wastes CPU time.

For example, Here we consider the *classic queuing problem*, where a single thread is producing some data and another is consuming it.

To make the problem more interesting, suppose that the producer has to wait until the consumer is finished before it generates more data.

We can say it, In a polling system, the consumer would waste many CPU cycles while it waited for the producer to produce. Once the producer was finished, it will start polling, wasting more CPU cycle waiting for the consumer to finish, and so on. Clearly, this situation is undesrirable.

If we want to avoid polling, Java supports an elegant interprocess communication mechanism via the wait( ), notify( ) and notifyAll( ) method. These methods are implemented as final methods in Object, so all classes have them.

All three methods can be called only from within a synchronized method. Although conceptually advanced from a computer perspective, the rules for using these methods are actually quite simple. These methods are :

* wait( ) tells the calling thread to give up the monitor and go to sleep until someother thread enters the same monitor and calls notify( ).
* notify( ) wakes up the first thread that called wait( ) on the same object.
* notifyAll( ) wakes up all the threads that called wait( ) on the same object. The highest priority thread will run first.

These methods are declared within Object, as shown here:

final void wait( ) throws InterruptedException

final void notify( )

final void notifyAll( )

Additional forms of wait( ) exist that allow you to specify a period of time to wait.

The following sample program incorrectly implements a simple form of the producer/consumer problem.

It consists of four classes: Q, the queue that we are trying to synchronize; Producer, the threaded object that is producing queue entries;

Consumer, the threaded object that is consuming queue entries; and PC, the tiny class that creates the single Q, Producer, and Consumer.

```
// An incorrect implementation of a producer and consumer problem
class Q {
  int n ;
  synchronized int get ( )
{
        System.out.println("Got: " + n);
        return n;
   }
  synchronized void put ( int n) {
        this .n = n;
        System.out.println("Put: " + n);
        return n;
   }
}
```

```
        class producer implements Runnable {
          Q q;
          Producer(Q q) {
                this.q = q;
                new Thread (this, " Producer") . start ( ) ;
        }
public void  run ( ) {
                int i = o ;
                while (true)       {
                        q.put (i++) ;
                }
        }
}
class Consumer implements Runnable {
        Q q ;
        Consumer ( Q q ) {
                this. q = q;
                new Thread (this, "Consumer"). start ( ) ;
        }
        public void run ( ) {
                while (true) {
                q. get ( ) ;
                }
        }
}
class PC {
        public static void main (String args [] ) {
        Q q = new Q ( ) ;
        new Producer (q) ;
        new Consumer (q) ;
        System. out println ("Press Control-C to stop.") ;
        }
}
```

Although the put( ) and get( ) methods on Q are synchronized, nothing stops the producer from overrunning the consumer, nor will anything stop the consumer from consuming the same queue value twice. Thus, you get the erroneous output shown here (the exact output will vary with processor speed and task load):

```
Put:    1
Got:    1
Got:    1
Got:    1
Got:    1
Got:    1
Put:    2
Put:    3
Put:    4
Put:    5
Put:    6
Put:    7
Got:    7
```

As we can see, after the producer put 1, the consumer started and got the same 1 five times in a row. Then, the producer resumed and produced 2 through 7 without letting the consumer have a chance to consume them.

The proper way to write this program in Java is to use wait( ) and notify( ) to signal in both directions, as shown here:

// An correct implementation of a producer and consumer problem

```
class Q {
  int n ;
        boolean valueSet = false;
synchronized int get ( )        {
    if ( ! valueSet)
      try   {
        wait( ) ;
      }     catch(InterruptedException e)           {
        System.out.println("InterruptedException caught") ;
      }
        System.out.println("Got : " + n) ;
        valueSet = false;
        notify( ) ;
        return n;
    }
synchronized void put (int n)   {
    if (valueSet)
      try   {
```

```
                wait( ) ;
        }       catch(InterruptedException e)           {
                System.out.println("InterruptedException caught") ;
            }
                this.n = n;
                valueSet = true;
                System.out.println("Put : " + n) ;
                notify( ) ;
        }
  }
        class producer implements Runnable {
          Q q;
          Producer(Q q) {
                this .q = q;
                new Thread (this, " Producer") . start ( ) ;
        }
public void  run ( ) {
                int i = o ;
                while (true)       {
                        q.put (i++) ;
                }
        }
}
class Consumer implements Runnable {
        Q q ;
        Consumer ( Q q ) {
                this. q = q;
                new Thread (this, "Consumer"). start ( ) ;
        }
        public void run ( ) {
                while (true) {
                  q. get ( ) ;
                }
        }
}
class PCFixed {
```

```
        public static void main (String args [] ) {
        Q q = new Q ( ) ;
        new Producer (q) ;
        new Consumer (q) ;
        System. out println ("Press Control-C to stop.") ;
        }
}
```

Inside get( ), wait( ) is called. This causes its execution to suspend until the Producer notifies you that some data is ready. When this happens, execution inside get( ) resumes.

After the data has been obtained, get( ) calls notify( ). This tells Producer that it is okay to put more data in the queue. Inside put( ), wait( ) suspends execution until the Consumer has removed the item from the queue. When execution resumes, the next item of data in the queue, and notify( ) is called. This tells the Consumer that it should now remove it.

Here is some output from this program, which shows the clean synchronous behavior:

```
Put :  1
Got :  1
Put :  2
Got :  2
Put :  3
Got :  3
Put :  4
Got :  4
Put :  5
Got :  5
```

14.10 Deadlock

We have discussed earlier, The special type of error that we need to avoid that relates specifically to multitasking is deadlock, which occurs when two threads have a circular dependency on a pair of synchronized objects.

Here we discuss an example, suppose one thread enters the monitor on object X and another thread enters the monitor on object Y. If the thread inX tries to call any synchronized method on Y, it will block as expected. However, if the thread in Y, in turn, tries to call any synchronized method on X, the thread wait forever, because to access X, it would have to release its own lock on Y so that the first thread could complete.

We can say that the Deadlock is a difficult error to debug for two reasons:

(A) Generally it occurs only rarely, when the two threads time-slice in just the right way.

(B) Deadlock involves more than two threads and two synchronized object. (i.e. deadlock can occur through a more convoluted sequence of events than just described.)

To understand deadlock properly, it is useful to see it in action.

Here we will discuss the next example which creates two classes, A and B, with methods foo( )

and bar( ), respectively, which pause briefly before trying to call a method in the other class.

The main class, named Deadlock, creates an A and a B instance, and then starts a second thread to set up the deadlock condition. The foo( ) and bar( ) methods use sleep( ) as a way to force the deadlock condition to occur.

// An example program that demonstrates the deadlock situation

```
class A {
  synchronized void foo (B b) {
        String name = Thread.currentThread ( ). getName ( ) ;
        System.out.printIn (name + " entered A.foo") ;
        try {
              Thread.sleep (1000) ;
         }    catch (Exception e)
              {
              system.out.printIn ("A Interrupted") ;
         }
        System.out.println ("name + " trying to call B.last ( ) ") ;
        b.last();
  }
        synchronized void last ( )  {
          System.out.printIn ("Inside A. last ") ;
        }
  }
class B {
  synchronized void bar (A a) {
        String name = Thread.currentThread ( ). getName ( ) ;
        System.out.println (name + " entered B.bar") ;
        try {
              Thread.sleep (1000) ;
         }    catch (Exception e)  {
              system.out.println ("B Interrupted") ;
         }
         System.out.println ("name + " trying to call A.last ( ) ") ;
         a.last ( ) ;
    }
  synchronized void last ( ) {
        System.out.println ("Inside A. last") ;
    }
}
```

```
class Deadlock implements Runnable {
        A a = new A ( ) ;
        B b = new B ( ) ;
        Deadlock ( ) {
                Thread. current Thread ( ) . setName  ( "MainThread") ;
                Thread t = new Thread (this, "RacingThread") ;
                t. start ( ) ;
                a. foo (b) ;                    // get lock on a in this thread.
                System. out. println ("Back in main thread") ;
}
   public void run ( ) {
        b. bar (a) ;                            // get lock on b in other thread.
        System. out println ("Back in other thread") '
}
        public static void main (String args [ ] )  {
                new Deadlock ( ) ;
        }
}
```

The Sample output of the program would be :

```
MainThread entered A.foo
RacingThread entered B.bar
MainThread trying to call B.last ( )
RacingThread trying to call A.last ( )
```

Since the program has deadlocked, you need to press CTRL-C to end the program. You can see a full thread and monitor cache dump by pressing CTRL-BREAK on a PC (or CTRL-\ on Solaris.). You will see that Racing Thread owns the monitor on b, while it is waiting for the monitor on a. At the same time, MainThread owns a and is waiting to get b. This program will never complete.

14.11 Suspending, Resuming and Stopping Threads

We can say that sometimes suspending execution of a thread is useful.

For example, a separate thread is used to display the time of day. If the user doesn't want a clock, then its thread can be suspended. Whatever the case, suspending a thread is a simple matter. Once suspended, restarting the thread is also a simple matter.

The mechanisms to suspend, stop, and resume threads distinguishes between Java 2 and old versions.

Therefore you should use the Java 2 approach for all new code, you still need to understand how these operations were accomplished for earlier Java environments.

For example, you may need to update or maintain older, legacy code. You also need to understand why a change was made for Java 2.

Suspending, Resuming, and stopping Threads Using Java 1.1 and Earlier Version:

Earlier to Java 2, a program used suspend ( ) and resume ( ), which are methods defined by Thread, to pause and restart the execution of a thread.

They have the following forms shown here:

final void suspend( )

final void resume( )

An example program that demonstrates these methods is given below:

```
// Using suspend( ) and resume( ) method
class NewThread implements Runnable {
  String name ;                                   // name of thread
  Thread t ;
  NewThread (String threadname) {
        name = threadname ;
        t = new Thread (this, name) ;
        System.out.println ("New thread : " + t) ;
        t.start ( ) ;                                   // Start the thread
  }
  // This is the entry point for thread.
  public void run ( ) {
        try {
          for (int i = 15; i > 0; i--) {
                System.out.println(name + " : " + i) ;
                Thread.sleep(200) ;
          }
        } catch (InterruptedException e) {
          System.out.println(name + " interrupted.") ;
        }
        System.out.println(name + " exiting.") ;
  }
}
class SuspendResume {
  public static void main (String args [ ] ) {
        NewThread ob1 = new NewThread ("One") ;
        NewThread ob2 = new NewThread ("Two") ;
try {
  Thread.sleep (1000) ;
  ob1.t.suspend ( ) ;
  System.out.println ( "Suspending thread One" ) ;
```

```
            Thread.sleep (1000) ;
            ob1.t.resume ( ) ;
            System.out.println ( "Resuming thread One" ) ;
             ob2.t.suspend ( ) ;
            System.out.println ( "Suspending thread Two" ) ;
            Thread.sleep (1000) ;
             ob2.t.resume ( ) ;
            System.out.println ( "Resuming thread Two" ) ;
          } catch (InterruptedException e) {
            System.out.println ( "Main thread Interrupted" ) ;
          }
          // Wait for threads to finish
          try {
            System.out.println ( "Waiting for threads to finish. " ) ;
            ob1.t.join ( ) ;
            ob2.t.join ( ) ;
          } catch (InterruptedException e) {
            System.out.println ( "Main thread Interrupted" ) ;
          }
          System.out.println ( "Main thread exiting. " ) ;
    }
}
```

The Sample output of this program is shown here:

```
New thread: Thread [One,5,main]
One: 15
New thread: Thread [Two,5,main]
Two: 15
One: 14
Two: 14
One: 13
Two: 13
One: 12
Two: 12
One: 11
Two: 11
Suspending thread One
```

```
Two: 10
Two: 9
Two: 8
Two: 7
Two: 6
Resuming thread One
Suspending thread Two
One: 10
One: 9
One: 8
One: 7
One: 6
Resuming thread Two
Waiting for threads to finish.
Two: 5
One: 5
Two: 4
One: 4
Two: 3
One: 3
Two: 2
One: 2
Two: 1
One: 1
Two exiting.
One exiting.
Main thread exiting.
```

Here also we can say that the thread class defines a method called stop( ) that stops a thread.

The signature of this is represented as:

```
void stop( )
```

If once a thread has been stopped, it cannot be restarted using resume( ) method.

14.12 Summary

Multithreading is a conceptual programming paradigm where a program is divided into two or more subprograms, which can be implemented at the same time another paralled. For example, one subprogram can display an animation on the screen while another may build the next animation to be displayed. This is something similar to dividing a task into subtasks and assigning them to different people for execution independently and simultaneously.

Therefore we can say the multithreading is a powerful programming tool that makes Java uniquely different from its fellow programming languages. Mulithreading is very useful in a number of ways. Mulithreading enables programmers or users to accomplish multiple things at one time.

Which can divide a long program into threads and execute them in parallel.

The creation of threads in Java is very simple. Threads are implemented in the form of objects that contain a method called ***run()*** method. This method is the heart and soul of any thread. It makes up the entire body of a thread and is the only method in which the thread's behavior can be implemented.

we have seen threads that use their own data and methods provided inside their run ( ) methods. What happens when they try to use data and methods outside themselves? On such occasion, they may compete for the same resources and may lead to serious problems. For example, one thread may try to read a record from a file while another is still writing to the same file. Depending on the situation, we amy get strange results. Java enables us to overcome this problem using a technique known as synchronization.

In case of Java, the multithreading helps to solve such problems by keeping a watch on such locations.

The Multithreading techniques are gaining acceptance and respect day to day as they improve the software development process.

14.13 Glossary

Multithreading It allows two or more pieces of the same program to execute concurrently.

Multithreaded program A program that contain multiple flows of control.

lightweight-threads are threads in Java are subprograms of a main application program and share the same memory space.

Running means that the processor has given its time to the thread for its execution.

By creating a thread class: Define a class that extends Thread class and override its run( ) method with the code required by the thread.

By converting a class to a thread: Define a class that implements Runnable interface. The Runnable interface has only one method, run( ), that is to be defined in the method with the code to be executed by the thread.

Monitor is like a key and the thread that holds the key can only open the lock.

getName: To obtain a thread's name.

getPriority: To obtain a thread's priority.

isAlive: To determine the thread is still running.

run means Entry point for the thread.

sleep means Suspend a thread for a period of time.

start means Start a thread by calling its run method.

Override: In the Software, the process of suspending the existing value of variable or program.

14.14 Further Readings

1. Java, UPTEC & Excel Books
2. Programming With Java, E. balaguruswamy, TMH

3. Complete Java Book, Madhav Oke, BPB
4. Complete Java 2 Certification Study Guide, BPB
5. Java2: The Complete Reference, Patrick Naughton & Herbert Schildt
6. Java & XML, O' Reilly
7. Who's Afraid of Java? Steve Heller, AP Professional
8. Teach Yourself Java 1.1 Programming in 24 Hours, Rogers Cadenhead, BPB
9. Java Now!, Kris Jamsa, Jamsa Press
10. The Java Language Specification, James Gosling, Bill Joy, Guy Steele, Addison-Wesley.

14.15 Answers to Self Learning Exercises

I

1. A thread is similar to a program that has a single flow of control. It has a beginning, a body, and an end and it execute commands sequentially.
2. The multithreading is a powerful programming tool that makes Java uniquely different from its fellow programming languages. Mulithreading is very useful in a number of ways. Mulithreading enables programmers or users to accomplish multiple things at one time. Which can divide a long program into threads and execute them in parallel.
3. These five states are following: -

   (i) Newborn State

   (ii) Runnable State

   (iii) Running State

   (iv) Blocked State

   (v) Dead State
4. The thread levaes control joins the queue at the end and again waits for its turn. This process of assigning time to threads is called time-slicing.
5. (i) The threads in Java are subprograms of a main application program and share the same memory space, they are named as lightweight-threads or lightweight processes.

   (ii) A program that contain mulitipe flows of control is known as *multithreaded program*.

II:

1. A new thread can be created in two ways:-

   (i) By creating a thread class

   (ii) By converting a class to a thread.
2. The Thread class in Java supports the following priorities in many ways:

   (i) MIN_PRIORITY

   (ii) NORM_PRIORITY

   (iii) MAX_PRIORITY
3. One thread may try to read a record from a file while another is still writing to the same file. Depending on the situation, we amy get strange results. Java enables us to overcome this problem using a technique known as synchronization. The keyword *synchronised* in

Java helps to solve such problems by keeping a watch on such locations. for example, the method that will read information from a file and the method that will update the same file may be declared as synchronized.

14.16 Unit End Questions

I. State True or False:

(i) Multithreading is a conceptual programming paradigm where a program is divided into two or more subprograms.

(ii) Each flow of control can be considered as a separate small program or module known as thread

(iii) A program that contain mulitipe flows of control is known as multithreaded program.

(iv) A suspended thread can be revived by using the **resume( )** method.

(v) Java permits us to set the priority of a thread by using the setPriority( ) method.

(vi) The **run( )** method in Java is known as the heart and soul of any thread.

(vii) A monitor is like a key and the thread that holds the key can only open the lock.

(viii) getPriority method is used to obtain a thread's priority.

(ix) sleep method is used to suspend a thread for a period of time.

(x) Deadlock is a difficult error to debug for two reasons.

II. Long Essay Type Questions:

Q1. What do you understand by Multithreading ? Describe its use in Java.

Q2. Explain the different stages of the Thread Life Cycle with suitable examples?

Q3. Describe the methods of creation of a thread in brief?

Q4. What do you mean by thread priority? Explain it.

Q5. Explain the Following Terms:

(i) Thread Synchronization

(ii) Interthread communication

(iii) Deadlock

(iv) Time Slicing