SURESH GYAN VIHAR UNIVERSITY
Accredited by NAAC with 'A+' Grade

# MASTER OF SCIENCES

## (M.Sc.)

## MMT-105

## PROGRAMMING

## IN C++

**Semester-I**

**Author-Dr. Harsh Verdhan Harsh**

**SURESH GYAN VIHAR UNIVERSITY**

**Centre for Distance and Online Education,**

**Mahal Road, Jagatpura, Jaipur-302017**

Published by:

**S. B. Prakashan Pvt. Ltd.**

WZ-6, Lajwanti Garden, New Delhi: 110046

Tel.: (011) 28520627 | Ph.: 9205476295

Email: info@sbprakashan.com | Web.: www.sbprakashan.com



**Designed & Graphic by :** S. B. Prakashan Pvt. Ltd.

Printed at :

**Suresh Gyanvihar University**
**Department of Mathematics**
**School of Science**

---

**M.Sc., Mathematics - Syllabus – I year – I Semester (Distance Mode)**

**COURSE TITLE : PROGRAMMING IN C++**

**COURSE CODE : MMT-105**

**COURSE CREDIT : 4**

---

## COURSE OBJECTIVES

While studying the **PROGRAMMING IN C++**, the Learner shall be able to:

CO 1: Develop programming skills in C++ and its object oriented concepts

CO 2: Review about the inline functions

CO 3: Represent arrays in C++

CO 4: Predict the uses of constructors and destructors.

CO 5: Describe the existing classes.

## COURSE LEARNING OUTCOMES

After completion of the **PROGRAMMING IN C++**, the Learner will be able to:

CLO 1: Interpret the concept of control structures and able to write simple programs using class concepts.

CLO 2: Describe the uses of function overloading.

CLO 3: Enable to write moderate level programs using Object concept.

CLO 4:Demonstrate and understanding to apply operator overloading concept.

CLO 5:Enable to efficiently use the techniques, skills, and computational skills to solve real time numerical problems

---

**BLOCK I: INTRODUCTION**

Beginning with C++ & Tokens, Expressions and Control Structures, Applications of C++– A simple C++ Program—– An Example with Class– Structure of C++ Program–Creating the Source File– Compiling and Linking–Introduction– Token and Keyword.

**BLOCK II: Functions in C++ and classes**

Introduction– the Main Function– Function Prototyping– Call by Reference–Return by Reference– Inline Function– Defaults Arguments– const Arguments– Function Overloading– Friend and Virtual Functions– C Structures Revisited– Specifying a Class– Defining Membership Functions– A C++ Program with Class– Making an Outside Function Inline– Nesting of Member Functions– Private Member Functions– Arrays with an Class

**BLOCK III: Objects & Constructors**

Introduction– – Memory Allocation for Objects– Static Data Member– Static Member Functions– Arrays of Objects– Objects as Function Arguments– Friendly Functions– Returning Objects– const Member Functions– Pointers of Members– Local Classes– Constructors–Parameterized Constructors– Multiple constructors in a class– Constructors with Default Arguments.

**BLOCK IV: Destructors & Operator Overloading and Types Conversions**

Introduction –– Dynamic Initialization of Objects– Copy Constructor– Dynamic Constructors– Constructing Two–Dimensional Arrays– const Objects –Destructors– Introduction– Defining Operator Overloading– Overloading Unary Operators– Overloading Binary Operators– Overloading Binary Operators Using Friends– Manipulation of Strings Using Operators– Rules For Overloading Operators– Type Conversions.

**BLOCK V: Inheritance: Extending Classes and Pointers, Virtual Functions and Polymorphism**

Introduction–Defining Derived Classes– Single Inheritance–Making a Private Member Inheritance– Making a Private Member Inheritable– Multilevel Inheritance–Multiple Inheritance–Hierarchical Inheritance–Hybrid Inheritance–Virtual Base Classes–Abstract Classes– Constructors in Derived Classes– Member Classes: Nesting of Classes–Introduction– Pointers to Objects–this Pointer– Pointers to Derived Classes–Virtual Functions– Pure Virtual Functions.

**REFERENCE BOOKS:**


1. E.Balagurusamy, Object Oriented Programming with C++, 4th Edition, The McGraw–Hill CompanyLtd, New Delhi, 2008.
2. V. Ravichandran, Programming with C++, Second Edition Tata McGraw – Hill, New Delhi, 2006.
3. H.Schildt,The complete Reference of C++, Tata–McGraw–Hill publishing Company Ltd. New Delhi, 2003.
4. S.B. Lipman and J.Lafer, C++ Primer, Addition Wesley, Mass., 1998.
5. Ashok N.Kamthane, Object Oriented Programming with ANSI and TURBO C++, Pearson Education(P) Ltd, 2003.
6. BjarmeStroustrup, The C++ Programming Language, AT & T Labs, Murray Hills, NewJersey, 1998.

Block-I

Unit – I: Introduction

Unit – II: Tokens, Expressions and Control Structures

Block- II

Unit – III: Functions in C++

Unit – IV: Classes and Objects - I

Block - III

Unit – V: Classes and Objects - II

Unit – VI: Constructors and Destructors

Block - IV

Unit – VII: Constructors and Destructors

Unit – VIII: Operator Overloading and Type Conversions

Block - V

Unit – IX: Inheritance: Extending Classes

Unit – X : Pointers, Virtual Functions and Polymorphism

## CONTENTS

# BLOCK – I

Objectives:

After Completion of this block, students will be able to

1. Understand basic data types and its limits
2. Familiarize control structures
3. Write simple programs using Class concept and execute them.

# UNIT – I Beginning with C++

## 1.1. Introduction

C++ is an object-oriented programming language. It was developed by Bjarne Stroustrup at AT & T Bell Laboratories in Murray Hill, New Jersey, USA in the early 1980's. It was developed by combining the best of Simula67 and C, and support object-oriented programming features. Thus, C++ is an extension of C with a major addition of the class construct feature of Simula67. Since the class was a major addition to the original C language, Stroustrup initially called the new language "C with classes". Later, in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, suggesting that C++ is an incremented version of C.

During the early 1990's the language underwent a number of improvements and changes. In November 1997, the ANSI/ISO standards committee standardised these changes and added several new features to the language specifications.

C++ is a superset of C. Almost all C programs are also C++ programs. However, there are a few minor differences that will prevent a C program to run under C++ compiler.

The most important facilities that C++ adds on to C are classes, inheritance, function overloading, and operator overloading. These features enables creation of abstract data types, inherit properties from existing data types and support polymorphism, there by making C++ a truly object-oriented language.

The object-oriented features in C++ allow programmers to build large programs with clarity, extensibility and ease of maintenance, incorporating the spirit and efficiency of C. The addition of new features has transformed C from a language that currently facilitates top-down, structured design, to one that provides bottom-up, object-oriented design.

## 1.2 Applications of C++

C++ is a versatile language for handling very large programs. It is suitable for virtually any programming task including development of editors, compilers, databases, communication systems and any complex real-life application systems.

- Since C++ allows us to create hierarchy-related objects, we can build special object-oriented libraries which can be used later by many programmers.
- While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get close to the machine-level details.
- C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object.

## 1.3 A Simple C++ program

The following program prints a string on the screen.

```
#include <iostream>

using namespace std;

int main()

{

cout<<"Welcome to C++ Language\n";

return 0;

}
```

**Program Features**

Like C, the C++ program is a collection of functions. The above example contains only one function, main(). The execution begins at main(). Every C++ program must have a main(). C++ is a free-form language. With a few exceptions, the compiler ignores carriage returns and white spaces. Like C, the C++ statements terminate with semicolons.

**Comments**

C++ has a new comment statement // (double slash). Comments start with a double slash symbol and terminate at the end of the line. A comment may start anywhere in the line, and whatever follows till the end of the line is ignored. Note that there is no closing symbol.

A double slash comment is basically a single line comment. Multiline comment can be written as follows:

```
//This is an example for
//multiline comment statement in C++
```

The C comment symbols /*..... */ are valid and are more suitable for multiline comments.

The above multiline comment can be written as follows:

```
/*This is a an example for
multiline comment statement in C++ */
```

We can use either or both styles. However, we cannot insert a //style comment within the text of a program line. For example, the double slash comment cannot be used in the manner as shown below:

```
for (j=0; j < n; /*loops n times */ j++)
```

**Output Operator**

The statement

```
cout<< "Welcome to C++ Language";
```

causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, cout and <<. The identifier cout (pronounced 'C out') is a predefined object that represent the standard output stream in C++. Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices.

The operator << is called the *insertion* or *put to* operator. It inserts (or sends) the contents of the variable on its right to the object on its left.

The following statement will display the content of string

```
cout<<string;
```

Note that << is a bit-wise left-shift operator, and it can still be used for this purpose. Depending on the context, the operator can be used for different purposes. This concept is known as *operator overloading*, an important aspect of polymorphism.

**The iostream file**

A C++ program typically contains pre-processor directive statements at the beginning. Such statements are preceded with a # symbol to indicate the presence of a pre-processor directive to the compiler, which in turn lets the pre-processor handle the # directive statement. All C++ programs begin with a #include directive that include the specified header file contents into the main program.

The following directive

```
#include<iostream>
```

causes the preprocessor to add the contents of the iostream file to the program. It contains declarations for the identifier cout and the operator <<. Old version of C++ uses a header file called iostream.h. This is one of the

changes introduced by ANSI C++. We should use iostream.h if the compiler does not support ANSI C++ features.

The header file iostream should be included at the beginning of all programs that use input/output statements.

| Header file | Contents and purpose | New version |
|---|---|---|
| <ctype.h> | Contains function prototypes for functions that test characters for certain properties, and function prototypes for functions that can be used to convert lowercase letter to uppercase letters and vice versa | <cctype> |
| <math.h> | Contains function prototypes for math library functions | <cmath> |
| <stdio.h> | Contains function prototypes for the standard input/output library functions and information used by them | <cstdio> |
| <stdlib.h> | Contains function prototypes for conversion of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions | <cstdlib> |
| <string.h> | Contains function prototypes for C-style string processing functions | <cstring> |
| <iostream.h> | contains function prototypes for the standard input and standard output functions | <iostream> |

| | | |
|---|---|---|
| <iomanip.h> | Contains function prototypes for the stream manipulators that enable formatting of steams of data | <iomanip> |
| <fstream.h> | Contains function prototypes for functions that perform input from files on disk and output to files on disk. | <fstream> |

**Namespace**

Namespace is a new concept introduced by the ANSI C++ standards committee. This defines a scope for the identifiers that are used in a program. For using the identifiers defined in the namespace scope we must include the using directive, like

using namespace std;

Here, std is the namespace where ANSI C++ standard class libraries are defined. All ANSI C++ programs must include this directive. This will bring all the identifiers defined in std to the current global scope. **using** and **namespace** are the new keywords of C++.

**Return type of main()**

In C++, main() returns an integer type value to the operating system. Therefore, every main() in C++ should end with a return() statement; otherwise a warning or an error might occur. Since main() returns an integer type value, return type for main() is explicitly specified as int. Note that the default return type for all functions in C++ is **int**.

**More C++ Statements**

The following program read two numbers from the keyboard and display their average on the screen.

```
//Average of two numbers
#include<iostream>
using namespace std;
int main()
{
float n1, n2, sum, avg;
cout<<"Enter two numbers ";
cin>>n1;
cin>>n2;
sum = n1 + n2;
avg = sum/2;
cout<<"The sum of two numbers is "<<sum<<"\n";
cout<<"Average = "<<avg<<"\n";
return 0;
}
```

The output of the program is

Enter two numbers 9 5

The sum of two numbers is 14

Average = 7

**Variables**

The program uses four variables n1, n2, sum and avg. They are declared as type float by the statement.

float n1, n2, sum, avg;

All variables must be declared before they are used in the program.

**Input operator**

The statement

cin>>n1;

is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable n1. The identifier cin (pronounced 'C in') is a predefined object in C++ that corresponds to the standard input stream. Here, this stream represents the keyboard.

The operator >> is known as *extraction* or get from operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right. This corresponds to the familiar scanf() operation. Like <<, the operator >> can also be overloaded.

**Cascading of I/O Operators**

We have used the insertion operator << repeatedly in the last two statements for printing results.

The statement

cout<<" The sum of two numbers is "<<sum<<"\n";

first sends the string "The sum of two numbers is " to cout and then sends the value of sum. Finally, it sends the new line character so that the next output

will be in the new line. The multiple use of << in one statement is called *cascading.* When cascading an output operator, we should ensure necessary blank spaces between different items. Using the cascading technique, the last two statements can be combined as follows:

```
cout<<" The sum of two numbers is "<<sum<<"\n"
     <<"Average = "<< avg <<"\n";
```

This is one statement by provides two lines of output. If we want only one line of output, the statement would be:

```
cout<<" The sum of two numbers is "<<sum<<","
     <<"Average = "<< avg <<"\n";
```

The output will be:

```
The sum of two numbers is 14, Average = 7
```

We can also cascade input operator >> as shown below:

```
cin>>n1>>n2;
```

The values are assigned left to right. That is, if we key in two values, say 9 and 5, then 9 will be assigned to n1 and 5 to n2.

### 1.4 An example with Class

One of the major features of C++ is classes. They provide a method of binding together data and functions which operate on them. Like structures in C, classes are user-defined data types.

The following program uses class.

```
//use of class
#include<iosteam>
using namespace std;
```

```
class person
{
char name[30];
int age;

public:
void getdata(void);
void display(void);

};

void person :: getdata(void)
{
cout<<"Enter name: ";
cin >> name;
cout<<"\nEnter age: ";
cin>>age;
}

void person :: display(void)
{

cout<<"\nName : "<<name;
cout<<"\nAge : "<<age;
}

int main()
{

person p;
p.getdata();
p.display();
return 0;
}
```

The output of the program is

Enter Name: Anandan

Enter Age: 20

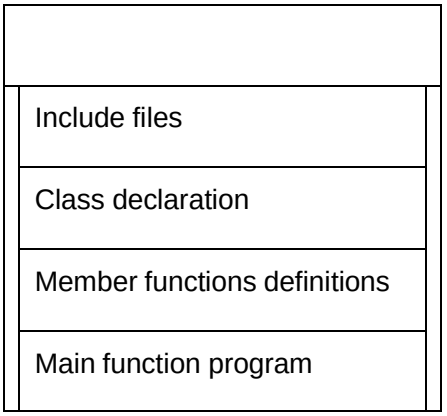
Name: Anandan
Age: 20

The program defines person as a new data of type class. The class person includes two basic data type items and two functions to operate on that data. These functions are called member functions. The main program uses person to declare variables of its type. Class variables are known as objects. Here, p is an object of type person. Class objects are used to invoke the functions defined in that class.

**Note:** cin can read only one word and therefore we cannot use names with blank spaces.

### 1.5 Structure of C++ program

A typical C++ program would contain four sections as shown below. These sections may be placed in separate code files and then compiled independently or jointly.




It is a common practice to organize a program into three separate files. The class declarations are placed in a header file and the definitions of member functions go into another file. This approach enables the programmer to separate the abstract specification of the interface (class

definition) from the implementation details (member functions definition). Finally, the main program that uses the class is placed in a third file which "includes" the previous two files as well as any other files required.

This approach is based on the concept of client-server model. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

### 1.6. Creating the Source file

Like C programs, C++ programs can be created using any text editor. For example, on the UNIX, we can use vi or ed text editor for creating and editing the source code. On the DOS system, we can use edlin or any other editor available or a word processor system under non-document mode.

Some systems such as Turbo C++ provides an integrated environment for developing and editing programs. Appropriate manuals should be consulted for complete details.

The file name should have a proper file extension to indicate that it is a C++ program file. C++ implementations use extensions such as .c, .cc, .cpp, and .cxx. Turbo C++ and Borland C++ use .c for C programs and .cpp (C plus plus) for C++ programs. Zortech C++ system uses .cxx while UNIX AT & T version uses .C (capital C) and .cc. The operating system manuals should be consulted to determine the proper file name extensions to be used.

### 1.7 Compiling and Linking

The process of compiling and linking again depends upon the operating system. A few popular systems are discussed below:

**UNIX AT & T C++**

The process of implementation of a C++ program under UNIX is similar to that of a C program. We should use the “CC” (uppercase) command to compile the program. Note that “cc” (lowercase) is used to compile C program. The command

CC example.c

At the UNIX prompt would compile the C++ program source code contained in the file example.C. The compiler would produce an object file example.o and then automatically link with the library functions to produce an executable file. The default executable filename is a.out.

A program spread over multiple files can be compiled as follows:

CC file1.C file2.o

The statement compiles only the file file1.C and links it with the previously compiler file2.o file. This is useful when only one of the files needs to be modified. The files that are not modified need not be compiled again.

**Turbo C++ and Borland C++**

Turbo C++ and Borland C++ provide an integrated program development environment under MS DOS. They provide a built-in editor and a menu bar which includes options such as File, Edit, Compile and Run.

We can create and save the source files under the File option, and edit them under the Edit option. We can then compile the program under the Compile option and execute it under the Run option. The Run option can be used without compiling the source code. In this case, the RUN command causes the system to compile, link and run the program in one step. Turbo

C++ being the most popular compiler, creation and execution of programs under Turbo C++ involves the following steps.

1. Develop the program (source code)
2. Select a suitable file name under which you would like to store the program.
3. Create the program in the computer and save it under the filename you have decided. This file is known as *source code* file.
4. Compile the source code. The file containing the translated code is called object code file. If there are any errors, debug them and compile the program again.
5. Link the object code with other library code that are required for execution. The resulting code is called the executable code. If there are errors in linking, correct them, compile the program again.
6. Run the executable code and obtain the results, if there are no errors.
7. Debug the program, if errors are found in the output.
8. Go to Step 4, and repeat the process again.

**Visual C++**

It is a Microsoft application development system for C++ that runs under Windows. Visual C++ is a visual programming environment in which basic program components can be selected through menu choices, buttons, icons, and other predetermine methods.

The Microsoft Corporation has introduced a Windows based C++ development environment named as Microsoft Visual C++ (MSVC). This development environment integrates a set of tools that enable a programmer to create and run C++ programs with ease and style. Microsoft calls this integrated development environment (IDE) as Visual Workbench. Microsoft Visual Studio, a product sold by Microsoft Corporation, also includes, Visual C++, in addition to other tools like Visual Basic, Visual J++, Visual Foxpro etc.

**Example Programs**

1. Write a program to display the following output using a single cout statement.

   Maths = 90, Physics = 77, Chemistry = 69.

```
//To display given output using single cout statement
#include<iostream>
using namespace std;

int main()

{

int M = 90, P = 77, C = 69;

cout<<"Maths = "<<M <<"Physics = "<< P <<"Chemistry = "<<C;

return 0;

}
```

**2. Write a program to read two numbers from the keyboard and display the larger value on the screen.**

```
//To print larger of given two integers without using conditional
statement

#include<iostream>
#include<cmath>

using namespace std;

int main()

{

int x,y;

clrscr();

cout<<"Enter two numbers\n";
```

```
cin>>x>>y;
cout<<"The larger among given numbers is"<< (x+y + abs(x-y))/2;
return 0;
}
```

**3. Write a program to read the values of a, b and c and display the value of x, where x = a/b - c.**

```
#include<iostream>
using namespace std;
int main()
{
float a, b, c, x;
clrscr();
cout<<"Enter three numbers\n";
cin>>a>>b>>c;
x = a/b – c;
cout<<"The value of the expression is "<<x;
return 0;
}
```

4. **Write a C++ program that will ask for a temperature in Fahrenheit and display it in Celsius using class called temp and member function.**

```
//Convert temperature from Fahrenheit to Centrigrade using Class

#include<iostream>
using namespace std;

class temp
{
float c;
public:
void Celsius (float f) {c = (f – 32) * 5./9;}
void display() { cout<< “Celsius value is “<<c; }
};

int main()
{
float ff;
temp T;
cout<<”Enter Temperature in Fahrenheit\n”;
cin>>ff;
T.Celsius(ff);
T.display();

return 0;
}
```

## SUMMARY

- C++ Is a superset of C language.
- C++ adds a number of object-oriented features such as objects, inheritance, function overloading and operator overloading to C. These feature enable building of programs with clarity, extensibility and case of maintenance.
- C++ can be used to build a variety of systems such as editors, compilers, databases, communication systems, and many more complex real-life application systems.
- C++ supports interactive input and output features and introduces a new comment symbol // that can be used for single line comments. It also supports C-style comments.
- Like C programs, execution of all C++ programs begins at main() function and ends at return ( ) statement. The header file **iostream** should be included at the beginning of all programs that use input/output operations.
- All ANSI C++ programs must include **using namespace std** directive.
- A typical C++ program would contain four basic sections, namely, include files section, class declaration section, member function section and main Function section,
- Like C programs, C++ programs can be created using any text editor, & Most compiler systems provide an integrated environment for developing and executing programs. Popular systems are UNIX AT&T C++,, Turbo C++ and Microsoft Visual C++.

**Review Questions:**

1. Why do we need the preprocessor direction #include <iostream>?
2. How does main() function in C++ differ from main() in C?
3. What do you think is the main advantage of the comment // in C++ as compared to the old C type comment?

**Programming Exercises:**

1. Use a class concept to find the sum of two numbers.
2. Write a C++ program that will ask for a temperature in Celsius and display it in Fahrenheit using class called temp and member function.

## UNIT – II

## TOKENS, EXPRESSIONS AND CONTROL STRUCTURES

### 2.1 Introduction

C++ is a superset of C and therefore most constructs of C are legal in C++, with their meaning unchanged. However, there are some exceptions and additions. We shall discuss these exceptions and additions with respect to tokens and control structures.

### 2.2 Tokens

The smallest individual unit in a program are known as tokens. C++ has the following tokens:

* Keywords
* Identifiers
* Constants
* Strings
* Operators

### 2.3 Keywords

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements.

Many of the keywords are common to both C and C++. Additional keywords have been added to the ANSI C keywords in order to enhance its features and make it an object-oriented language. The C++ keywords are as follows:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| asm | Auto | break | case | catch | char | class | const |
| continue | default | delete | do | double | else | enum | extern |
| float | For | friend | goto | If | inline | int | long |
| new | operator | private | protected | public | register | return | short |
| signed | sizeof | static | struct | switch | template | this | throw |
| try | typedef | union | unsigned | virtual | void | volatile | while |

C++ keywords

| | | | | | |
|---|---|---|---|---|---|
| bool | const_cast | dynamic_cas | t explicit | export | false |
| namespace | reinterpret_cast | static_cast | true | typeid | typename |
| mutable | using | wchar_t | | | |

Keywords added by ANSI C++

## 2.4 IDENTIFIERS AND CONSTANTS

Identifiers refer to the names of variables, functions, arrays, classes etc., created by the programmer. They are the fundamental requirements of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

- Only alphabetic character, digits and underscore are permitted.
- The name cannot start with a digit.
- Uppercase and lowercase letters are distinct.
- A declared keyword cannot be used as a variable name.

A major difference between 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 restriction on the length of such a variable name.

*Constants* refers to fixed value that do not change during the execution of a program.

Like C, C++ supports several kinds of literal constants. They include integers, characters, floating point numbers and strings. Literal constants do not have memory locations.

Examples:

123 //decimal integer

12.34 //floating point integer

O37 //octal integer

| | |
|---|---|
| OX2 | // 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 for character sets that cannot fit a character into a single byte. Wide-character literals begin with the letter L.

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

**2.5 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 type representation is machine specific in C++. The following table lists all combinations of the basic data types and modifiers along with their size and range for a 16-bit word machine.

| 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 | -32768 to 32767 |
| short int | 2 | -32768 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 |

Size and range of C++ basic data types

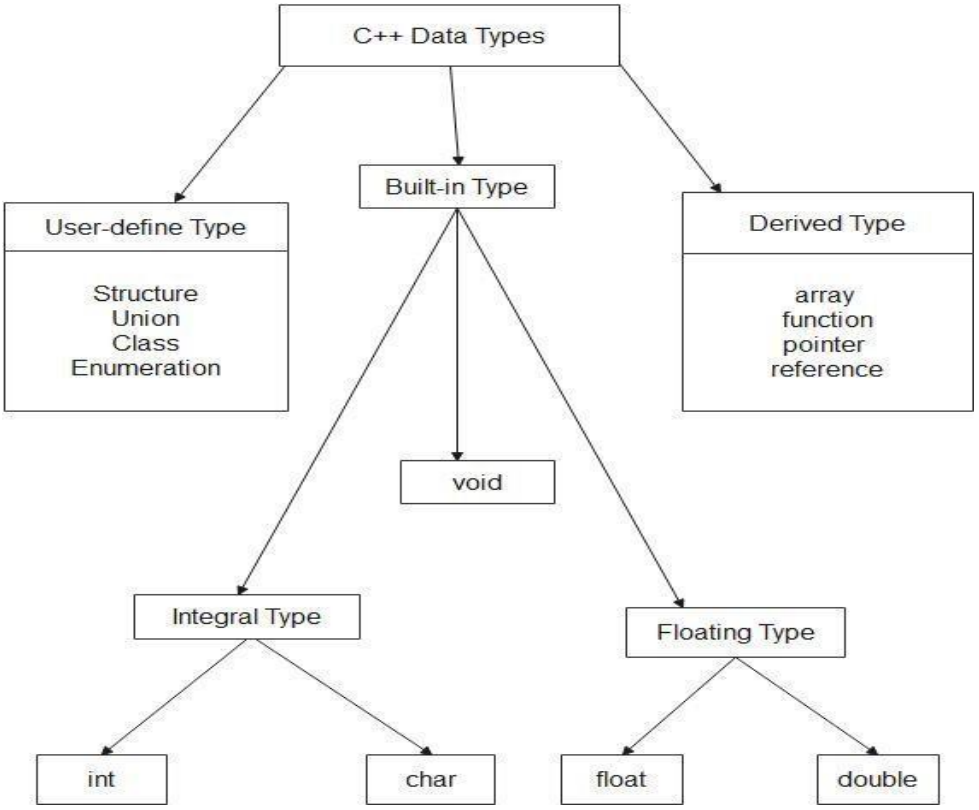



Fig. 2.1 Hierarchy of C++ data types

The type **void** was introduced in ANSI C. Two normal uses of **void** are (1) to specify the return type of a function when it is not returning any value, and (2) to indicate an empty argument list to a function.

Example:

```
void funct1(void);
```

Another use of **void** is in the declaration of generic pointers.

Example:

```
void *gp;       //gp becomes a generic pointer
```

A generic pointer can be assigned a pointer value of any basic data type, but it may not be dereferenced. For example,

```
int *ip;            //int pointer
gp = ip;            //assign int pointer to void pointer
```

are valid statements. But, the statement,

```
*ip = *gp;
```

Is illegal. It would not make sense to dereference a pointer to a **void** value.

Assigning any pointer type to a **void** pointer without using cast is allowed in C++. and ANSI C. In ANSI C, we can also assign a **void** pointer to a non-**void** pointer without using a cast to non-void pointer type. This is not allowed in C++. For example,

```
void   *ptr1;
char   *ptr2;
ptr2 = ptr1;
```

are all valid statements in ANSI C but not in C++. A **void** pointer cannot be directly assigned to other type pointers in C++. We need to use a cast operator as shown below:

```
ptr2 = (char *) ptr1;
```

## 2.6 User-Defined Data Types

### 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 Type

An enumerated data type is a 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. This facility provides an alternative means for creating symbolic constants. The syntax of an **enum** statement is similar to that of the **struct** statement. Example:

```
enum shape {circle, square, triangle};

enum color {red, blue, green, yellow};
```

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

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

color background; //background is of type color
```

**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:

```
color background = blue;                //allowed
color background = 7;          //Error in C++
color background = (color) 7;          //ok
```

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

```
int c = red;                //valid, color 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 color {red, blue = 4, green = 8};
enum color {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 predecessors.

C++ also permits the creating of anonymous **enums** (i.e. **enums** without tag name). 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:
                ……
                ……
        }
cout<<”Enter shape code:”;
cin>>code;
   }
cout<<”BYE\n”;
return 0;
}
```

## 2.7 DERIVED DATA TYPES

### Arrays

It is a group of related data items that share a common name. The application 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. The following is valid in C++

```
char string[4] = "xyz";
```

### Functions

Function have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programs. Many of these modifications and improvements were driven by the requirements of the object-oriented concepts of C++. Some of these were introduced to make the C++ program more reliable and readable.

### Pointers

Pointers are declared and initialized as in C. Examples:

```
int *ip; // 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 the contents of what it 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 a constant. In this case, neither the address assigned to the pointer cp nor the contents it points to can be changed.

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

## 2.8 SYMBOLIC CONSTANTS

There are two ways of creating symbolic constants in C++

* Using the qualifier **const**, and
* Defining a set of integer constants using **enum** keyword.

In both C and C++, any value declared as **const** cannot be modified by the program in any way. However, there are some differences in implementation. In C++, we can use **const** in a constant expression, such as

```
const int size = 10;
char name[size];
```

This would be illegal in C. **const** allows us to create typed constants instead of having to use **#define** to create constants that have no type information.

As with **long** and **short**, if we use the **const** modifier alone, it defaults to **int**. For example,

```
const size = 10;
```

means const int size = 10;

The named constants are just like variables except that their values cannot be changed.

C++ requires a const to be initialized. ANSI C does not require an initializer; if none is given, it initializes the **const** to 0.

The scoping of **const** value differs. A **const** in C++ defaults to the internal linkage and therefore, it is local to the file where it is declared. In ANSI C, **const** values are global in nature. They are visible outside the file in which they are declared. However, they can be make local by declaring them as **static**. To give a **const** value an external linkage so that it can be referenced from another file, we must explicitly defines it as an **extern** in C++. Example:

```
extern const total = 100;
```

Another method of naming integer constants is by enumeration as under:

```
enum {x, y, z};
```

This defines X, Y and Z as integer constants with values 0, 1, and 2 respectively. This is equivalent to:

```
const x = 0;

const y = 1;

const z = 2;
```

We can also assign values to X, Y and Z explicitly. Example:

```
enum{X = 100, Y = 50, Z = 200};
```

such values can be any integer values.

## 2.9 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 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 int, and therefore,

sizeof('x')

is equivalent to

sizeof(int)

in C. In C++, however, **char** is not promoted to the size of **int** and therefore,

sizeof('x')

equals

sizeof(char)

**2.10 DECLARATION OF VARIABLES**

In C, all variables must be declared before they are used in executable statements. This is true with C++ as well. However, there is a significant difference between C and C++ with regard to the place of their declaration in the program. C requires all the variables to be defined at the beginning of a scope. When we read a C program, we usually comes across a group of variable declarations at the beginning of each scope level. Their actual use appears elsewhere in the scope. Sometimes far away from the place of declaration. Before using a variable, we should go back to the beginning of the program to see whether it has been declared and, if so, of what type.

C++ allows the declarations of a variable anywhere in the scope. This means that a variable can be declared right at the place of its first use. This makes the program much easier to write and reduces the errors that may be caused by having to scan back and forth. It also makes the program easier to understand because the variables are declared in the context of their use. The example below illustrate this point

```
int main()
{
float x;
float sum = 0;
for (int i = 1; i< 5; i++)
        {
        cin>>x;
        sum = sum + x;
        }
float average;
average = sum/(i-1);
cout<<average; return 0;
return 0;
}
```

The only disadvantage of this style of declaration is that we cannot see all the variables used in a scope at a glance.

## 2.11 DYNAMIC INITIALIZATION OF VARIABLES

In C, a variable must be initialized using a constant expression, and the C compiler would fix the initialization code at the time of compilation. In C++, however, permits initialization of the variables at run time. This is referred to the *dynamic initialization*. In C++, a variable can be initialized at run time using expressions at the place of declaration. For example, the following are valid initialization statements:

```
……..
……..
int n = strlen(string);
…….
float area = 3.14159*rad*rad;
```

Thus, both the declaration and the initialization of a variable can be done simultaneously at the place where the variable is used for the first time. The following two statements float average; average = sum/i; can be combined into a single statement

```
float average = sum/i;          //initialize dynamically at run time
```

Dynamic initialization is extensively used in object-oriented programming. We can create exactly the type of object needed, using information that is known only at the run time.

## 2.12 REFERENCE VARIABLES

C++ introduces a new kind of variables known as the *reference* variable. A *reference* variable provides an *alias* (alternative name) for a previously defined variables. For example, if we make the variable **sum** a reference to the variable **total** then **sum** and **total** can be used interchangeably to represent that variable. A reference variable is created as follows:

```
data-type & reference-name = variable-name
```

Example:

```
float total = 100;

float &sum = total;
```

**Total** is a **float** type variable that has already been declared; **sum** is the alternative name declared to represent the variable **total**. Both the variable refer to the same data object in the memory. The statements

```
cout<<total;
```

and

```
cout<<sum;
```

both print the value 100. The statement

```
total = total + 10;
```

will change the value of both total and sum to 110. Likewise, the assignment

```
sum = 0;
```

will change the value of both the variables to zero.

A reference variable must be initialized at the time of declaration. This establishes the correspondence between the reference and the data object which it names. It is important to note that the initialization of a reference variable is completely different from assignment to it.

C++ assigns additional meaning to the symbol &. Here & is not an address operator. The notation **float &** means reference to **float**. Other examples are:

```
int n[10];
int &x = n[10];          //x is alias for n[10]
char &a = 'n';           //initialize reference to a literal
```

The variable **x** is an alternative to the array element **n[10]**. The variable **a** is initialized to the newline constant. This creates a reference to the otherwise unknown location where the newline constant **n** is stored.

The following references are also allowed:

i)
```
int x;
  int *p = &x;
  int &m = *p;
```

ii)
```
int &n = 50;
```

The first set of declarations causes **m** to refer to **x** which is pointed to by the pointer **p** and the statement in (ii) creates an **int** object with value 50 and the name **n**.

A major application of reference variables is in passing arguments to functions. Consider the following:

```
void f(int &x)    //uses reference
 {
 x = x + 10;
 }
 int main()
 {
int m = 10;
f(m);                        //function call
……..
…….
}
```

When the function call f(m) is executed, the following initialization occurs:

```
int &x = m;
```

Thus, x becomes an alias of m after executing the statement

```
f(m);
```

such function calls are known as *call by reference*. The call by reference mechanism is useful in object-oriented programming because it permits the manipulation of objects by reference, and eliminates the copying of object parameters back and forth. It is also important to note that references can be created not only for built-in data types but also for user-defined data types such as structures and classes.

**2.13 OPERATORS IN C++**

C++ has rich set of operators. All C operators are valid in C++ also. In addition, C++ introduce some new operators. In addition to the insertion operator << and the extraction operator >>, other new operators are:

:: scope resolution operator

::* pointer-to-member declaratory

->*  pointer-to-member operator

.*  pointer-to-member operator

delete  memory release operator

endl  line feed operator

new  memory  allocation  operator

setw field width operator

In addition, C++ also allows us to provide new definition to some of the built-in operators. That is, we can give several meanings to an operator, depending upon the types of arguments used. This process is known as *operator overloading*.

**2.14 Scope Resolution Operator**

Like C, C++ is also a block-structured language. Blocks and scopes can be used in constructing programs. Note that same variable name can 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 be local to that block. Consider the following segment of a program:

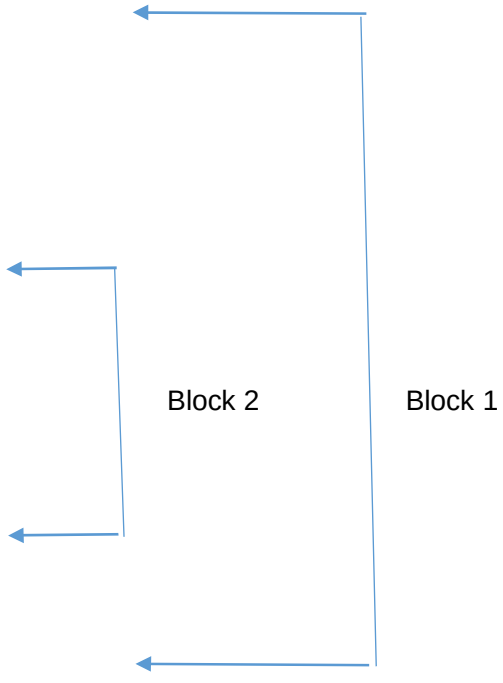
```
……..
…….
{
int x = 10;
……..
……..
}
…….
…….
```

```
{
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 is common:

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



Block 2 is contained in block 1. Note that a declaration in an inner block *hides* a declaration of the same variable in an outer block and, therefore, each declaration of **x** 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. In 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). The following program illustrates this feature:

```
#include<iostream>

using namespace std;

int m = 10;                 //global m

int main()

{
int m = 20;                         //m redeclared, local to main

        {
        int k = m;
        int m = 30;       // m declared again local to inner block

        cout<<"We are in inner block\n";
        cout<<"k =  "<<k<<"\n";
        cout<<"m = "<<m<<"\n";
        cout<<"::m = "<<::m <<"\n";
        }

cout<<"\n We are in outer  block\n";
```

```
cout<<"m = "<< m<<"\n";
cout<<"::m = "<< ::m << "\n";
return 0;
}
```

The output of the above program would be:

```
We are in inner block
k = 20
m = 30
:: m = 10
We are in outer block
m = 20
::m = 10
```

In the above program, the variable **m** is declared at three places, namely, outside the **main()** function, inside the **main()**, and inside the inner block.

Note that ::m will always refer to the global **m**. In the inner block, ::m refers to the value 10 and not 20.

A major application of the scope resolution operator is in the class to which a member function belongs.

### 2.15 Member Dereferencing Operators

C++ permits us to define a class containing various types of data and functions as members. It also permits us to access the class members through pointers. In order to achieve this, C++ provides a set of three pointer-to-member operators.

| Operator | Function |
|---|---|
| ::* | To declare a pointer to a member of a class |
| * | To access a member using object name and a pointer to that member |
| ->* | To access a member using a pointer to the object and a pointer to that member |

## 2.16 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. Examples:

```
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;

Assigns 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 in:

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[0]** 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 others 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**.

If sufficient memory is not available for allocation, like **malloc()**, **new** will 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.

### 2.17 Manipulators

Manipulators are operators that are used to format the data display. The most commonly used manipulators are **endl** and **setw**.

The **endl** manipulator, when used in an output statement, causes a linefeed to be inserted. It has the same effect as using the newline character "\n". For example, the statement

```
……..
……..
cout<< “m = “<< m<<endl
    << “n = “<< n<<endl
    << “p = “<< p<<endl
……..
……..
```

would cause three lines of output, one for each variable. If we assume the values of the variables as 2597,14, and 175 respectively, the output will appear as follows:

m = | 2 | 5 | 9 | 7 |

n = | 1 | 4 |

p =

| | | |
|---|---|---|
| 1 | 7 | 5 |

It is important to note that this form is not the ideal output. It should rather appear as under

m = 2597

n = 14

p = 175

Here, the numbers are right justified, This form of output is possible only if we can specify a common field width for all the numbers and force them to be printed right-justified. The **setw** manipulator does this job. It is used as follows:

cout<< setw(5) << sum << endl;

The manipulator **setw(5)** specifies a field width 5 for printing the value of the variable sum. This value is right-justified within the field as shown below:

| | | | | |
|---|---|---|---|---|
| | | 3 | 4 | 5 |

The following program illustrates the use of **endl** and **setw.**

```
#include<iostream>
#include<iomanip>                //for setw

using namespace std;

int main()
{
int Basic= 950, Allowance = 95, Total = 1045;
```

```
cout<<setw(10)<<"Basic"<<setw(10)<<Basic<<endl
     <<setw(10)<<"Allowance"<<setw(10)<<Allowance<<endl
     <<setw(10)<<"Total"<<setw(10)<<Total<<endl;
return 0;
}
```

The output of the above program is:

```
         Basic       950
     Allowance        95
         Total      1045
```

Note that character strings are also printed right-justified.

We can also write our own manipulators as follows:

```
#include<iostream>
ostream & symbol (ostream & output)
{
return output <<"\t Rs.";
}
```

The **symbol** is the new manipulator which represents **Rs.**. The identifier **symbol** can be used whenever we need to display the string Rs.

### 2.18 Type Cast Operator

C++ permits explicit type conversion of variables or expressions using the type cost 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

### 2.19 Expressions and Their Types

An expression is a combination of operators, constants and variables arranged as per the rules of the language. It may also include function calls which return values. An expression may consist of one or more operands, and zero or more operators to produce a value. Expressions may be of the following seven types:

- Constant expressions
- Integral expressions

- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions

An expression may also use combinations of the above expressions. Such expressions are known as compound expressions.

**Constant Expressions**

Constant expressions consist of only constant values. Examples:

15

20 + 5/2.0

'x'

**Integral Expressions**

Integral expressions are those which produce integer results after implementing all the automatic and explicit type conversions. Examples:

m
m * n – 5
m * 'x'
5 + int(2.0)

where m and n are integer variables.

**Float Expressions**

Float expressions are those which, after all conversions, produce floating-point results. Examples:

x + y
x * y/10
5 + float (10)
10.75

Where x and y are floating point variables.

**Pointer Expressions**

Pointer expressions produce address values. Examples:

&m

ptr

ptr + 1

“xyz”

Where **m** is a variable and **ptr** is a pointer.

**Relational Expressions**

Relational expressions yield results of type **bool** which takes a value **true** or **false**. Examples:

x <= y

a + b == c+d

m + n > 100

When arithmetic expressions are used on either side of a relational operator, they will be evaluated first and then the results compared. Relational expressions are also known as *Boolean expressions*.

**Logical Expressions**

Logical expressions combine two or more relation expressions and produce bool type results. Examples:

a > b && x == 10

x == 10 || y == 5

**Bitwise Expressions**

Bitwise expressions are used to manipulate data at bit level. They are basically used for testing or shifting bits. Examples:

x<< 3 //shift three bit position to left

y >>1 //shift one bit position to right

Shift operators are often used for multiplication and division by powers of two.

ANSI C++ has introduced what are termed as operator keywords that can be used as alternative representation for operator symbol.

**2.20 Special Assignment Expressions**

**Chained Assignment**

```
x = (y = 10);
```

or

```
x = y = 10;
```

First 10 is assigned to y and then to x.

A chained statement cannot be used to initialize variables at the time of declaration. For instance, the statement

```
float a = b = 12.34;          //wrong
```

is illegal. This may be written as

```
float a = 12.34, b = 12.34;   //correct
```

**Embedded Assignment**

```
x = (y = 50) + 10;
```

(y=50) is an assignment expression known as embedded assignment. Here, the value 50 is assigned to y and then the result 50 + 10 = 60 is assigned to x. This statement is

```
y = 50;
x = y + 10;
```

**Compound Assignment**

Like C, C++ supports a *compound assignment* operator which is a combination of the assignment operator with a binary arithmetic operator. For example, the simple assignment statement

```
x = x + 10;
```

may be written as

```
x += 10;
```

The operator += is known as *compound assignment operator* or *short-hand assignment operator*. The general form of the compound assignment operator is:

```
Variable1 op = variable2;
```

where op is a binary arithmetic operator. This means that

```
variable1 = variable1 op variable2;
```

### 2.21 Implicit Conversions

We can mix data types in expression. For example,

```
m = 5 + 2.75;
```

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

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**.

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*.

### 2.22 Operator Overloading

Overloading means assigning different meanings to an operation, depending on the context, C++ permits overloading of operators, thus allowing us to assign multiple meanings to operators. Actually, we have used

the concept of overloading in C also. For example, the operator * when applied to a pointer variable, gives the value pointed to by the pointer. But it is also commonly used for multiplying two numbers. The number and type of operands decide the nature of operation to follow.

The input/output operators << and >> are good examples of operator overloading. Although the built-in definition of the << operator is for shifting of bits, it is also used for displaying the values of various data types. This has been made possible by the header file *iostream* where a number of overloading definitions for << are included. Thus, the statement

```
cout<<75.86;
```

invokes the definition for displaying a **double** type value, and

```
cout<<"well done";
```

invokes the definition for displaying a **char** value. However, none of these definitions in *iostream* affect the built-in meaning of the operator.

Similarly, we can define additional meanings to other C++ operators. For example, we can define + operator to add two *structures* or *objects*. Almost all C++ operators can be overloaded with a few exceptions such as the member-access operators (. and .*), conditional operator (?:) scope resolution operator (::) and the size operator (sizeof).

### 2.23 Operator Precedence

Although C++ enables us to add multiple meanings to the operators, yet their association and precedence remain the same. For example, the multiplication operator will continue having higher precedence than the add operator. The table gives the precedence and associativity of all the C++ operators. The groups are listed in the order of decreasing precedence. The labels prefix and postfix distinguish the uses of ++ and --. Also, the symbols +, - , *, and & are used as both unary and binary operators.

| Operator | Associativity |
|---|---|
| :: | Left to right |
| -> . () [] postfix ++ postfix -- | Left to right |
| prefix ++ prefix -- - ! unary + unary - | Right to left |
| -> * * | Left to right |
| */% | Left to right |
| +- | Left to right |
| << >> | Left to right |
| < <= > >= | Left to right |
| == != | Left to right |
| & | Left to right |
| ^ | Left to right |
| \| | Left to right |
| && | Left to right |
| \|\| | Left to right |
| ?: | Left to right |
| = *= /= %= += = | Right to left |
| << = >> = &= ^= \|= | Left to right |
| , (comma) | Left to right |

### 2.24 Control Structures

In C++, a large number of functions are used that pass messages, and process the data contained in objects. A function is set up to perform a task. When the task is complex, many different algorithms can be designed to achieve the same goal. Some are simple to comprehend, while others are not. Three control structures are :

1. Sequence structure (straight line)
2. Selection structure (branching)
3. Loop structure (iteration or repetition).

The following shows how these structures are implemented using one-entry, one-exit concept, a popular approach used in modular programming.

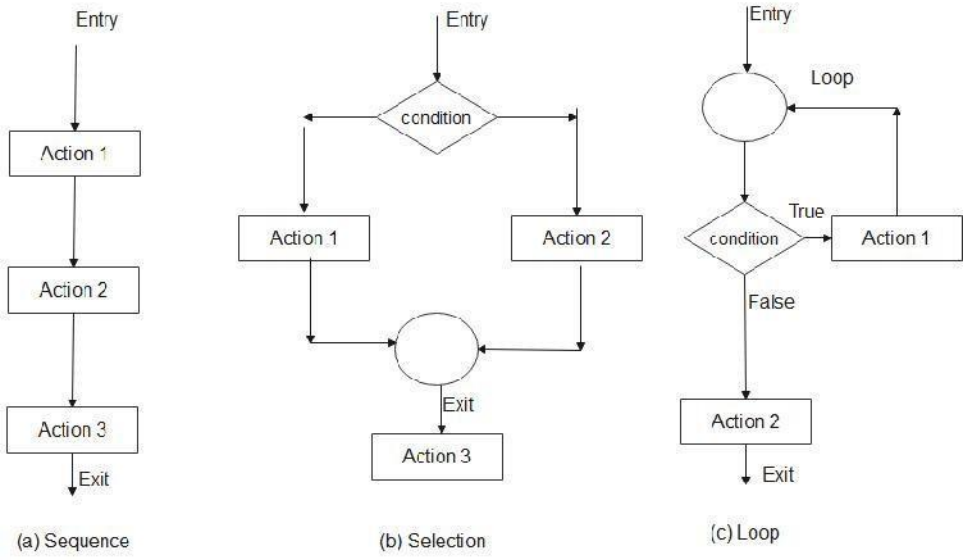



Fig. 2.2 Basic Control Structures

It is important to understand that all program processing can be coded by using only these three logic structures. The approach of using one or more of these basic control constructs in programming is known as *structured programming*, an important technique in software engineering.

Using these three basic constructs, we may represent a function structure either in detail or in summary form as shown above.

Like C, C++ also supports all the three basic control structures, and implements them using various control statements as shown in Fig

This shows that C++ combines the power of structured programming with the object-oriented paradigm.

**The if statement**

The if statement is implemented in two forms:

- Simple **if** statement
- **if ……. else** statement

**Form 1**

```
if (expression is true)
{
action1;
}
action2;
action3;
```

**Form 2**

```
if (expression is true)
{
action1;
}
else
{
action2;
}
action3;
```

**The switch statement**

This is a multiple-branching statement where, based on a condition, the control is transferred to one of the many possible points. This is implemented as follows:

```
switch (expression)
{
case1:
{
action 1;
}
case2:
{
action2;
}
```

```
case3:
{
action3;
}
default:
{
action4;
}
}
action5;
```

**The do-while statement**

The **do-while** is an *exit-controlled loop*. Based on a condition, the control is transferred back to a particular point in the program. The syntax is as follows:

```
do
{
action1;
}while (condition is true);
action2;
```

**The while statement**

This is also a loop structure, but it is an *entry-controlled* one. The syntax is as follows:

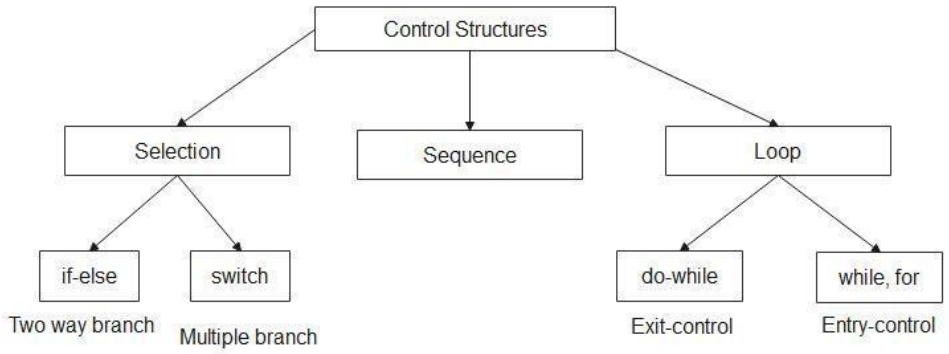
```
while (condition is true)
{
action1;
}
action2;
```

**The for statement**

The **for** is an *entry-controlled* loop and is used when an action is to be repeated for a predetermined number of times. The syntax is as follows:

```
for (initial value; test; increment)
{
action1;
}
action2;
```

The syntax of the control statements in C++ is very much similar to that of C and therefore they are implemented as and when they are required.




C++ statements to implement in two forms

**SUMMARY**

- C++ provides various types of tokens that Include keywords, identifiers, constants, strings, and operators.
- Identifiers refer to the names of variables, functions, arrays, classes, etc.
- C++ provides an additional use of **void**, for declaration of generic pointers.
- The enumerated data types differ slightly in C++. The tag names of the *enumerated* data types become new type names. That is, we can declare new variables using these tag names.

- In C++, the size of character array should be one larger than the number of characters in the string.
- C++ adds the concept of constant pointer and pointer to constant. In case of constant pointer we cannot modify the address that the pointer is initialized to. In case of pointer to a constant, contents of what it points to cannot be changed.
- Pointers are widely used in C++ for memory management and to achieve polymorphism.
- C++ provides a qualifier called **const** to declare named constants which are just like variables except that their values cannot be changed. A **const** modifier defaults to an **int**.
- C++ is very strict regarding type checking of variables. It does not allow to equate variables of two different data types. The only way to break this rule is type casting.
- C++ allows us to declare a variable anywhere in the program, as also its initialization at run time, using the expressions at the place of declaration.
- A reference variable provides an alternative name for a previously defined variable. Both the variables refer to the same data object in the memory. Hence, change in the value of one will also be reflected in the value of the other variable.
- A reference variable must be initialized at the time of declaration, which establishes the correspondence between the reference and the data object that it names.
- A major application of the scope resolution (::) operator is in the classes to identify the class which a member function belongs.
- In addition to malloc(), calloc() and free() functions, C++ also provides two unary operator, **new** and **delete** to perform the task of allocating and freeing the memory in a better and easier way.
- C++ also provides manipulators to format the data display. The most commonly used manipulators are **endl** and **setw**.

- C++ supports seven types of expressions. When data types are mixed in an expression, C++ performs the conversion automatically using certain rules.
- C++ also permits explicit type conversion of variables and expressions using the type cast operators.
- Like C, C++ also supports the three basic control structures namely, sequence, selection, and loop, and implements them using various control statements such as, **if, if...else, switch, do..while, while** and **for**.

**Review Questions:**

1. Enumerate the rules of naming variables in C++, How do they differ from ANSI C rules?
2. An unsigned int can be twice as large as the signed int. Explain how?
3. Why does C++ have type modifiers?
4. What are the applications of **void** data type in C++?
5. Can we assign a **void** pointer to an **int** type painter? If not, why? How can we achieve this?
6. Describe, with examples, the uses of enumeration data types.
7. Describe the differences in the implementation **enum** data type in ANSI C and C++.
8. Why is an array called a derived data type?
9. The size of a **char** array that is declared to store a string should be one larger than the number of characters in the string. Why?
10. The **const** was taken from C++ and incorporated in ANSI C, although quite differently. Explain.
11. How does a constant defined by **const** differ from the constant defined by the preprocessor statement #define?
12. In C++, a variable can be declared anywhere in the scope. What is the significance of this feature?
13. What do you mean by dynamic initialization of a variable? Give an example.

14. What is a reference variable? What is its major use?
15. List at least four new operators added by C++ which aid OOP.
16. What is the application of the scope resolution operator :: in C++?
17. What are the advantages of using **new** operator as compared to the function **malloc()**?
18. Illustrate with an example, how the setw manipulator works.
19. How do the following statements differ?
    (a) char * const p;
    (b) char const *p;

**Exercise Programs:**

1. Write a function using reference variables as arguments to swap the values of a pair of integers.
2. Write a function that creates a vector of user-given size **M** using **new** operator.
3. Write a program to print the following output using **for** loops
   1
   22
   333
   4444
   ……
4. Write a program to evaluate the following functions to 0.0001% accuracy.
   (a) $Sinx = x - x^3/3! + x^5/5! - \ldots$
   (b) $Cosx = 1 - x^2/2! + x^4/4! - \ldots.$
5. Write a program to calculate the variance and standard deviation of N numbers.

# BLOCK – II

## Objectives

After Completion of this block, students will be able to

1. Create user-defined functions
2. Understand the inline functions.
3. Use function overloading.
4. Write “friend” functions to perform operations on different classes

# UNIT – III

# Functions in C++

## 3.1 Introduction

Functions play an important role in C program development. Dividing a program into functions is one of the major principles of top-down, structured programming. Another advantage of using functions is that it is possible to reduce the size of a program by calling and using them at different places in the program.

When a 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.

In this unit, we shall briefly discuss the various new features that are added to C++ functions and their implementation.

## 3.2 The 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 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 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 is any problem. The normal convention is that an exit value of zero means the program ran successfully, while a non-zero value means there was a problem. The explicit use of a **return(0**) statement will indicate that the program was successfully executed.

### 3.3 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 of arguments and the type of return values. With function prototyping, a template is always used 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 exist in the convention C functions.

Note that 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 a 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 accepted at the place of declaration. At this stage, the compiler only checks for the type of 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, it 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, w1, 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. Note that, 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 forgone 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(...);
```

### 3.4 Call by Reference

In traditional 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. Considering the following function:

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

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

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

will exchange the values of m and n using their aliases (reference variables) **a** and **b**. 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 neater in its approach.

### 3.5 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 on 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 illegal and assigns -1 to a if it is larger, otherwise -1 to b.

### 3.6 Inline Functions

One of the objectives of using functions in a program is to save some 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 stack, 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 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);

On the execution of these statements, the values of c and d will be 27 and 64 respectively. If 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 the 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** function diminishes 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);}

Note 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 and 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.

```
//Program to illustrate inline function

#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 the above program would be

121.228

1.25713

### 3.7 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. the 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 of call. One important point to note is that only the trailing arguments can have default values and therefore we must add defaults from *right* to *left*. We cannot provide a default value 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 use only those arguments that are meaningful to a particular situation. Program below illustrates the use of default arguments

```
#include<iostream>

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();                                  //use default values for arguments

amount = value (5000.00, 5);                  // default for 3rd argument

cout<<"n Final value = "<<amount<<"n";

amount = value (10000.00, 5, 0.30);     // pass all arguments explicitly

cout<<"n Final value = "<<amount<<"n";

printline('=');                          //use default value for second 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 (sum);
}
```

```
void printline(char ch, int len)
{
for (int i=1; i<= len; i++)
cout<<ch;
cout<<"\n";
}
```

The output of the program would be

```
**********************************************************
Final value = 10056.2
Final value =  17129.3
==========================================
```

Advantage 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.

### 3.8 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.

### 3.9 Function Overloading

Overloading refers to the use of the same thing for different purposes. C++ also permits overloading of functions. This means that we can use the same function name to create functions that perform a variety of different tasks. This is known as function polymorphism in OOP.

Using the concept of function overloading, we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type. For example, an overloaded **add**() function handles different types of data as shown below:

```
// Declarations
int add(int a. int b);              //prototype 1
int add(int a, int b. int c); //prototype 2
double add (double x. double y); //prototype 3
double add (int p, double q);          //prototype 4
double add(double p,  Int q);          //prototype 5

// Function calls
cout<<add{5, 10);              //uses prototype 1
cout << add(15, 10.0);         //uses prototype 4
cout<<add{12.5, 7.5);          //uses prototype 3
cout<< add(5, 10, 15);         //uses prototype 2
cout<< add{0.75, 5);           // uses prototype 1
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution,

A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same and use that function.
2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

   **char** to **int**

   **float** to **double**

   to find a match.
3. When either of them fails, the compiler tries to use the built-in conversions (the implicit assignment conversions) to the actual arguments and then uses the function whose match is unique, If the conversion is possible to have multiple matches, then the compiler will generate an error message. Suppose we use the following two functions:

   ```
   long square (long n);
   double square (double x);
   ```

   A function call such as

   ```
   square(10);
   ```

   will cause an error because **int** argument can be converted to either **long** or **double**, thereby creating an ambiguous situation as to which version of **square()** should be used.
4. If all of the steps fail, then the compiler will try the user-defined conversions in combination with integral promotions and built-in conversions to find a unique match. User-defined conversions are often used in handling class objects.

The following program illustrates function overloading.

FUNCTION OVERLOADING

```
// Function volume() is overloaded three times

#Include<iostream>

using namespace std;

int volume (int);                    //prototype
double volumne(double, int);         //prototype
long volume (long, int, int);        //prototype

int main()
{

cout<<volume(10)<<"\n";
cout<<volume(2.5,8)<<"\n";
cout<<volume(100L,75,15)<<"\n";

return 0;

}

//function definitions

int volume(int s)
        {
        return(s*s*s);
        }
double volume(double r, int h)
        {
        return(3.1459*r*r*h);
        }
long volume(long l, int b, int h)
        {
        return (l*b*h);
        }
```

The output of program would be:

1000

157.26

112500

Overloading of the functions should be done with caution. We should not overload unrelated functions and should reserve function overloading for functions that perform closely related tasks. Sometimes, the default arguments may be used instead of overloading. This may reduce the number of functions to be defined.

Overloaded functions are extensively used for handling class objects,

### 3.10 Friend and Virtual Functions

C++ introduces two new type of functions, namely, friend function and virtual function. They are basically introduced to handle some specific tasks related to class objects.

### 3.11 Math Library Functions

The standard C++ supports many math functions that can be used for performing certain commonly used calculations. Most frequently used math library functions are summarized below:

| **Functions** | **Purpose** |
|---|---|
| ceil(x) | Rounds x to the smallest integer not less than x<br>Ceil(8.1) = 9 and ceil(-8.8) = -8. |
| cos(x) | Trigonometric cosine of x ( x in radians) |
| exp(x) | Exponential function $e^x$ |
| fabs(x) | Absolute value of x |

| | |
|---|---|
| floor(x) | Rounds x to the largest integer not greater than x |
| log(x) | Natural logarithm of x |
| log10(x) | Common logarithm |
| sin(x) | Trigonometric sine of x |
| sqrt(x) | Square root of x |
| tan(x) | Trigonometric tangent of x |
| pow(x,y) | x raised to power y |

**Note:** The argument variable **x** and **y** are of type **double** and all functions return the data type **double.**

To use the math library functions, we must include the header file **math.h** in conventional C++ and **cmath** in ANSI C++.

**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 ls **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 returnvalues.
- 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 argument. 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.

**Review Questions**

1. What are the advantages of function prototypes in C++?
2. Describe the different styles of writing prototypes.
3. What is the main advantage of passing arguments by reference?
4. When will you make a function **inline** ? Why ?
5. How does an **inline** function differ from a preprocessor macro?
6. When do we need to use default arguments in a function?
7. What is the significance of an empty parenthesis in a function declaration?
8. What do you meant by overloading of a function? When do we use this concept?

**Programming Exercises:**

1. Write a function to read a matrix of size m $\times$ n from the keyboard.
2. Write a program to read a matrix of size m $\times$ n from the keyboard and display the same on the screen using functions.
3. Rewrite the program of Exercise 2 to make the row parameter of the matrix as a default argument.
4. The effect of a default argument can be alternatively achieved by overloading. Discuss with an example.
5. Write a macro that obtains the largest of three numbers.
6. Redo Exercise 5 using inline function. Test the function using a main program.
7. Write a function power() to raise a number **m** to a power **n**. The function takes a **double** value for **m** and **int** value for **n**, and returns the result correctly, Use a default value of 2 for n to make the function to calculate squares when this argument is omitted. Write a **main** that gets the values of m and n from the user to test the function.
8. Write a function that performs the same operation as that of Exercise 7 but takes an **int** value for **m**. Both the functions should have the same name. Write a **main** that calls both the functions. Use the concept of function overloading.

**UNIT – IV**

# Classes and Objects - I

## 4.1 Introduction

The most important feature of C++ is the "class". Its significance is highlighted by the fact that Stroustrup initially gave the name "C with classes" to his new language. A class is an extension of the idea of structure used in C. It is a new way of creating and implementing a user-defined data type. In this unit, we review the concept of structure found in C and then the ways in which classes can be designed, implemented and applied.

## 4.2 C Structures Revisited

We know that one of the unique features of the C language is structures. They provide a method of packing together data of different types. A structure is a convenient tool for handling a group of logically related data items. It is a user-defined data type with a *template* that serves to define its data properties. Once the structure type has been defined, we can create variables of that type using declarations that are similar to the built-in type declarations. For example, consider the following declaration:

```
struct student
{
char name[20];
int roll_number;
float total_marks;
};
```

The keyword **struct** declares **student** as a new data type that can hold three fields of different data types. These fields are known as *structure members* of *elements*. The identifier student, which is referred to as *structure name* or *structure tag*, can be used to create variables of type student.

Example:

```
struct student A;     //C declaration
```

**A** is a variable of type **student** and has three member variables as defined by the template. Member variables can be accessed using the *dot* or *period* operator as follows:

```
strcpy(A.name, "John");

A.roll_number = 999;

A.total_marks = 595.5;

Final_total = A.total_marks + 5;
```

Structures can have *arrays, pointers* or *structures* as members.

**Limitations of C Structure**

The standard C does not allow the *struct* data type to be treated like built-in types. For example, consider the following structure:

```
struct complex

{

float x;
float y;

};

struct complex c1, c2, c3;
```

The complex numbers c1, c2, and c3 can easily be assigned values using the dot operator, but we cannot add two complex numbers or subtract one from the other. For example,

```
c3 = c1 + c2;
```

is illegal in C.

Another important limitation of C structures is that they do not permit data hiding. Structure members can be directly accessed by the structure

variables by any function anywhere in their scope. In other words, the structure members are public members.

**Extensions to Structures**

C++ supports all the features of structures as defined in C. But C++ has expanded its capabilities further to suit its OOP philosophy. It attempts to bring the user-defined types as close as possible to the built-in data types, and also provides a facility to hide the data which is one of the main principles of OOP. *Inheritance*, a mechanism by which one type can inherit characteristics from other types, is also supported by C++.

In C++, a structure can have both variables and functions as members. It can also declare some of its members as 'private' so that they cannot be accessed directly by the external functions.

In C++, the structure names are stand-alone and can be used like any other type names. In other words, the keyword struct can be omitted in the declaration of structure variables. For example, we can declare the student variable A as

```
student A; // C++ declaration
```

Note that this is an error in C.

C++ incorporates all these extensions in another user-defined type known as **class**. There is very little syntactical difference between structures and classes in C++ and, therefore, they can be used interchangeably with minor modifications. Since class is a specially introduced data type in C++, most of the C++ programmers tend to use the structures for holding only data, and classes to hold both the data and functions. Therefore, we will not discuss structures any further.

Note that the only difference between a structure and a class in C++ is that, by default, the members of the classes are *private*, while by default, the members of a structure are *public*.

### 4.3 Specifying a Class

A class is a way to bind the data and its associated functions together. It allows the data (and functions) to be hidden, if necessary, from external use. When defining a class, we are creating a new abstract data type that can be treated like any other built-in data type. Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declarations describes the type and scope of its members. The class function definition describe how the class functions are implemented.

The general form of a class declaration is:

```
class class_name
{
private:
variable  declarations;
function  declarations;
public:
variable  declarations;
function declarations;
}
```

The **class** declaration is similar to a **struct.** The keyword **class** specifies, that what follows is an abstract data of type *class_name*. The body of a class is enclosed within braces, and terminated by a semicolon. The class body contains the declaration of variables and functions. These functions and variables are collectively called *class members*. They are usually grouped under two sections, namely, *private* and *public* to denote which of the members are *private* and which of them are *public*. The keywords **private** and **public** are known as visibility labels. Note that these keywords are followed by a colon.

The class members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. The data hiding (using private declaration) is the key feature of object–oriented programming. The use of the keyword **private** is optional. By default, the members of a class are private. If both the labels are missing, then, by default, all the members are **private**. Such a class is completely hidden from the outside world and does not serve any purpose.

The variables declared inside the class are known as *data members* and the functions are known as *member functions*. Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class. The binding of data and functions together into a single class-type variable is referred to as *encapsulation*.

**A simple Class example**

A typical class declaration would look like:

```
class item
{
int number;                    //variable declaration
float cost;                    // private by default
public:
void getdata(int a, float b);  //functions declaration
void putdata(void);            // using prototype
};                             //ends with semicolon
```

We usually give a class some meaningful name, such as item. This name now becomes a new type identifier that can be used to declare *instances* of that class type. The class item contains two data members and two function members. The data members are private by default while both

the functions are public by declaration. The function **getdata()** can be used to assign values to the member variables number and cost, and **putdata()** for displaying their values. These functions provide the only access to the data members from outside the class. This means that data cannot be accessed by any function that is not a member from outside the class. This means that the data cannot be accessed by any function that is not a member of the class **item**. Note that the functions are declared not defined. Actual function definitions will appear later in the program. The data members are usually declared as **private** and the member functions as **public**.

**Creating Objects**

Note that the declarations of **item** as shown above does not define any objects of **item** but only specifies *what* they will contain. Once a class has been declared, we can create variables of that type by using the class name (like any other built-in type variable). For example,

```
Item x;                    //memory for x is created
```

Create a variable x of type **item**. In C++, the class variables are known as *objects*. Therefore, x is called an object of type **item**. We may also declare more than one object in one statement. Example:

```
Item x, y, z;
```

The declaration of an object is similar to that of a variable of any basic type. The necessary memory space is allocated to an object at this stage. Note that class specification, like a structure provides only a template and does not create any memory space for the *objects*.

Objects can also be created when a class is defined by placing their names immediately after the class braces, as we do in the case of structures.

That is to say, the definition

```
class item
{
……
……
}x, y, z;
```

would create the objects **x, y** and **z** of type **item**. This practice is seldom followed because we would like to declare the objects close to the place where they are used and not at the time of class definition.

**Accessing Class Members**

As pointed out earlier, the private data of a class can be accessed only through the member functions of that class. The **main()** cannot contain statements that *access* **number** and **cost** directly.

The following is the format for calling a member function:

```
Object-name.function-name(actual-arguments);
```

For example, the function call statement

```
x.getdata(100,75.5);
```

is valid and assigns the value 100 to **number** and 75.5 to **cost** of the object **x** by implementing the getdata() function. The assignments occur in the actual function. Similarly, the statement

```
x.putdata();
```

would display the values of data members. Remember, a member function can be invoked only by using an object (of the same class). The statement like

```
getdata(100,75.5);
```

has no meaning.

Similarly, the statement

```
x.number = 100;
```

is also illegal. Although **x** is an object of the type **item** to which **number** belongs, the number (declared private) can be accessed only through a member function and not by the object directly.

It may be recalled that objects communicate by sending and receiving messages. This is achieved through the member functions. For example,

```
x.putdata();
```

send a message to the object **x** requesting it to display its contents.

A variable declared as public can be accessed by the objects directly. Example:

```
class xyz
{
int x;
int y;
public:
int x;
};
…….
……
xyz p;
p.x = 0;                    //error, x is private
p.z = 10;                   //ok. Z is public
……..
……..
```

### 4.4 Defining Member Functions

Member functions can be defined in two places:

- outside the class definition
- inside the class definition

It is obvious that, irrespective of the place of definition, the function should perform the same task. Therefore, the code for the function body would be identical in both the cases. However, there is a subtle difference in the way the function header is defined. Both these approaches are discussed in detail in this section.

**Outside the Class Definition**

Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body. Since C++ does not support the old version of function definition, the ANSI *prototype* form must be used for defining the function header.

An important difference between a member function and a normal function is that a member function incorporates a membership 'identity label' in the header. The 'label' tells the compiler that which class the function belongs to. The general form of a member function definition is:

```
return-type class-name :: function-name(argument declaration)
{
Function body
}
```

The membership label class-name:: tells the compiler that the function function-name belongs to the class class-name. That is, the scope of the function is restricted to the class-name specified in the header line. The symbol :: is called the *scope resolution* operator.

For instance, consider the member-functions **getdata()** and **putdata()** as discussed above. This may be coded as follows:

```
void item :: getdata(int a, float b)
{
number = a;
cost = b;
}
void item::putdata(void)
{
cout<<"number = "<<number<<"\n";
cout<<"cost       ="<<cost<<"\n";
}
```

Since these functions do not return any value, their return-type is void. Function arguments are declared using the ANSI prototype.

The member function have some special characteristics that are often used in the program development. These characteristics are:

- Several different classes can use the same function name. The 'membership label' will resolve their scope.
- Member function can access the private data of the class. A nonmember function cannot do so. (however, an exception to this rule is a friend function).
- A member function can call another member function directly, without using the dot operator.

**Inside the Class Definition**

Another method of defining a member function is to replace the function declaration by the actual function definition inside the class. For example, we could define the **item** class as follows:

```
class item
{
int number;
float cost;
public:
void getdata( int a, float b);          //declaration
//inline function
void putdata(void)                      //definition inside the class
{
cout<<"number = "<<number<<"\n";
cout<<"cost       ="<<cost<<"\n";
}
};
```

When a function is defined inside a class, it is treated as an inline function. Therefore, all the restrictions and limitations that apply to an inline function are also applicable here. Normally, only small functions are defined inside the class definition.

### 4.5 A C++ Program with Class

```
#include<iostream>

using namespace std;

class item

{

int number;              //private by default
float cost;              //private by default

public:

void getdata(int a, float b);      //prototype declaration to be defined

//function defined inside class

void putdata(void)
        {

        cout<<"number: "<< number<<"\n";
        cout<<"cost      : "<< cost<<"\n";

        }

};

// ……. Member function definition ……


void item:: getdata(int a, float b)              //use membership label

{

number = a;              //private variables

cost = b;                //directly used

}
```

```
// ……. Main program ……..
int main()
{
item x;                    // create object x
cout<<"\n object x "<< "\n";
x.getdata(100, 299.95);          //call member function
x.putdata();                     //call member function
item y;                          //create another object
cout<<"\n object Y" << "\n";
y.getdata(200, 175.50);
y.putdata();
return 0;
}
```

This program features the class **item**. This class contain two private variables and two public functions. The member function **getdata()** which has been defined outside the class supplies values to both the variables.
Note the use of statement such as

```
number = a;
```

in the function definition of getdata(). This shows that the member functions can have direct access to private data items.

The member function **putdata()** has been defined inside the class and therefore behaves like an inline function. This function displays the values of the private variables **number** and **cost**.

The program creates two objects, **x** and **y** in two different statements. This can be combined in one statement.

```
item x, y;                    //creates a list of objects
```

Here is the output of the program:

```
Object x
number: 100
cost      : 299.95
object y
number: 200
cost      : 175.5
```

For the sake of illustration only one member function is shown as **inline** and the other as an 'external' member function. Both can be defined as **inline** or external functions.

### 4.6 Making an Outside Function Inline

One of the objectives of OOP is to separate the details of implementation from the class definition. It is therefore a good practice to define the member functions outside the class.

We can define a member function outside the class definition and still make it inline by just using the qualifier inline in the header line of function definition. Example:

```
class item
{
……
……..
public:
void getdata(int a, float b);                    //declaration
};
```

```
inline void item:: getdata(int a, float b)                    //definition
{
number = a;
cost = b;
}
```

### 4.7 Nesting of Member Functions

A member function of a class can be called only by an object of that class using a dot operator. However, there is an exception to this. A member function can be called by using its name inside another member function of the same class. This is known as *nesting of member functions*. The following program illustrates this feature.

```
// Nesting of member functions
        #include<iostream>
        using namespace std;
        class set
        {
        int m,n;
        public:
        void input(void);
        void display (void);
        int largest (void);
        };
        int set largest(void)
        { (m >= n)? return (m) : return (n); }
```

```
void set :: input (void)
{
cout<<"Input values of m and n”<< "\n";
cin>> m>> n;
}
void set :: display(void)
{
cout<<”Largest value = “
        <<largest()<<”\n”;        //calling member function
}
int main()
{
set A;
A.input();
A.display();
return 0;
}
```

### 4.8 Private Member Functions

Although it is normal practice to place all the data items in a private section and all the functions in public. Some situations may require certain functions to be hidden (like private data) from the outside calls. Tasks such as deleting an account in a customer file, or providing increment to an employee are events of serious consequences and therefore the functions handling such tasks should have restricted access. We can place these functions in the private section.

A private member function can only be called by another function that is a member of its class. Even an object cannot invoke a private function using the dot operator. Consider a class as defined below:

```
class sample
{
int k;
void read(void);                    // private member function
public:
void update(void);
void write(void);
};
```

If **s1** is an object of **sample**, then

```
s1.read();        //won't work; objects cannot access private members
```

is illegal. However, the function **read()** can be called by the function **update()** to update the value of **m**.

```
void sample :: update(void)
{
read();           //simple call; no object used
}
```

### 4.9. Arrays within a Class

The arrays can be used as member variables in a class. The following class definition is valid.

```
const int size = 10;                //provides value for array size
class array
{
int a[size];                // 'a' is int type array
public:
void setval(void);
void display(void);
};
```

The array variable **a[ ]** declared as a private member of the class **array** can be used in the member functions, like any other array variable. We can perform any operations on it. For instance in the above class definition, the member function **setval()** sets the values of elements of the array a[ ], and **display()** function displays the values. Similarly, we may use other member functions to perform any other operations on the array values.

## SUMMARY

- A class is an extension to the structure data type. A class can have both variables and functions as members.
- By default, members of the class are private whereas that of structure are public.
- Only the member functions can have access to the private data members and private functions. However the public members can be accessed from outside the class.
- In C++, the class variables are called *objects*. With objects we can access the public members of a class using a dot operator.
- We can define the member functions inside or outside the class. The difference between a member function and a normal function is that a member function uses a membership 'identity' label in the header to indicate the class to which it belongs.

**Review Questions**

1. How do structures in C and C++ differ?
2. What is a class? How does it accomplish data hiding?
3. How does a C++ structure differ from a C++ class?
4. What are objects? How are they created?
5. How is a member function of a class defined?
6. Can we use the same function name for a member function of a class and an outside function, in the same program file? If yes, how are they distinguished? If no, give reasons.
7. Describe the mechanism of accessing data members and member functions in the following cases:
    a) Inside the **main** program.
    b) Inside a member function of the same class.
    c) Inside a member function of another class.

## Programming Exercises

1. Define a class to represent a bank account. Include the following members:

    Data members

        1. Name of the depositor
        2. Account number
        3. Type of account
        4. Balance amount in the account

    Member functions

        1. To assign initial values
        2. To deposit an amount
        3. To withdraw an amount after checking the balance
        4. To display name and balance

        Write a main program to test the program.
2. Modify the class and the program of Exercise 1 for handling 10 customers.

# BLOCK – III

## Objectives

After Completion of this block, students will be able to

1. Get clear picture of arrays
2. Understand how memory is allocated to objects
3. Write moderate level programs using Object concept

# UNIT – V

# Classes and Objects - II

## 5.1 Memory Allocation for Objects

Memory space for objects is allocated when they are declared and not when the class is specified. This statement is only partly true. Actually, the member functions are created and placed in the memory space only once when they are defined as a part of a class specification. Since all the objects belonging to that class use the same member functions, no separate space is allocated for member functions when the objects are created. Only space for member variables is allocated separately for each object. Separate memory locations for the objects are essential, because the member variables will hold different data values for different objects.

## 5.2 Static Data members

A data member of a class can be qualified as static. The properties of a **static** member variable are similar to that of a C static variable. A static member variable has certain special characteristics. These are:

- It is initialized to zero when the first object of its class is created. No other initialization is permitted.
- Only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created
- It is visible only within the class, but its lifetime is the entire program.

Static variables are normally used to maintain values common to the entire class. For example, a static data member can be used as a counter that records the occurrences of all the objects. The following program illustrates the use of a static data member.

```
//Static class member
#include<iostream>
using namespace std;
class item
{
static int count;
int number;
public:
void getdata(int a)
        {
        number = a;
        count++;
        }
void getcount(void)
    {
    cout<<"count: ";
    cout<<count<<"\n";
    }
};

int item:: count;
int main()
{
Item a, b, c;          //count is initialized to zero
a.getcount();          //display count
b.getcount();
c.getcount();
```

```
a.getdata(100);              //getting date into object a
b.getdata(200);              //getting date into object b
c.getdata(300);              //getting date into object c
cout<<"After reading data"<<"\n";

a.getcount();                //display count
b.getcount();
c.getcount();
return 0;
}
```

The output of the program would be

```
count: 0
count: 0
count: 0

After reading data
count: 3
count: 3
count: 3
```

Notice the following statement in the program

```
int item:: count;                    //definition of static data member
```

Note that the type and scope of each static member variable must be defined outside the class definition. This is necessary because the static data members are stored separately rather than as a part of an object. Since they are associated with the class itself rather than with any class object, they are also known as class variables.

The static variable count is initialized to zero when the objects are created. The count is incremented whenever the data is read into an object. Since the data is read into objects three times, the variable count is incremented three times. Because there is only one copy of count shared by all the three objects, all the three output statements cause the value 3 to be displayed.

Static variables are like non-inline functions as they are declared in a class declaration and defined in the source file. While defining a static variable, some initial value can also be assigned to the variable. For instance, the following definition gives count the initial value 10.

```
int item:: count = 10;
```

### 5.3 Static Member functions

Like static member variable, we can also have static member functions. A member function that is declared static has the following properties:

- A static function can have access to only other static members (functions or variables) declared in the same class
- A static member function can be called using the class name (instead of its objects) as follows:

```
class-name:: function-name;
```

The following program illustrates the implementation of these characteristics. The static function showcount() displays the number of objects created till that moment. A count of number of objects created is maintained by the static variable count.

The function **showcode()** displays the code number of each object.

```
//Static member function
#include<iostream>
using namespace std;

class test
{
int code;
static int count;                    //static member variable
public:
void setcode(void)
                    {
                    code=++count;
                    }
void showcode(void)
                    {
                    cout<<"object number: "<<code <<"\n";
                    }

static void showcount(void)                    //static member function
                    {
                    cout<<"count: "<<count<<"\n";
                    }
};

int test:: count;
```

```
int main()
{
test t1, t2;
t1.setcode();
t2.setcode();
test::showcount();                    //accessing static function
test t3;
t3.setcode();
test::showcount();
t1.showcode();
t2.showcode();
t3.showcode();

return 0;
}
```

The output of program would be:

```
count: 2
count: 3

object number: 1
object number: 2
object number: 3
```

Note that the statement

```
code=+=count;
```

is executed whenever **setcode**() function is invoked and the current value of count is assigned to **code**. Since each object has its own copy of code, the value contained in **code** represents a unique number of its object.

Note that the following definition will not work:

```
static void showcount()
{
cout<<code;              //code is not static
}
```

### 5.4 Array of Objects

We know that an array can be of any data type including **struct**. Similarly, we can also have arrays of variables that are of the type class. Such variables are called *arrays of objects*. Consider the following class definition:

```
class employee
{
char name[30];
float age;
public:
void getdata(void);
void putdata(void);
};
```

The identifier **employee** is a user-defined data type and can be used to create objects that relate to different categories of the employees. Example:

```
employee manager[3];          //array of manager
employee foreman[15];         //array of foreman
employee worker[75];          //array of worker
```

The array manager contains three objects (managers), namely, **manager[0], manager[1]**, and **manager[2]**, of type **employee** class. Similarly, the **foreman** array contains 15 objects (foreman) and the **worker** array contains 75 objects(workers).

Since an array of objects behaves like any other array, we can use the usual array-accessing method to access individual elements, and then the dot member operator to access the member functions. For example, the statement

```
manager[i].putdata();
```

Will display the data of the ith element of the array **manager**. That is, this statement request the object **manager[i]** to invoke the member function **putdata()**.

An array of objects is stored inside the memory in the same way as a multi-dimensional array. Note that only the space for data items of the objects is created. Member functions are stored separately and will be used by all the objects.

```
//Array of objects
#include<iostreamh>
using namespace std;

class employee
{
char name[30];          //string as class member
float age;
```

```
public:
void getdata(void);
void putdata(void);
};
void employee::getdata(void)
{
cout<<"Enter name: "; cin>>name;
cout<<"Enter age: "; cin>>age;
}
void employee:: putdata(void)
{
cout<<"Name: "<< name<<"\n";
cout<<"Age :"<<age<<"\n";
}
const int size = 3;
int main()
{
employee manager[size];
for (int i=0; i< size; i++)
        {
        cout<<"\n Details of manager"<< i+1<<"\n";
        manager[i].getdata();
        }
        cout<<"\n";
        for (i=0; i< size; i++)
                {
                cout<<"\n Manager"<< i+1 <<"\n";
                manager[i].putdata();
                }
return 0;
}
```

This being an interactive program, the input data and the program output are shown below:

Interactive input

Details of manager 1:

Enter name: xxx

Enter age : 45

Details of manager 2:

Enter name: yyy

Enter age : 55

Details of manager 3:

Enter name: zzz

Enter age : 65

Program output

Manager 1:
Name: xxx
Age : 45

Manager 2:
Name: yyy
Age : 55

Manager 3:
Name: zzz
Age : 65

### 5.5 Objects as Function Arguments

Like any other data type, an object may be used as function argument. This can be done in two ways:

- A copy of the entire object is passed to the function
- Only the address of the object is transferred to the function.

The first method is called pass-by-value. Since a copy of the object is passed to the function, any changes made to the object inside the function do not affect the object used to call the function. The second method is called pass-by-reference. When an address of the object is passed, the called function works directly on the actual object used in the call. This means that any changes made to the object inside the function will reflect in the actual object. The pass-by-reference method is more efficient since it requires to pass only the address of the object and not the entire object.

The following program illustrates the use of objects as function arguments. It performs the addition of time in the hour and minute format.

```
//Objects as arguments

#include<iostream>

using namespace std;

class item
{

int hours;
int minutes;

public:
void gettime(int h, int m)

                {

                hour = h; minutes = m;

                }
```

```
void puttime(void)
{
cout<<hours<<"hours and ";
cout<<minutes<<"minutes"<<"\n";
}
void sum(time, time);            //declaration with objects as arguments
};
void time:: sum(time t1, time t2)                    //t1, t2 are objects
{
minutes = t1.minutes + t2.minutes;
hours = minutes/60;
minutes = minutes % 60;
hours= hours + t1.hours + t2.hours;
}
int main()
{
time T1, T2, T3;
T1.gettime(2, 45);               //get T1
T2.gettime(3, 30);               //get                    T2
T3.sum(T1,T2);                          // T3 = T1 + T2
cout<< "T1 = "; T1.puttime();            //display  T1
cout<<"T2 = "; T2.puttime();             //display  T2
cout<<"T3 = "; T3.puttime();             //display T3
return 0;
}
```

The output of the program would be

T1 = 2 hours and 45 minutes

T2 = 3 hours and 30 minutes

T3 = 6 hours and 15 minutes

Since the member function sum() is invoked by the object T3, with the objects T1 and T2 as arguments. It can directly access the hours and minutes variables of T3. But, the member of T1 and T2 can be accessed only by using the dot operator (like T1.hours and T1.minutes). Therefore, inside the function sum(), the variables hours and minutes refer to T3, T1.hours and T1.minutes refers to T1 and T2.hours and T2.minutes refer to T2.

An object can also be passed as an argument to a non-member function. However, such functions can have access to the public member functions only through the objects passed as arguments to it. These functions cannot have access to the private data members.

### 5.6 Friendly Functions

We know that private members cannot be accessed from outside the class. That is, a non-member function cannot have an access to the private data of a class. However, there could be a situation where we would like two classes to share a particular function. For example, consider a case where two classes, **manager** and **scientist**, have been defined. We would like to use a function **income_tax()** to operate on the objects of both these classes. In such situations, C++ allows the common function to be made friendly with both the classes. In such situations, C++ allows the common function to be made friendly with both the classes, thereby allowing the function to have access to the private data of these classes. Such a function need not be a member of any of these classes.

To make an outside function **"**friendly**"** to a class, we have to simply declare this function as a **friend** of the class as shown below:

```
class ABC
{
……..
…….
public:
……..
……
friend void xyz(void); //declaration
};
```

The function declaration should be preceded by the keyword **friend**. The function is defined elsewhere in the program like a normal C++ function. The function definition does not use either the keyword **friend** or the scope resolution operator ::. The functions that are declared with the keyword **friend** are known as friend functions. A function can be declared as a **friend** in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class.

A friend function possesses certain special characteristics:

- It is not in the scope of the class to which it has been declared as **friend**.
- Since it is not in the scope of the class, it cannot be called using the object of that class.
- It can be invoked like a normal function without the help of any object.
- Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name (e.g. A.x).

- It can be declared either in the public or the private part of a class without affecting its meaning.
- Usually, it has the objects as arguments.

```
//Friend function

#include<iostream>

using namespace std;

class sample

{

int a;
int b;

public:
void setvalue() {a = 25; b = 40;}
friend float mean (sample s);
};

float mean (sample s)
                {
                return float(s.a + s.b)/2.0;
                }

int main()

{

sample x;               // object x

x.setvalue();

cout<<"mean value = "<<mean(x)<<"\n";

return 0;

}
```

The output of the program would be

mean value = 32.5

The friend function access the class variables **a** and **b** by using the dot operator and the object passed to it. The function call **mean(x)** passes the object **x** by value to the friend function.

Member functions of one class can be **friend** of another class. In such cases, they are defined using the scope resolution operator as shown below:

```
class x
{
……………
……..
int fun1();                    //member function of x
};
class y
{
……..
…….
friend int x:: fun1();                  // func1() of X is a friend of Y
……..
};
```

The function fun1() is a member of class x and a friend of class y.

We can also declare all the member functions of one class as the friend functions of another class. In such cases, the class is called a **friend class**. This can be specified as follows:

```
class z
{
……
  friend class x; // all member functions of x are friends to  z
};
```

```
//Program using friend function to add data objects of two different classes
/* use of friend function to access members of two classes*/
#include<iostream>
using namespace std;
class ABC;                  //forward declaration
class XYZ
{
int x;
public:
void setvalue(int i)
{ x = i; }
friend void max(XYZ, ABC);
};
class ABC
{
int a;
public:
void setvalue(int i) { a = i;}
friend void max(XYZ, ABC);
};
void max(XYZ m, ABC n)
{ (m.x >= n.a) ? cout<<m.x : cout<<n.a; }
int main()
{
ABC abc;
abc.setvalue(10);
XYZ xyz;
```

```
xyz.setvalue(20);
max(xyz, abc);
return 0;
}
```

The output of program would be:

20.

The function max() has arguments from both XYZ and ABC. When the function max() is declared as a friend in XYZ for the first time, the compiler will not acknowledge the presence of ABC unless its name is declared in the beginning as class ABC; This is known as 'forward" declaration.

As pointed out earlier, a friend function can be called by reference. In this case, local copies of the objects are not made. Instead, a pointer to the address of the object is passed and the called function directly works on the actual object used in the call.

This method can be used to alter the values of the private members of a class. Note that altering the values of private members is against the basic principles of data hiding. It should be used only when absolutely necessary.

The following program shows how to use a common friend function to exchange the private values of two classes. The function is called by reference.

```
//Program for swapping private data of classes
#include<iostream>
using namespace std;
class class2;
```

```
class class1
{
int value1;
public:
void indata(int a) { value1 = a;}
void display(void) { cout<< value1 <<"\n";}
friend void exchange (class1&, class2&);
};
class class2
{
int value2;

public:
{
void indata(int a) { value2 = a;}
void display(void) { cout<< value2 <<"\n";}
friend void exchange (class1&, class2&);
};
void exchange(class1& x, class2& y)
{
int temp = x.value1;
x.value1 = y.value2;
y.value2 = temp;
}
int main()
{
class1 c1;
class2 c2;
c1.indata(100);
c2.indate(200);
cout<<"Values before exchange"<<"\n";
c1.display();
c2.display();
```

```
exchange(c1, c2);       //swapping
cout<<"values after exchange"<<"\n";
c1.display();
c2.display();

return 0;
}
```

The objects x and y are aliases of c1 and c2 respectively. The statements

```
int temp = x.value1;

x.value1 = y.value2;

y.value2 = temp;
```

directly modify the values of **value1** and **value2** declared in class1 and class2.

The output of program would be:

```
Values before exchange

100

200


Values after exchange

200

100
```

### 5.7 Returning objects

A function cannot only receive objects as arguments, but also can return them. The example in the following program illustrates how an object can be created (within a function) and returned to another function.

```
#include<iostream>

using namespace std;

class complex
{
float x;          //real part
float y;          //imaginary part
public:
void input(float real, float imag)
{x=real, y = imag;}
friend complex sum(complex, complex);
void show(complex);
};
complex sum (complex c1, complex c2)
                  {
                  complex c3;
                  c3.x = c1.x + c2.x;
                  c3.y = c1.y + c2.y;
                  return (c3);
                  }
void complex: show(complex c)
                  { cout<<c.x <<" +j ("<< c.y <<")\n"; }
int main()
{
complex A, B, C;
A.input(3.1, 5.65);
B.input(2.75, 1.2);
C = sum(A, B);
```

```
cout<<"A = "; A.show(A);
cout<<"B = "; B.show(B);
cout<<"C = "; C.show(C);
return 0;
}
```

Upon execution, the above program would generate the following output:

A = 3.1 + j (5.65)

B = 2.75 + j (1.2)

C = 5.85 + j (6.85)

The program adds two complex numbers **A** and **B** to produce a third complex number **C**, and displays all the three numbers.

### 5.8 Const Member Function

If a member function does not alter any data in the class, then we may declare it as a const member function as follows:

```
void mul(int, int) const;
double get_balance() const;
```

The qualifier **const** is appended to the function prototypes (in both declaration and definition). The compiler will generate an error message if such functions try to alter the data values.

### 5.9 Pointer to Members

It is possible to take the address of a member of a class and assign it to a pointer. The address of a member can be obtained by applying the operator & to a "fully qualified" class member name. A class member pointer can be declared using the operator ::* with the class name.

For example, given the class:

```
class A
{
private:
        int m;
public:
void show();
};
```

We can define a pointer to the member m as follows:

```
Int A::* ip = &A:: m;
```

The **ip** pointer created thus acts like a class member in that it must be invoked with a class object. In the statement above, the phrase **A::*** means "pointer-to-member of A class". The phrase **&A::m** means the "address of the m member of A class".

Note that the following statement is not valid:

```
int *ip = &m;          //won't work
```

This is because **m** is not simply an **int** type data. It has meaning only when it is associated with the class to which it belongs. The scope operator must be applied to both the pointer and the member.

The pointer **ip** can now be used to access the member **m** inside member functions (or friend functions). Let us assume that **a** is an object of **A** declared in a member function. We can access **m** using the pointer **ip** as follows:

```
cout<<a.*ip;          //display
cout<<a.m;            //same as above
```

Now, look at the following code:

```
ap = &a;                    //ap is a pointer to object a
cout<<ap -> *ip;            //display m
cout<<ap -> m;              //same as above
```

The dereferencing operator ->* is used to access a member when we use pointers to both the object and the member. The dereferencing operator .* is used when the object itself is used with the member pointer. Note that ***ip** is used like a member name.

We can also design pointers to member functions when, then, can be invoked using the dereferencing operators in the **main** as shown below:

```
(object-name .* pointer-to-member function) (10);
(pointer-to-object ->* pointer-to-member function) (10)
```

The precedence of () is higher than that of .* and ->*, so the parentheses are necessary.

/*Program to illustrate the use of dereferencing operators to access the class members.*/

```
#include<iostream>
using namespace std;
class M
{
int x;
int y;
public:
void set_xy(int a, int b)
                { x=a; y = b; }
friend int sum(M m);
};
```

```
int sum (M m)
{
int M ::* px = &M :: x;
int M ::* py = &M :: y;
M *pm = &m;
int S = m.*px + pm->*py;
return S;
}
int main()
{
M n;
void (M :: *pf)(int, int) = &M :: set_xy;
(n.*pf)(10, 20);
cout<<"Sum ="<< sum(n)<<"\n";
M *op = &n;
(op -> *pf)(30, 40);
cout<< "Sum = "<< sum(n)<<"\n";
return 0;
}
```

The output of the above program would be:

```
Sum = 30
Sum = 70
```

### 5.10 Local Classes

Classes can be defined and used inside a function or a block. Such classes are called local classes.

Example:

```
void test (int a)                    //function
{
……….
……….
        class student          //local class
                {
                ……
                ……             //class definition
                ……
};
……
……..
student s1(a);                       //create student object
……..                                // use student object

}
```

Local classes can use global variables (declared above the function) and static variables declared inside the function but cannot use automatic local variables. The global variables should be used with the scope operator (::).

There are some restrictions in constructing local classes. They cannot have static data members and member functions must be defined inside the local classes. Enclosing function cannot access the private members of a local class. However, we can achieve this by declaring the enclosing function as a friend.

```
//Program for function returning objects
#include<iostream>
using namespace std;
class matrix
{
int m[3][3];
public:
void read(void)
        {
        cout<<"Enter the elements of the 3 × 3 matrix:\n";
        int i,j;
        for (i=0; i < 3; i++)
                for (j=0; j <3; j++)
                        {
                        cout<<"m["<<i<<"]["<<j<<"]";
                        cin>>m[i][j];
                        }
        }
void display (void)
        {
        int i,j;
        for (i=0; i < 3; i++)
                {
                cout<<"\n";
                for (j=0; j <3; j++)
                        {
                        cout<<m[i][j]<<"\t";
                        }
                }
        }
```

```
friend trans(matrix x1)
        {
        matrix m2;              //creating an object
        int i,j;
        for (i=0; i < 3; i++)
                for (j=0; j <3; j++)
                        m2.m[i][j] = m1.m[j][i];
        return (m2);            //returning an object
        }
int main()
{
matrix mat1, mat2;
mat1.read();
cout<<"\n You entered the following matrix:";
mat1.display();

mat2 = trans(mat1);
cout<<"\n Transposed matrix:";
mat2.display();
return 0;
}
```

The output of the program would be as follows:

Enter the elements of the 3× 3 matrix:

m[0][0] = 1

m[0][1] = 2

m[0][2] = 3

m[1][0] = 4

m[1][1] = 5

m[1][2] = 6

m[2][0] = 7

m[2][1] = 8

m[2][2] = 9

You entered the following matric:

1 2 3

4 5 6

7 8 9

Transposed matrix:

1 4 7

2 5 8

3 6 9

The program finds the transpose of a given $3 \times 3$ matrix and stores it in a new matrix object. The display member function displays the matrix elements;

## SUMMARY

- The memory space for the objects is allocated when they are declared, Space for member variables is allocated separately for each object, but no separate space is allocated for member functions.
- A data member of a class can be declared as a **static** and is normally used to maintain values common to the entire class.
- The **static** member variables must be defined outside the class.
- A static member function can have access to the static members declared in the same class and can be called using the class name.
- C++ allows us to have arrays of objects.
- We may use objects as function arguments.

- A function declared as a **friend** is not in the scope of the class to which it has been declared as friend. It has full access to the private members of the class.
- A function can also return an object.
- If a member function does not alter any data in the class, then we may declare it as a **const** member function. The keyword **const** is appended to the function prototype.
- It is also possible to define and use a class inside a function. Such a class is called a local class.

**Review Questions**

1. When do we declare a member of a class **static**?
2. What is a friend function? What are the merits and demerits of using friend functions?
3. State whether the following statements are TRUE or FALSE.
    a) Data items in a class must always be private.
    b) A function designed as private is accessible only to member functions of that class.
    c) A function designed as public can be accessed like any other ordinary functions.
    d) Member functions defined inside a class specifier become inline functions by default.
    e) Classes can bring together all aspects of an entity in one place.
    f) Class members are public by default.
    g) Friend functions have access to only public members of a class.
    h) An entire class can be made a friend of another class.
    i) Functions cannot return class objects.
    j) Data members can be initialized inside class specifier.

**Programming Exercises**

1. Create two classes DM and DB which store the value of distances. DM stores distances in meters and centimeters and DB in feet and inches. Write a program that can read values for the class objects and add one object of DM with another object of DB.

   Use a friend function to carry out the addition operation. The object that stores the results may be a DM object or DB object, depending on the units in which the results are required.

   The display should be in the format of feet and inches or meters and centimeters depending on the object on display.

**UNIT – VI**

# Constructors and Destructors

## 6.1 Introduction

We have seen, so far, a few examples of classes being implemented. In all the cases, we have used member functions such as **putdata()** and **setvalue()** to provide initial values to the private member variables. For example, the following statement

A.input();

Invokes the member function input(), which assigns the initial values to the data item of object **A**. Similarly, the statement

x.getdata(100,299.95);

passes the initial values as arguments to the function **getdata()**, where these values are assigned to the private variables of object **x**. All these "function call" statements are used with the appropriate objects that have already been created. These functions cannot be used to initialize the member variables at the time of creation of their objects.

Providing the initial values as described above does not conform with the philosophy of C++ language. The aim of C++ is to create user-defined data types such as **class** that behave very similar to the built-in types. This means that we should be able to initialize a **class** type variable (object) when it is declared, much the same way as initialization of ordinary variable. For example,

int m = 20;

float x = 5.75;

are valid initialization statements for basic data types.

Similarly, when a variable of built-in type goes out of scope, the compiler automatically destroys the variables. But, it has not happened with the objects we have so far studied. It is therefore clear that some more features of classes need to be explored that would enable us to initialize the objects when they are created and destroy them when their presence is no longer necessary.

C++ provides a special member function called the *constructor* which enables an object to initialize itself when it is created. This is known as *automatic initialization* of objects. It also provides another member function called the *destructor* that destroys the objects when they are no longer required.

### 6.2 Constructors

A constructor is a 'special' member function whose task is to illustrate the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

A constructor is declared and defined as follows:

```
//class with a constructor

class integer
{
int m,n;
public:
integer (void);                 //constructor declared
.............
.............
};
integer :: integer (void)       //constructor defined
{ m = 0; n = 0; }
```

When a class contains a constructor like the one defined above, it is guaranteed that an object created by the class will be initialized automatically. For example, the declaration

```
integer int1;                    // object int1 created
```

not only creates the object int1 of type **integer** but also initializes its data members **m** and **n** to zero. There is no need to write any statement to invoke the constructor function (as we do with the normal member functions). If a 'normal' member function is defined for zero initialization, we would need to invoke this function for each of the objects separately. This would be very inconvenient, if there are a large number of objects.

A constructor that accepts no parameters is called the default constructor. The default constructor for class A is A:: A(). If no such constructor is defined, then the compiler supplies a default constructor. Therefore a statement such as

```
A a;
```

Invokes the default constructor of the compiler to create the object a.

The constructor functions have some *special characteristics*. These are:

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, and they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be virtual.
- An object with a constructor (or destructor) cannot be used as a member of a union.

- They make ‘implicit calls’ to the operators new and delete when memory allocation is required.

Note that when a constructor is declared for a class, initialization of the class objects becomes mandatory.

### 6.3 Parameterized Constructors

The constructor integer(), defined above, initializes the data members of all the objects to zero. However, in practice it may be necessary to initialize the various data elements of different objects with different values when they are created. C++ permits us to achieve this objective by passing arguments to the constructor function when the objects are created. The constructors that can take arguments are called *parameterized constructors*.

The constructor integer() may be modified to take arguments as shown below:

```
class integer
{
int m,n;
public:
integer (int x, int y);              //parameterized constructor
.............
.............
};
integer :: integer (int x, int y)
{ m = x; n = y; }
```

When a constructor has been parameterized, the object declaration statement such as

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared.

This can be done in two ways:

- By calling the constructor explicitly.
- By calling the constructor implicitly.

The following declaration illustrates the first method:

```
integer int1 = integer(0, 100);          //explicit call
```

This statement creates an integer int1 and passes the values 0 and 100 to it. The second is implemented as follows:

```
integer int1(0,100);                     //implicit call
```

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement.

Note that when the constructor is parameterized, we must provide appropriate arguments for the constructor. The following program demonstrates the passing of arguments to the constructor functions.

```
//class with constructors

#include<iostream>

using namespace std;

class integer
{
int m,n;
public:
integer (int x, int y);              //constructor declared
void display(void)
{
cout<<"m = "<<m<<"\n";
cout<<"n = "<<n<<"\n";
}
};
```

```
integer :: integer (int x, int y)          //constructor defined
{ m = x; n = y; }

int main()
{
integer int1(0,100);                       //constructor called implicitly
integer int2 = integer(25, 75);            //constructor called explicitly
cout<< "\nObject1 "<<"\n";
int1.display();
cout<< "\nObject2 "<<"\n";
int2.display();
return 0;
}
```

The output of the above program would be:

```
Object1

m = 0
n = 100

Object2

m = 25
n = 75
```

The constructor functions can also be defined as inline functions. Example:

```
class integer
{
int m,n;
public:
integer (int x, int y)              //Inline constructor
{ m = x; n = y; }
……….
……….
};
```

The parameters of a constructor can be of any type except that of the class to which it belongs. For example,

```
class A
{
……..
……..
public:
A(A);
};
```

Is illegal.

However, a constructor can accept a reference to its own class as a parameter. Thus, the statement

```
class A
{
……..
……..
public:
A(A&);
};
```

Is valid. In such cases, the constructor is called the *copy constructor*.

### 6.4 Multiple Constructors in a Class

So far we have used two kinds of constructors. They are:

```
integer();              //no arguments
integer(int, int);      //two arguments
```

In the first case, the constructor itself supplies the data values and no values are passed by the calling program. In the second case, the function call passes the appropriate values from **main()**. C++ permits us to use both these constructors in the same class.

For example, we could define a class as follows:

```
class integer
{
int m,n;
public:
integer () ( m =0; n=0;)          //constructor 1
integer (int a, int b)
{m = a; n = b;}                   //constructor 2
Integer (integer &i)
{m=i.m; n = i.n;}                 //constructor 3
};
```

This declares three constructors for an **integer** object. The first constructor receives no arguments, the second receives two **integer** arguments and the third receives one integer object as an argument. For example, the declaration

```
integer I1;
```

would automatically invoke the first constructor and set both **m** and **n** of **I1** to zero. The statement

```
integer I2(20, 40);
```

would call the second constructor which will initialize the data members **m** and **n** of **I2** to **20** and **40** respectively. Finally, the statement

```
integer I3(I2);
```

would invoke the third constructor which copies the values of **I2** into **I3**. In other words. It sets the value of every data element of **I3** to the value of the corresponding data element of **I2**. As mentioned earlier, such a constructor is called the *copy constructor*. The process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor function is defined in a class, we say that the constructor is overloaded.

```
//Program for Overloaded Constructor
#include<iostream>
using namespace std;
class complex
{
float x, y;
public:
complex() {  }                          //constructor no arg
complex(float a) {x = y = a;}           //constructor-one arg
complex (float real, float imag)        //constructor-two arg
{ x = real; y = imag;}
friend complex sum(complex, complex);
friend void show(complex);
};
complex sum(complex c1, complex c2)                 //friend
{
complex c3;
c3.x = c1.x + c2.x;
c3.y = c1.y + c2.y;
return (c3);
}
void show (complex c)
{ cout<<c.x <<" + j (" << c.y << ")\n"; }
```

```
int main()
{
complex A(2.7, 3.5);          //define & initialize
complex B(1.6);               //define & initialize
compelx C;                    //defined
C = sum(A, B);                      //sum() is a friend
cout<<"A = "; show(A);              //show() is also friend
cout<<"B = "; show(B);
cout<<"C = "; show(C);

//Another way to give initial values (second method)
complex P,Q,R;                //define P, Q and R
P = complex(2.5, 3.9);        //initialize  P
Q = complex(1.6, 2.5);        //initialize  Q
R = sum(P, Q);

cout<<"\n";
cout<<"P = "; show(P);
cout<<"Q = "; show(Q);
cout<<"R = "; show(R);
return 0;
}
```

The output of program would be:

A = 2.7 + j (3.5)
B = 1.6 + j (1.6)
C = 4.3 + j (5.1)

P = 2.5 + j (3.9)

Q = 1.6 + j (2.5)

R = 4.1 + j (6.4)

There are three constructors in the class **complex**. The first constructor, which takes no arguments, is used to create objects which are not initialized; the second, which takes one argument, is used to crate objects and initialize them; and the third which takes two arguments, is also used to create objects and initialize them to specific values. Note that the second method of initializing values looks better.

Let us see the first constructor again.

complex() { }

It contains the empty body and does not do anything. We just stated that this is used to create objects without any initial values. Note that C++ compiler has an *implicit constructor* which creates objects, even though it was not defined in the class.

This works fine as long as we do not use any other constructors in the class. However, once we define a constructor, we must also define the "do-nothing" implicit constructor. This constructor will not do anything and is defined just to satisfy the compiler.

### 6.5 Constructors with Default Arguments

It is possible to define constructors with default arguments. For example, the constructor complex() can be declared as follows:

complex(float real , float imag = 0);

The default value of the argument **imag** is zero.

Then, the statement

complex C(5.0);

Assigns the value **5.0** to **real** variable and **0.0** to **imag** (by default).

However, the statement

complex C(2.0, 3.0);

assigns **2.0** to **real** and **3.0** to **imag**. The actual parameter, when specified, overrides the default value. As pointed out earlier, the missing arguments must be the trailing ones.

It is important to distinguish between the default constructor **A::A()** and the default argument constructor **A::A(int = 0)**. The default argument constructor can be called with either one argument or no arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

A a;

The ambiguity is whether to 'call' **A::A()** or **A::A(int = 0)**.

**SUMMARY**

- C++ provides a special member function called the constructor which enables an object to initialize itself when it is created, This is known as *automatic initialization* of objects.
- A constructor has the same name as that of a class.
- Constructors are normally used to initialize variables and to allocate memory.
- Similar to normal functions, constructors may be overloaded.

**Review Questions**

1. What is a constructor? Is it mandatory to use constructor in a class?
2. How do we invoke a constructor function
3. List some of the special properties of the constructor functions.

4. What is a parameterized constructor?
5. Can we have more than one constructors in a class? If yes, explain the need for such a situation.

**Programming Exercises:**

Write a complete program to test your class to see that it does the following tasks:

(a) Create uninitialized string objects

(b) Creates objects with string constants

(c) Concatenates two strings properly.

(d) Displays a desired string object.

# BLOCK – IV

## Objectives

After Completion of this block, students will be able to

1. Initialize objects dynamically.
2. Use constructors and Destructors
3. Apply operator overloading concept

# UNIT – VII

# CONSTRUCTORS AND DESTRUCTORS

### 7.1 Dynamic Initialization of Objects

Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during run time. One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different formats of data at run time depending upon the situation.

Consider the long term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment, The following program illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

```
//Dynamic initialization of Constructors

//long-term fixed deposit system

#include<iostream>

using namespace std;

class fixed_deposit
        {
        long int P_amount;              // Principal amount
        int Years;                      //period of investment
        float Rate;                     // Interest rate
        float R_value;                  // Return value
        public:
        fixed_deposit () { }
        fixed_deposit (long int p, int y, float r=0.12);
```

```
        fixed_deposit (long int p, int y, int r);
        void display(void);
        };
fixed_deposit :: fixed_deposit (log int p, int y, float r)
        {
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;
        for (int i = 1; i <= y; i++)
                R_value = R_value *(1.0 + r);
        }
fixed_deposit :: fixed_deposit (long int p, int y, int r)
        {
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;
        for (int i = 1; i <= y; i++)
                R_value = R_value *(1.0 +float (r)/100);
        }
void fixed_deposit :: display()
        {
        cout<< "\n"<<Principal Amount ="<< P_amount<<"\n"
                <<"Return Value ="<<R_value<<"\n";
        }
int main()
{
fixed_deposit FD1, FD2, FD3;                    //deposits created
long int p;                                     //principal amount
int y;                                  //investment period, years
float r;                                //interest rate, decimal  form
int R;                                  // interest rate, percent  form
```

```
cout<<”Enter amount, period, interest rate (in percent)\n”;
cin>>p>>y>>R;
FD1 = fixed_deposit (p, y, R);
cout<< "Enter amount, period, interest rate(decimal form)\n";
cIn>> p>> y>> r;
FD2 = fixed_deposit (p,y,r);
cout<< "Enter amount and period\n”;
cin>>p >> y:
FD3 = fixed_deposit (p.y);

cout<< "\nDeposIt 1”;
FD1.display();

cout<< "\nDeposIt 2”;
FD2.display();

cout<< "\nDeposIt 3”;
FD3.display();

return 0;
}
```

The output of Program would be:

```
Enter amount, period, interest rate (in percent)

10000 3 18

Enter amount, period, interest rate (in decimal form)

10000, 3 0.18

Enter amount and period

10000 3
```

Deposit 1

Principal Amount = 10000
Return Value = 16430.3

Deposit 2

Principal Amount = 10000
Return Value = 16430.3

Deposit 3

Principal Amount = 10000
Return Value = 14049.3

The program uses three overloaded constructors. The parameter values to these constructors are provided at run time. The user can provide input in one of the following form:

1. Amount, period and interest in decimal form.
2. Amount, period and interest in percent form.
3. Amount and period,

Note that the constructors are overloaded with the appropriate parameters, the one that matches the input values is invoked. For example, the second constructor in invoked for the forms (1) and (3}, and the third is invoked for the form (2). Note that, for form (3), the constructor with default argument is used. Since input to the third parameter is missing, it uses the default value for **r**.

### 7.2 Copy Constructor

A copy constructor is used to declare and initialize an object from another object. For example, the statement

integer I2(I1);

would define the object I2 and at the same time initialize it to the values of **I1**.

Another form of this statement is

integer I2 = I1;

The process of initializing through a copy constructor is known as *copy initialization*. Note that the statement

I2 = I1;

Will not invoke the copy constructor. However, if **I1** and **I2** are objects, this statement is legal and simply assigns the values of **I1** to **I2**, member-by-member. This is the task of the overloaded assignment operator (=).

A copy constructor takes a reference to an object of the same class as itself as an argument. Consider a simple example of constructing and using a copy constructor

```
//Program Copy constructor
#include<iostream>
using namespace std;

class code
{
int id;
public:
code() { }                  //constructor
code(int a) {id = a;}       //constructor again
code(code &x)
{
id = x.id;                  //copy in the value
}
```

```
void display(void)
{ cout<<id; }
};

int main()
{
code A(100);                    // object A is created and initialized
code B(A);                      //copy constructor called
code C = A;                     //copy constructor called again
code D;                         //D is created, not initialized
D = A;                          //copy constructor not called
cout<< "\n id of A : "; A.display();
cout<< "\n id of B : "; B.display();
cout<< "\n id of C : "; C.display();
cout<< "\n id of D : "; D.display();
return 0;
}
```

The output of the program would be

id of A: 100

id of B: 100

id of C: 100

id of D: 100

A reference variable has been used as an argument to the copy constructor. We cannot pass the argument by value to a copy constructor.

When no copy constructor is defined, the compiler supplies its own copy constructor.

**7.3 Dynamic Constructors**

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory. Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator. The following program shows the use of new, in constructors that are used to construct strings in objects.

```
//program for Constructors with a new
#include<iostream>
#include<cstring>
using namespace std;

class String
{
char *name;
int length;
public:
String()                        //constructor - 1
{
length = 0;
name = new char [length + 1];
}
```

```
String (char* s)                    // Constructor - 2
{
length = strlen(s);
name = new char [length + 1];          // one additional character for \0
strcpy(name, s);
}
void display (void)
{ cout<<name<<"\n"; }
void join(String &a, String &b);
};

void String :: join (String &a, String &b)
{
length = a.length + b.length;
delete name;
name = new char[length + 1];          // dynamic allocation

strcpy(name, a.name);
strcat(name, b.name);
};

int main()
{
char * first = "Joseph";
String name1(first), name2("Louis"), name3("Lagrange"), s1, s2;
```

```
s1.join(name1, name2);
s2.join(s1, name3);
name1.display();
name2.display();
name3.display();
s1.display();
s2.display();
return 0;
}
```

The output of the program would be:

```
Joseph
Louis
Lagrange
Joseph Louis
Joseph Louis Lagrange
```

Note: This program uses two constructors. The first is an empty constructor that allows us to declare an array of strings. The second constructor initializes the **length** of the string, allocates necessary space for the string to be stored and creates the string itself. Note that one additional character space is allocated to hold the end-of-string character '\0'.

The member function **join()** concatenates two strings. It estimates the combined length of the strings to be joined, allocates memory for the combined string and then creates the same using the string function **strcpy()** and **strcat()**. Note that in the function **join()**, **length** and **name** are members

of the object that calls the function, while **a.length** and **a.name** are members of the argument object **a**. The **main()** function program concatenates three strings into one string. The output is as shown below:

Joseph Louis Lagrange

### 7.4 Constructing Two-dimensional Arrays

We can construct matrix variables using the class type objects. The following program illustrates how to construct a matrix of size m × n.

```
#include<iostream>

using namespace std;

class matrix
{
int **p;            //pointer to matrix
int d1, d2;

public:
matrix (int x, int y);
void get_element(int i, int j, int value)
{ p[i][j] = value; }

int &put_element(int i, int j)
{ return p[i][j]; }
};

matrix :: matrix (int x, int y)
{
d1 = x;
d2 = y;
p = new int *[d1];                  //creates an array pointer
for (int i=0; i < d1; i++)
p[i] = new int [d2];                // creates space for each row
}
```

```
int main()
{
int m,n;
 cout<<"Enter size of matrix: ";
cin>> m>>n;

matrix A(m,n);                          //matrix object A constructed
cout<<"Enter matrix elements row by row \n";
int i,j, value;
for (i=0; i< m; i++)
        for (j=0; j < n; j++)
                {
                cin>> value;
                A.get_element(i,j,value);
                }
cout<<"\n";
cout<<A.put_element(1,2);
return 0;
}
```

The output of the program would be:

Enter the size of matrix: 3 4

Enter the matrix elements row by row

11 12 13 14

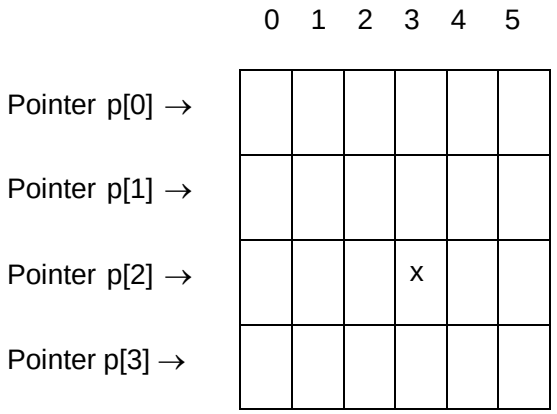
15 16 17 18

19 20 21 22

17

17 is the value of the element (1,2).

The constructor first creates a vector pointer to an **int** of size **d1**. Then, it allocates, iteratively an **int** type vector of size **d2** pointed at by each element **p[i]**. Thus, space for the element of a d1 × d2 matrix is allocated from free store as shown below:




x represents the element p[2][3].

### 7.5 Const Objects

We may create and use constant objects using const keyword before object declaration. For example, we may create X as a constant object of the class matrix as follows:

```
const matrix X(m,n);          //object X is constant
```

Any attempt to modify the values of m and n will generate compile-time error. Further, a constant object can call only const member functions. As we know, a const member is a function prototype or function definition where the keyword const appears after the function's signature.

Whenever const objects try to invoke nonconst member functions, the compiler generates errors.

### 7.6 Destructors

A destructor, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a

tilde. For example, the destructor for the class integer can be defined as shown below:

```
~integer() {}
```

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible. It is a good practice to declare destructors in a program since it releases memory space for future use.

Whenever new is used to allocate memory in the constructors, we should use delete to free that memory. For example, the destructor for the matrix class discussed above may be defined as follows:

```
matrix :: ~matrix()
{
for (int i=0; i < d1; i++)
delete p[i];
delete p;
}
```

This is required because when the pointers to objects go out of scope, a destructor is not called implicitly.

The example below illustrates that the destructor has been invoked by the compiler.

Program - Implementation of Destructors

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

int count = 0;
class test
{
```

```
public:
test()
{
count++;
cout<<"\n\n Constructor msg: object number"<<count<<"created ..";
}
~test()
{
cout<<"\nDestructor msg: Object number "<<count<<"destroyed..";
count--;
}
};

int main()
{
cout<<"Inside the main block..";
cout<<"\n\n Creating first object T1..";
test T1;
{
// block 1
cout<<"\n\n Inside Block 1..";
cout<<"\n\n creating two more objects T2 and T3..";
test T2, T3;
cout<<"\n\n Leaving block 1..";
}

cout<<"\n\n back inside the main block..";
return 0;
}
```

The output of the program would be:

Inside the main block..

Creating first object T1..

Constructor msg. object number 1 created..

Inside block 1..

creating two more objects T2 and T3..

Constructor msg: object number 2 created..

Constructor msg: object number 3 created..

Leaving block 1..

Destructor msg: object number 3 destroyed..

Destructor msg: object number 2 destroyed..

back inside the main block..

Destructor msg: object number 1 destroyed..

Note: A class constructor is called everytime an object is created. Similarly, as the program control leaves the current block the objects in the block start destroyed and destructors are called for each one of them. Note that the objects are destroyed in the reverse order of their creation. Finally, when the main block is exited, destructors are called corresponding to the remaining objects present inside main.

Similar functionality as depicted in program 6.7 can be attained by using static data members with constructors and destructors. We can declare a static integer variable count inside a class to keep a tract of the number of its objects instantiations. Being static, the variable will be initialized only once. i.e. when the first object instance is created. During all subsequent object creations, the constructors will increment the count variable by one. Similarly, the destructor will decrement the count variable by one as and when an object gets destroyed. To realize this scenario, the code in the program will change slightly, as shown below:

```
#include<iostream>
using namespace std;
class test
{
private:
static int count = 0;
public:
……
……
}
test()
{
……
count++;
}
~test()
{
……
count--;
}
```

The primary use of destructors is in freeing up the memory reserved by the object before it gets destroyed. The following program demonstrates how a destructor releases the memory allocated to an object.

//Program for Memory allocation to an object using Destructor

```
#include<iostream>

using namespace std;

class test
{

int *a;

public:
test (int size)
{
a = new int(size);
cout<<"\n\n Constructor Msg: Integer array of size "<< size<<" created..";
}


~test()
{
delete a;
cout<<"\nDestructor Msg: freed up the memory allocated for integer array";
}


};

int main()
{
int s;
cout<<"Enter the size of the array...";
cin>>n;
cout<<"\n\n Creating an object of the test class...";
test T(s);
return 0;
}
```

The output of the program would be:

Enter the size of the array ... 5

creating an object of test class..

Constructor Msg: Integer array of size 5 created...

Destructor Msg: Freed up the memory allocated for integer array

**SUMMARY**

- When an object is created and initialized at the same time, a copy constructor gets called.
- We may make an object **const** if it does not modify any of its data values.
- C++ provides a member function called the destructor that destroys the objects when they are no longer required.
- Destructor never takes arguments.
- When the closing brace of a scope is encountered, the destructor for each object in the scope are called, and the objects are destroyed in reverse order.

**Review Questions**

1. What do you mean by dynamic initialization of objects? Why do we need to do this?
2. How is dynamic initialization of objects achieved?
3. Describe the importance of destructors.

# UNIT – VIII

## OPERATOR OVERLOADING AND TYPE CONVERSIONS

### 8.1 Introduction

Operator overloading is one of the many exciting features of C++ language. It is an important technique that has enhanced the power of extensibility of C++. C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C++ permits us to add to variables of user-defined types with the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as *operator overloading*.

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can almost create a new language of our own by the creative use of the function and operator overloading techniques. We can overload (give additional meaning to) all the C++ operators except the following:

- Class member access operator (., .*)
- Scope resolution operator(::)
- Size operator (sizeof)
- Conditional operator (?:)

The reason why we cannot overload these operators may be attributed to the fact that these operators take names (example class name) as their operand instead of values, as is the case with other normal operators. Note that the excluded operators are very few when compared to the large number of operators which qualify for the operator overloading definition.

Although the *semantics* of an operator can be extended, we cannot change its *syntax*, the grammatical rules that *govern* its use such as the number of operands, precedence and associativity. For example, the

multiplication operator will enjoy higher precedence than the addition operator. Note that when an operator is overloaded, its original meaning is not lost. For instance, the operator *. Which has been overloaded to add two vectors, can still be used to add two integers.

**8.2 Defining Operator Overloading**

To define an additional task to an operator, we must specify what it means to relation to the class to which the operator is applied. This is done with the help of a special function, called *operator function*, which describes the task. The general form of an operator function is:

```
return type class name :: operator op(arglist)
{
Function body                    //task defined
}
```

where return type is the type of value returned by the specified operation and op is the operator being overloaded. Operator *op* is preceded by the keyword **operator. Operator** *op* is the function name.

Operator function must be either member functions (non-static) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators. This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with **friend** functions. Arguments may be passed either by value or by reference. Operator functions are declared in the class using prototype as follows:

```
vector operator + (vector);                 //vector addition
vector operator –();                        //unary minus
friend vector operator + (vector, vector);  //vector addition
friend vector operator – (vector);          //unary minus
```

```
vector operator – (vector &a);          //subtraction
int operator ==(vector);                //comparison
friend int operator == (vector, vector); // comparison
```

vector is a data type of class and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements (as in mathematics).

The process of overloading involves the following steps:

1. Create a class that defines the data type that is to be used in the overloading operation.
2. Declare the operator function **operator** op() in the public part of the class. It may be either a member function or a **friend** function.
3. Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

```
op x or x op
```

for unary operators and

```
x op y
```

for binary operators. op x (or x op) would be interpreted as

```
operator op (x)
```

for **friend** functions. Similarly, the expression x op y would be interpreted as either

```
x.operator op (y)
```

In case of member functions, or

```
operator op (x, y)
```

In case of **friend** functions. When both the forms are declared, standard argument matching is applied to resolve any ambiguity.

### 8.3 Overloaded Unary Operators

Let us consider the unary minus operator. A minus operator when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item. We will see here how to overload this operator so that it can applied to an object in much the same way as is applied to an int or float variable. The unary minus when applied to an object should change the sign of each of its data items.

//Program to show how the unary minus operator is overloaded.

```
#include<iostream>
#include<iomanip>

using namespace std;

class ABC
{
int a,b,c;
public:
void get_data()
{
cout<<"enter three integers"<<endl;
cin>>a>>b>>c;
}
void put_data()
{
cout<<"\na  ="<<setw(5)<<a;
cout<<"\nb  ="<<setw(5)<<b;
cout<<"\nc ="<<setw(5)<<c;
}

void operator ~();

};

void ABC :: operator ~()
{ a=-a;b=-b;c=-c; }
```

```
int main()
{
ABC A;
A.get_data();
cout<<"\n You have entered the following numbers";
A.put_data();
~A;
cout<<"\n The negative of the given numbers";
A.put_data();
return 0;
}
```

The output of the program would be:

```
enter three integers
10
-20
30
You have entered the following numbers
a =    10
b =   -20
c =    30

The negative of the given numbers
a =    -10
b =     20
c =    -30
```

Note that the function operator - () takes no argument. Then, what does this operator function do? It changes the sign of data members of the object A. Since this function is a member function of the same class, it can directly access the members of the object which activated it.

Note that a statement like

A2 = -A1;

Will not work because, the function **operator** - () does not return any value. It can work if the function is modified to return an object.

It is possible to overload a unary minus operator using a friend function as follows:

```
friend void operator – (ABC &s);          //declaration
void operator – (ABC &s)                  //definition
{
s.x = -s.x; s.y = -s.y; s.z = -s.z; }
```

Note that the argument is passed by reference. It will not work if we pass argument by value because only a copy of the object that activated the call is passed to operator-(). Therefore, the changes made inside the operator function will not reflect in the called object.

### 8.4 Overloading Binary Operators

We have just seen how to overload an unary operator. The same mechanism can be used to overload a binary operator. We have used friend function to add two complex numbers.

A statement like

```
C = sum(A, B);                //functional notation
```

was used. The functional notation can be replaced by a natural looking expression

```
C = A + B;                //arithmetic notation
```

By overloading the + operator using an operator +() function. The following program illustrates how this can be accomplished.

```
// Program to add two complex numbers by overloading + operator

#include<iostream>

using namespace std;

class complex
{
float re, im;
public:
complex() { }                    //constructor 1
complex(float rep, float imp)
{ re = rep; im = imp;}           //constructor 2
void getdata()
{
cout<<"\nEnter the real part";
cin>>re;
cout<<"\nEnter the imaginary part";
cin>>im;
}
void display()
{
cout<< re<< " + (" << im <<") i";
}
complex operator + (complex);
};
```

```
complex complex:: operator + (complex T)
{
complex t;
t.re = re + T.re;
t.im = im + T.im;
return (t);
}
void main()
{
complex a, b, c;
a.getdata();
b = complex(5, 7);
c = a + b;
cout<< "\nThe sum of complex numbers\n";
a.display();
cout<<" and ";
b.display();
cout<<" is ";
c.display();
return 0;
}
```

The output of the program would be:

Enter the real part:

3

Enter the imaginary part

9

The sum of complex numbers

3 + (9)i

and

5 + (7)i

Is

8 + (16)i

In the above program consider the block:

```
complex complex :: operator + (complex T)
{
complex t;
t.x = x + T.x;
t.y = y + T.y;
return (t);
}
```

We should note the following features of this function:

1. It receives only one **complex** type argument explicitly.
2. It returns a **complex** type value.
3. It is a member function of **complex**.

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument. Where does the other value come from? Now let us look at the statement that invokes this function:

```
c = a + b;                    //invokes operator +() function
```

We know that a member function can be invoked only by an object of the same class. Here, the object **a** takes the responsibility of invoking the function and **b** plays the role of the argument that is passed to the function. The above invocation statement is equivalent to

```
c = a.operator+(b);           //usual function call syntax
```

Therefore, in the operator +() function, the data members of **a** are accessed directly and the data members of **b** (that is passed as argument) are accessed using the dot operator. Thus, both the objects are available for the function. For example, in the statement

```
t.x = x + T.x;
```

T.x refers to the object **b** and x refers to the object **a**. t.x is the real part of **t** that has been created specially to hold the result of addition of **a** and **b**. The function returns the complex **t** to be assigned to c.

As a rule, in overloading of binary operators, the left-hand operand is used to invoke the operator function and the right-hand operand is passed as an argument.

We can avoid the creation of the temp object **t** by replacing the entire function body by the following statement:

```
return complex((x + T.x), (y+T.y));          //invokes constructor 2
```

What does it mean when we use a class name with an argument list? When the compiler comes across a statement like this, it invokes an appropriate constructor, initializes an object with no name and returns the contents for copying into an object. Such an object is called a temporary object and goes out of space as soon as the contents are assigned to another

object. Using temporary objects can make the code shorter, more efficient and better to read.

### 8.5 Overloading Binary Operators using Friends

Friend functions may be used in the place of member functions for overloading a binary operator, the only difference being that a friend function requires two arguments to be explicitly passed to it, while a member function requires only one.

The complex number program discussed in the previous section can be modified using a friend operator function as follows:

1. Replace the member function declaration by the friend function declaration.

   friend complex operator + (complex, complex);

2. Redefine the operator function as follows:

```
complex operator + (complex a, complex b)
{
return complex((a.x+b.x),(a.y+b.y));
}
```

   In this case, the statement

```
c = a + b
```

   is equivalent to

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

In most cases, we will get the same results by the use of either a friend function or a member function. Why then an alternative is made available? There are certain situations where we would like to use a **friend** function rather than a member function. For instance, consider a situation where we

need to use two different types of operands for a binary operator, say one an object and another a built-in type data as shown below:

A = B + 2; (or A = B * 2;)

where **A** and **B** are objects of the same class, this will work for a member function but the statement

A = 2 + B; (or A = 2 * B;)

Will not work. This is because the left-hand operand which is responsible for invoking the member function should be an object of the same class. However, **friend** function allows both approaches.

It may be recalled that an object need to be used to invoke a **friend** function but can be passed as an argument. Thus, we can use a friend function with a built-in type data as the left-hand operand and an object as the right-hand operand. The following program illustrates this, using scalers multiplication of a vector. It also shows how to overload the input and output operators >> and <<.

```
//overloading operators using Friends
#include<iostream>
using namespace std;

const size = 3;
class vector
{
int v[size];
public:
vector();                    //constructs null vector
vector(int *x);              //constructs vector from array
friend vector operator *(int a, vector b);          //friend 1
friend vector operator *( vector b, int a);         //friend 2
friend istream & operator >> (istream&, vector &);
friend ostream & operator << (ostream&, vector &);
};
```

```
vector :: vector()
{
for (int i = 0; i < size; i++)
v[i] = 0;
}

vector :: vector(int *x)
{ for (int i = 0; i < size; i++) v[i] = x[i]; }

vector operator *(int a, vector b)
{
vector c;
for (int i = 0; i < size; i++)
        c.v[i] = a * b.v[i];
return c;
}

vector operator * (vector b, int a)
{
vector c;
for (int i = 0; i < size; i++) c.v[i] = b.v[i]*a;
return c;
}

istream & operator >> (istream &din, vector &b)
{
for (int i = 0; i < size; i++)
        din>> b.v[i];
return (din);
}
```

```
ostream & operator << (ostream &dout, vector &b)
{
dout<<"("<<b.v[0];
for (int i = 1; i < size; i++)
        dout << ", "<< b.v[i];
dout<<")";
return (dout);
}

int x[size] = {2, 4, 6};

int main()
{
vector m;
vector n = x;
cout<< "Enter elements of vector m"<< "\n";
cin >> m;                              //invokes operator >>() function
cout<<"\n";
cout<<"m = " << m<< "\n";              //invokes operator <<()
vector p, q;
p = 2* m;                              //invokes friend 1
q = n * 2;                             //invokes friend 2

cout<<"\n";
cout<<"p = " << p << "\n";             //invokes operator <<()
cout<<"q = " << q << "\n";
return 0;
}
```

The output of the program would be:

Enter elements of vector m

5 10 15

m = (5, 10, 15)

p = (10, 20, 30)

q = (4, 8, 12)

The program overloads the operator * two times, thus overloading the operator function operator*() itself. In both the cases, the functions are explicitly passed two arguments and they are invoked like any other overloaded function, based on the types of its arguments. This enables us to use both the forms of scaler multiplication such as

```
p = 2 * m;          // equivalent to p = operator * (2, m);
q = n * 2;          // equivalent to q = operator * (n, 2);
```

The program and its output are largely self-explanatory. The first constructor

vector();

constructs a vector whose elements are all zero. Thus,

vector m;

creates a vector m and initializes all its elements to 0. The second constructor

vector (int &x);

creates a vector and copies the elements pointed to by the pointer argument x into it. Therefore, the statement

int x[3] = {2, 4, 6};

vector n = x;

create n as a vector with components 2, 4 and 6.

## 8.6 Manipulation of Strings Using Operators

ANSI C implements strings using character arrays, pointers and string functions. There are no operators for manipulating the strings. One of the main drawbacks of string manipulations in C is that whenever a string is to be copied, the programmer must first determine its length and allocate the required amount of memory.

Although these limitation exist in C++ as well, it permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to the decimal numbers. (Recently, ANSI C++ committee has added a new class called **string** to the C++ class library that supports all kinds of string manipulations. For example, we shall be able to use statements like

```
string3 = string1 + string2;

if (string1 > string2) string = strtng1;
```

Strings can be defined as class objects which can be then manipulated like the built-in types. Since the strings vary greatly in size, we use *new* to allocate memory for each string and a pointer variable to point to the string array. Thus, we must create string objects that can hold these two pieces of information, namely, length and location which are necessary for string manipulations. A typical string class will look as follows:

```
class string
{
char *p;                // pointer to string
int len;                // length of string
public:
…….                     // member functions
…….                     //to initialize and
…….                     // manipulate strings
};
```

We shall consider an example to illustrate the application of overloaded operators to strings. The example shown in the following program overloads two operators, + and <= just to show how they are implemented. This can be extended to cover other operators as well.

```
//Mathematical operations on Strings

#include<iostream>
#include<cstring>

using namespace std;

class string

{

char *p;

int len;

public:

string() {len = 0; p 0;}          // create null string

string (const char * s);          // create string from arrays

string (const string & s);        // copy constructor

~ string(3 {delete p;}            // destructor

// + operator

friend strlng operator + ( const string &s, const string &t);

// << operator

friend int operator <= (const string &s, const string &t);

friend void show (const string s);

};
```

```
string ::string (const char *s)
{
len = strlen(s);
p = new char [len+1];
strcpy(p, s);
}
string:: string (const string & s)
{
len - s.len;
p = new char [ len+1];
strcpy(p, s.p);
}
// overloading + operator
string operator + (const string &s, const string &t)
{
string temp;
temp.len = s.len + t.len;
temp.p = new char [temp. len+1] ;
strcpy(temp.p, s.p);
strcat(temp.p, t.p);
return (temp);
}
```

```
// overloading <= operator
int operator <= (const string &s, const string &t)
        {
        int m = strlen(s.p);
        int n = strlen (t.p):
        (m <= n) ? return (1) : return (0);
        }
void show(const string s)
        {
        cout << s.p;
        }
int main()
        {
        string s1 = “New “;
        string s2 = "York”;
        string s3 = "Delhi';
        string t1, t2, t3;

        t1 = s1;
        t2 = s2;
        t3 = s1 + s2;

        cout << "\n t1 = "; show(t1);
        cout << "\n t2 = "; show(t2);
        cout << "\n t3 = "; show(t3);
```

```
cout<<"\n\n";
if (t1 <= t3)
        {
        show(t1);
        cout<< "Smaller than ";
        show(t3);
        cout<<"\n";
        }
else
        {
        show(t3);
        cout<< "Smaller than ";
        show(t1);
        cout<<"\n";
        }
return 0;
}
```

The following is the output of the program:

```
t1 = New
t2 = York
New Smaller than New Delhi
```

### 8.7 Rules for overloading Operators

Although it looks simple to redefine the operators, there are certain restrictions and limitations in overloading them. Some of them are listed below:

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type.
3. Overloaded operator follow the syntax rules of the original operators. They cannot be overridden.
4. There are some operators that cannot be overloaded.
5. We cannot use friend functions to overload certain operators. However, member functions can be used to overload them.
6. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
7. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
8. When using binary operators overloaded through a member function, the left-hand operand must be an object of the relevant class.
9. Binary arithmetic operators such as +, -, * and / must explicitly return a value. They must not attempt to change their own arguments.

Operators that cannot be overloaded

| Sizeof | Size of operator |
|---|---|
| . | Membership operator |
| .* | Pointer-to-member operator |
| :: | Scope resolution operator |
| ?: | Conditional operator |

Where friend cannot be used

| Sizeof | Size of operator |
|---|---|
| = | Assignment operator |
| () | Function call operator |
| [] | Subscripting operator |
| -> | Class member access operator |

### 8.8 Type Conversions

We know that when constants and variables of different types are mixed in an expression, C applies automatic conversion to the operands as per certain rules. Similarly, an assignment operators also causes the automatic type conversion. The type of data to the right of an assignment operator is automatically converted to the type of the variable on the left. For example, the statements

```
int m;
float x = 3.14159;
m = x;
```

convert x to an integer before its value is assigned to **m**. Thus, the fractional part is truncated. The type conversions are automatic as long as the data types involved are built-in types.

Consider the following statement that adds two objects and then assigns the result to the third object.

```
v3 = v1 + v2;          // v1 , v2 and v3 are class type objects
```

when the objects are of the same type, the operations of addition and assignment are carried out smoothly and the compiler does not make any complaints. We have seen, in the case of class objects, that the values of all the data members of the right-hand object are simply copied into the corresponding members of the object on the left-hand. What if one of the operators is an object and the other is a built-in type variable? Or, what if they belong to two different classes?

Since the user-defined data types are designed by us to suit our requirements, the compiler does not support automatic type conversions for such data types. We must, therefore, design the conversion routines by ourselves. If such operations are required,

Three types of situations might arise in the data conversion between incompatible types:

1. Conversion from basic type to class type
2. Conversion from class type to basic type
3. Conversion from one class type to another class type.

We shall discuss all the three cases in detail:

**Basic to Class Type**

The conversion from basic type to class type is easy to accomplish. It may be recalled that the use of constructor was illustrated in a number of examples to initialize objects. For example, a constructor was used to build a vector object from an **int** type array. Similarly, we used another constructor to build a string type object from a **char*** type variable. These are all exampleswhere

constructors perform a *defacto* type conversion from the argument's type to the constructor's class type.

Consider the following constructor

```
string :: string (char *a)
{
length = strlen(a);
p = new char[length + 1];
strcpy(p, a);
}
```

This constructor builds a **string** type object from **char*** type variable **a.** The variables **length** and **p** are data members of the class string. Once this constructor has been defined in the string class, it can be used for conversion from **char*** type to string type.

Example

```
string s1, s1;
char* name1 = "IBM PC";
char* name2 = "APPLE COMPUTERS";
s1 = string(name);
s2 = name2;
```

The statement

```
s1 = string(name);
```

first converts **name1** from **char*** type to **string** type and then assigns the string type values to the object **s1**. The statement

```
s2 = name2;
```

also does the same type by invoking the constructor implicitly.

Let us consider another example of converting an **int** type to a **class** type.

```
class time
{
int hrs;
int mins;
public:
.....
.....
time(int t)
{
hrs = t/60;                 // t in minutes
mins = t % 60;
}
};
```

The following conversion statements can be used in a function:

```
time T1;                    //object T1 created
int duration = 85;
T1 = duration;              //int to class type
```

After this conversion, the hrs member of T1 will contain the value of 1 and minus member a value of 25, denoting 1 hours and 25 minutes.

In both the examples, the left-hand operand of = operator is always a class object. Therefore, we can also accomplish this conversion using an overloaded = operator.

**Class to Basic Type**

The constructors did a fine job in type conversion from a basic to class type. What about the convention from a class to basic type? The constructor functions do not support this operation. Luckily, C++ allows us to define an overloaded casting operator that could be used to convert a class type data

to a basic type. The general form of an overloaded casting operator function, usually referred to as a conversion function is:

```
Operator typename()
{
……
……..          (function statements)
……..
}
```

This function converts a class type data to typename. For example, the **operator double** () converts a class object to type **double**, the **operator int()** converts a class type object to type int, and so on.

Consider the following conversion function:

```
vector :: operator double()
{
double sum 3 0;
for ( int i = 0: i < size; i++)
        sum = sum + v[i] *v[i];
return sqrt(sum);
}
```

This function converts a vector to the corresponding scalar magnitude. Recall that the magnitude of a vector is given by the square root of the sum of the squares of its components. The operator **double**() can be used as follows:

```
double length = double (V1);
        or
double length = V1;
```

where V1 is an object of type **vector**. Both the statements have exactly the same effect. When the compiler encounters a statement that requires the conversion of a class type to a basic type, it quietly calls the casting operator function to do the job.

The *casting operator function* should satisfy the following condition:

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

Since it is a member function, it is invoked by the object and, therefore, the values used for conversion inside the function belong to the object that invoked the function. This means that the function does not need an argument.

In the string example described in the previous section, we can do the conversion from **string** to **char*** as follows:

```
string :: operator char*()
{
return (p) ;
}
```

**One Class to Another Class Type**

We have just seen data conversion techniques from a basic to class type and a class to basic type. But there are situations where we would like to convert one class type data to another class type. Example:

```
objX = objY; // objects of different types
```

**objX** is an object of class **X** and **objY** is, an object of class **Y**. The **class Y** type data is converted to the **class X** type data and the converted value is assigned to the **objX**. Since the conversion takes place from **class Y** to **class X**, **Y** is known as the *source class* and **X** is known as the *destination class*.

Such conversions between objects of different classes can be carried out by either a constructor or a conversion function. The compiler treats them the same way. Then, how do we decide which form to use? It depends upon where we want the type-conversion function to be Located in the source class or in the destination class.

We know that the casting operator function

operator typename()

converts the class object of which it is a member to typename. The typename may be a built-in type or a user-defined one (another class type). In the case of conversions between objects, *typename* refers to the destination class, Therefore, when a class needs to be converted, a casting operator function can be used (i.e. source class). The conversion takes place in the source class and the result is given to the destination class object.

Now consider a single-argument constructor function which serves as an instruction for converting the *argument's type*, to the class type of *which it is a member*. This implies that the argument belongs to the *source class* and is passed to the *destination class* for conversion. This makes it necessary that the conversion constructor be placed in the destination class.

Table below provides a summary of all the three conversions, It shows that the conversion from a class to any other type (or any other class) should make use of a casting operator in the source class. On the other hand, to perform the conversion from any other typo/class to a class type, a constructor should be used in the destination class.

Table :Type conversions

| Conversion required | Conversion takes place in | |
|---|---|---|
| | Source Class | Destination Class |
| Basic $\rightarrow$ class | Not applicable | Constructor |
| Class $\rightarrow$ basic | Casting operator | Not applicable |
| Class $\rightarrow$ class | Casting operator | constructor |

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument. Since data members of the source class are private, we must use special *access* functions in the source class to facilitate its data flow to the destination class.

## SUMMARY

- Operator overloading is one of the important features of C++ Language, It is called compile time polymorphism.
- Using overloading feature we can add two user defined data types such as objects, with the same syntax, just as basic data types.
- We can overload almost all the C++ operators except the following:
    - class member access operators(., .*)
    - scope resolution operator (::)
    - size operator(sizeof)
    - conditional operator (?:)
- Operator overloading is done with the help of a special function, called operator function, which describes the special task to an operator.
- There are certain restrictions and limitations in overloading operators. Operator functions must either be member functions (non-static) or friend functions. The overloading operator must have at least one operand that is of user-defined type.
- The compiler does not support automatic type conversions for the user defined data types. We can use casting operator functions to achieve this.
- The casting operator function should satisfy the following conditions:
    - It must be a class member.
    - It must not specify a return type.
    - It must not have any arguments

**Review Questions**

1. What is operator overloading?
2. Why is it necessary to overload an operator?
3. What is an operator function? Describe the syntax of an operator function.
4. How many arguments are required in the definition of an overloaded unary operator?
5. A class alpha has a constructor as follows:

   alpha (int a, double b);

   Can we use this constructor to convert types?
6. What is a conversion function? How is it created? Explain its syntax.
7. A friend function cannot be used to overload the assignment operator =. Explain why?
8. When is a friend function compulsory? Give an example.
9. We have two classes **X** and **Y**. If **a** is an object of X and **b** in an object of Y and we want to say **a = b**; What type of conversion routine should be used and where?
10. State whether the following statements are TRUE or FALSE.
    a) Using the operator overloading concept, we can change the meaning of an operator.
    b) Operator overloading works when applied to class objects only,
    c) Friend functions cannot be used to overload operators.
    d) When using an overloaded binary operator, the left operand is implicitly passed to the member function,
    e) The overloaded operator must have atleast one operand that is user-defined type.
    f) Operator functions never return a value.
    g) Through operator overloading, a class type data can be converted to a basic type data.

h) A constructor can be used to convert a basic type to a class type data.

**Programming Exercise**

1. Create class MAT of size m x n. Define all possible matrix operations for MAT type objects.
2. Define a class String. Use overloaded == operator to compare two strings.
3. Define two classes Polar and Rectangle to represent points in the polar and rectangle systems. Use conversion routines to convert from one system to the other.

# BLOCK – V

Objectives:

After Completion of this block, students will be able to

1. Extend the existingclasses
2. Apply pointer concept to utilize memory efficiently
3. Write program to solve real-life problems.

# UNIT – IX

## INHERITANCE: EXTENDING CLASSES

### 9.1 Introduction

Reusability is yet another important feature of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again. It would not only save time and money but also reduce frustration and increase reliability. For, instance, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

Fortunately, C++ strongly supports the concept of *reusability*. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called *inheritance* (or derivation). The old class is referred to as the *base class* and the new one is called the *derived class* or *subclass*.

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class, is called single inheritance and one with several base classes is called multiple inheritance, On the other hand, the traits of one class may be inherited by more than one class. This process is known as hierarchical inheritance. The mechanism of deriving a class from another “derived class” is known as multilevel inheritance. Figure 9.1 shows various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance, (Some authors show the arrow in Opposite direction meaning "inherited from”).

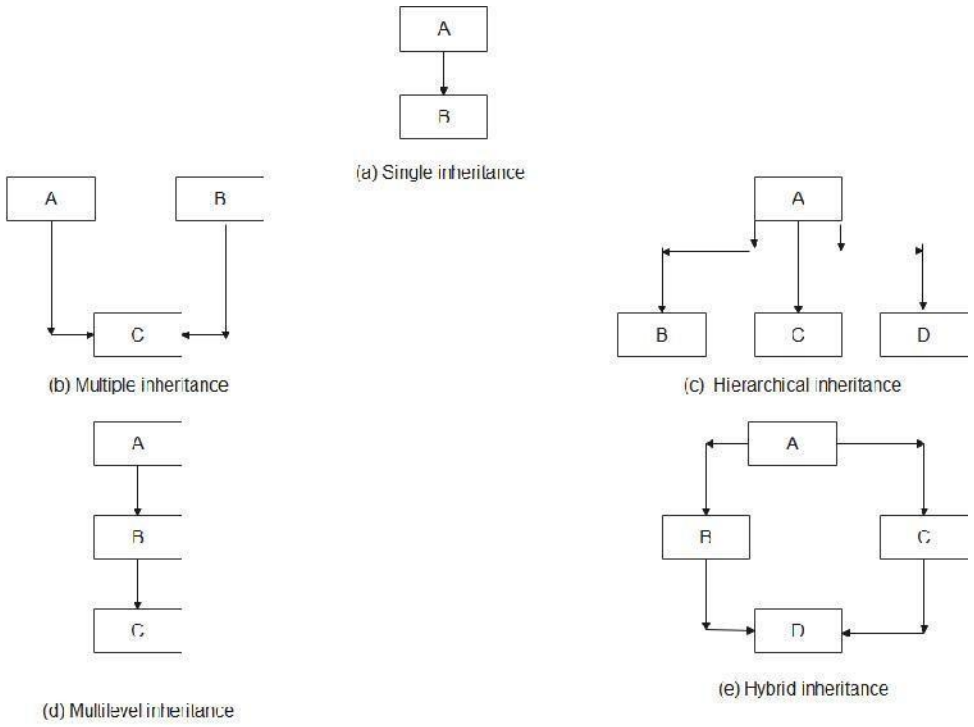



Fig. 9.1 Forms of Inheritance

### 9.2 Defining Derived Classes

A derived class can be defined by specifying its relationship with the base class in addition to Its own details, The general form of defining a derived class is:

**class** derived-class-name : visibility-mode base-class-name

{

…… //

…… // members of derived class

……

};

The colon indicates that the *derived-class-name* is derived from the *base•class-name*. The *visibility-mode* is optional and, if present, may be either **private** or **public**. The default visibility-mode is **private**. Visibility mode

.specifies whether the features of the base class are *privately derived* or *publicly derived*.

Examples:

```
class ABC : private XYZ        // private derivation
{
Members of ABC
};
class ABC ; public XYZ         //public derivation
{
Members of ABC
};
class ABC : XYZ                //private derivation by default
{
Members of ABC
};
```

When a base class in privately *inherited* by a derived class, 'public members' of the base class become ‘private members’ of the derived class and therefore the public members of the base class can, only be accessed by the member functions of the derived class, They are inaccessible to the objects of the derived class. Note that a public member of a class can be accessed by its own objects using the *dot operator*. The result is that no member of the base class is accessible to the objects of the derived class.

On the other hand, when the base class is *publicly inherited*, 'public members’ of the base class become 'public members’ of the derived class and therefore they are accessible to the objects of the derived class. In both

the cases, the *private members* are *not inherited* and therefore, the private members of a base class will never become the members of its derived class.

In inheritance, some of the base class data elements and member functions are 'inherited' into the derived class. We can **add** our own data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

### 9.3 Single Inheritance

Let us consider a simple example to illustrate inheritance, Program 9.1 shows a base class B and a derived class D. The class B contains one private data member, one public data member, and three public member functions. The class D contains one private data member and two public member functions.

```
// single inheritance
#include<iostream>

using namespace std;
class B
{
int a;                  //private, not inheritable
public:
int b;                  //public, ready for inheritance
int get_a();
void set_ab();
void put_a();
};
class D:public B                //public derivation
{
int c;
public:
void mul();
void display();
};
```

```
void B :: set_ab()
        {
        a=5;b=10;
        }
int B :: get_a()
        {
        return(a);
        }
void B :: put_a()
        {
        cout<<"\n a= "<<get_a();
        }
void D :: display()
        {
        cout<<"\n a="<<get_a();
        cout<<"\n b="<<b;
        cout<<"\n c="<<c;
        }
void D :: mul()
        {
        c=b*get_a();
        }
int main()
        {
        D d;
        d.set_ab();
        d.mul();
        d.display();
        d.put_a();
```

```
cout<<"\n Enter new value for b";
cin>>d.b;
cout<<"\n After changed value for b";
d.mul();
d.display();
return 0;
}
```

The output of the program would be:

```
a = 5
b = 10
c = 50
Enter new value for b
25
After changed value for b
a =  5
b =  25
c =  125
```

The class **D** is a public derivation of the base class **B**. Therefore, **D** inherits all the public members of **B** and retains their visibility. Thus, a **public** member of the base class **B** is also a public member of the derived class **D**, The **private** members of **B** cannot be *inherited* by **D.** The class **D**, in effect, will have more members than what it contains at the time of declaration.

The program illustrates that the objects of class **D** have access to all the public members of **B**.

Let us have a look at the functions **put_a()** and **mul()**;

```
void put_a()
{
cout<<"\n a= "<<get_a();
}
void mul()
{
c=b*get_a();                    // c = b * a
}
```

Although the data member **a** is private in **B** and cannot be inherited, objects of **D** are able to access it through an inherited member function of **B**.

Let us now consider the case of private derivation.

```
class B
{
int a;
public:
int b;                  //public, ready for inheritance
void set_ab();
void get_a();
void put_a();
};
class D:private B               //private derivation
{
int c;
public:
void mul();
void display();
};
```

In private derivation, the public members of the base class become private members of the derived class. Therefore, the objects of **D** can not have direct access to the public member functions of **B**.

The statement such as

```
d. set_ab();            // set_ab() is private
d.get_a();              // so also get_a()
d.put_a();              // and put_a()
```

will not work. However, these functions can be used inside **mul()** and **display()** like the normal functions as shown below:

```
void mul()
{
set_ab();
c=b*get_a();
}
void display()
{
cout<<"\n a="<<get_a();        //output value of 'a'
cout<<"\n b="<<b;
cout<<"\n c="<<c;
}
```

The following program 9.2 incorporates these modifications for private derivation.

```
//Single level inheritance - privately inherited
#include <iostream>
using namespace std;
```

```
class B
{
int a;
public:
int b;
void   get_ab();
int get_a(void);
void put_a(void);
};
class D : private B
{
int c;
public:
void mul(void);
void display(void);
};

void B:: get_ab(void)
{
cout << "Enter values for a and b: ";
cin >> a >> b;
}
int B :: get_a()
{
return (a);
}
void B :: put_a()
{
cout << "a = "<< a << "\n";
}
```

```
void D :: mul()
{
get_ab();
c = b * get_a();
}
void D:: display()
{
put_a();
cout << "b = " << b << "\n"
        << "c = " << c << "\n\n";
}

int main()
{
D d;
d.mul();
d.display();
return 0;
}
```

The output of the program would be:

```
Enter values of a and b
7       15
a = 7
b = 15
c = 105
```

Note that the **d.b = 20;** if inserted in main() will not work, since **b** has become private.

Suppose a base class and a derived class define a function of the same name. What will happen when a derived class object invokes the function?. In such cases, the derived class function supersedes the base class definition. The base class function, will be called only if the derived class does not redefine the function.

### 9.4 Making a private Member inheritable

We have just seen how to increase the capabilities of an existing class without modifying it. We have also seen that a private member of a base class cannot be inherited and therefore it is not available for the derived class directly. What do we do if the private data needs to be inherited by a derived class? This can be accomplished by modifying the visibility limit of the **private** member by making it **public**. This would make it accessible to all the other functions of the program, thus taking away the advantage of data hiding.

C++ provides a third *visibility modifier*, **protected**, which serve a limited purpose in inheritance. A member declared as **protected** is accessible by the member functions within its class and any class immediately derived from it. It cannot be accessed by the functions outside these two classes. A class can now use all the three visibility modes as illustrated below:

```
class alpha
{
private:                    //optional
…….                         // visible to member functions
…….                         //within its class
public:
……..                       //visible to all functions
…….                         //in the program
};
```

When a **protected** member is inherited in **public** mode, it becomes **protected** in the derived class too and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A **protected** member, inherited in the **private** mode derivation, becomes **private** in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance (since **private** members cannot be inherited).

The keywords **private, protected**, anid **public** may appear in any order and any number of times in the declaration of a class. For example,

```
class beta
{
protected;
……
 pub!ic:
…….
private:
…….
public:
……..
};
```

Is a valid class definition.

However, the normal practice is to use them as follows:

```
class beta
{
……..                    //private by default
……..
protected:
……
public:
……
};
```

It is also possible to inherit a base class in **protected** mode (known as *protected derivation*). In protected derivation, both the **public** and **protected** members of the base class become **protected** members of the derived class. Table 9.1 summarizes how the visibility of base class members undergoes modifications in all the three types of derivation.

Now let us review the access control to the **private** and **protected** members of a class. What are the various functions that can have access to these members'? They could be:

1. A function that is a friend of the class.
2. A member function of a class that is a friend of the class.
3. A member function of a derived class.

While the friend functions and the member functions of a friend class can have direct access to both the **private** and **protected** data, the member functions of a derived class can directly access only the **protected** data. However, they can access the **private** data through the member functions of the base class. A simplified view of access control to the members of the class is shown in 9.2.

Fig.9.2 A simple view of access control to the members of a class

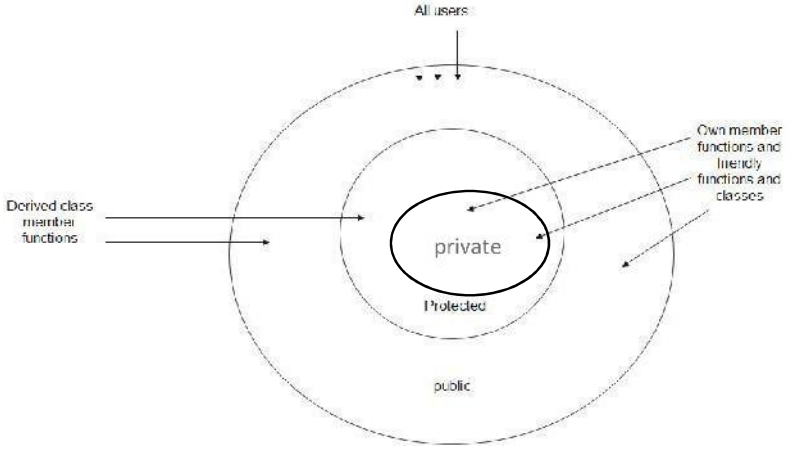


Table 9.1 Visibility of inherited members

| Base class visibility | Derived class visibility | | |
|---|---|---|---|
| | Public derivation | Private derivation | Protected derivation |
| Private | Not inherited | Not inherited | Not inherited |
| Protected | Protected | Private | Protected |
| Public | Public | Private | Protected |

**9.5 Multilevel Inheritance**

It is not uncommon that a class is derived from another derived class as shown in Fig. 9.3. The class **A** serves as a base class for the derived class **B**, which in turn serves as a base class for the derived class **C**, The class **B** is known as *intermediate base class* since it provides a link for the inheritance between **A** and **C**. The chain **ABC** is known as *inheritance path*.

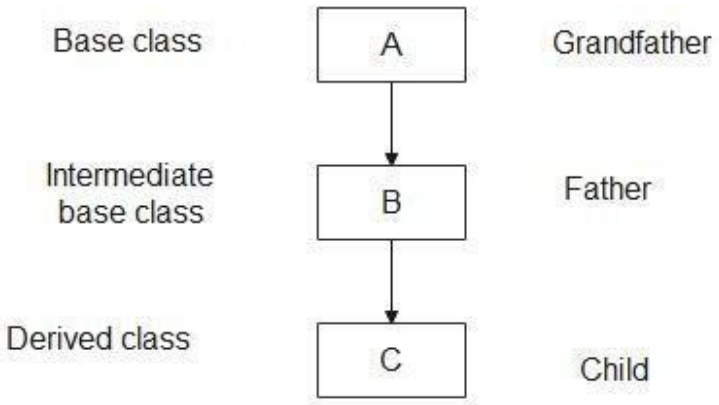



Fig. 9.3 Multilevel inheritance

A derived class with multilevel inheritance is declared as follows:

```
class A {...};                  // Base class
class B: public A {….};         // B derived from A
class C: public B {..,..};      // C derived from B
```

This process can be extended to any number of levels, Let us consider a simple example. Assume that the test results of a batch of students

are stored in three different classes. Class **student** stores the roll-number, class **test** stores the marks obtained in two subjects and class **result** contains the **total** marks obtained in the test. The class **result** can inherit the details of the marks obtained in the test and the roll-number of students through multilevel inheritance, Example:

```
/* multi level inheritance */

#include<iostreamh>
#include<iomanip>

using namespace std;

class student
{
protected:
int roll_number;

public:
void get_number();
void put_number();

};

void student :: get_number()
{
cout<<"\n Enter Roll number";
cin>>roll_number;
}

void student :: put_number()
{
cout<<"\n Roll number="<<setw(15)<<roll_number;
}
class test:public student
{
protected:
float sub1,sub2;
```

```
public:
void get_marks();
void put_marks();
};

void test :: get_marks()
{
cout<<"\n Enter marks in two subjects";
cin>>sub1>>sub2;
}

void test :: put_marks()
{
cout<<"\n Marks in sub1="<<setw(15)<<sub1;
cout<<"\n Marks in  sub2="<<setw(15)<<sub2;
}

class result:public test
{
float total;
public:
void display();
};

void result :: display()
{
total=sub1+sub2;
put_number();
put_marks();
cout<<"\n Total="<<setw(15)<<total;
}
```

```
int main()
{
int i,n;
result student[10];
cout<<"\n How many students?";
cin>>n;
for(i=0;i<n;i++)
{
student[i].get_number();
student[i].get_marks();
}
cout<<"\n Result";
for (i=0;i<n;i++)
student[i].display();
return 0;
}
```

The class result, after inheritance from 'grandfather' through 'father', would contain the following members:

```
private:
float total;                    //own member
protected:
int roll_number;                //inherited from student via test
float subl;                     //inherited from test
float sub2;                     //inherited from test
public:
void get_number(int):           //from student via test
void put_number (void);         //from student via test
void getmarks (float, float); //from test
```

```
    void put_marks (void);          //from  test
    void display(void);             //own member
```

The inherited functions **put_number()** and **put_marks()** can be used in the definition of **display()** function.

The output of the program would be:

```
    How many students?
    2
    Enter roll number
    111
    Enter marks in two subjects
    75 90
    Enter roll number
    222
    Enter marks in two subjects
    99 95
    Result
    Roll  number  =  111
    Marks  in  sub1  =  75
    Marks  in  sub2  =  90
    Total =             165

    Roll number =    222
    Marks in sub1 =   99
    Marks in sub2 =   95
    Total =             194
```

### 9.6 Multiple Inheritance

A class can inherit the attributes of two or more classes as shown in Fig. 9.4. This is known as *multiple inheritance*. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and the intelligence of another.

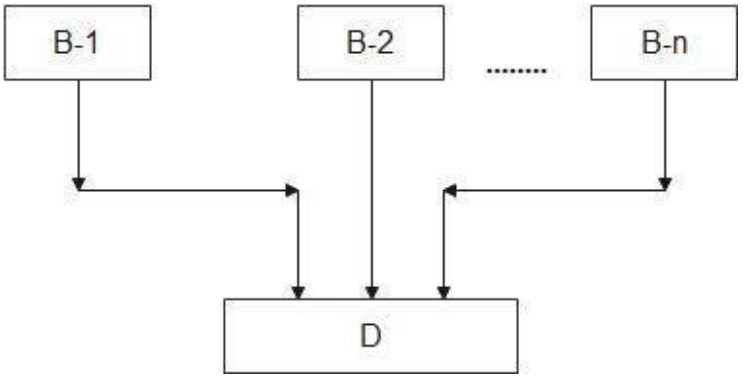



Fig. 9.4 Multiple Inheritance

The syntax of a derived class with multiple base classes is as follows:

```
class D : visibility B-1, visibility B-2, ….
{
……..
…….        (Body of D)
……
};
```

where, visibility may be either **public** or **private**. The base classes are separated by commas. Example:

```
class P : public M, public  N
{
public:
void display(void);
};
```

Classes M and N have been specified as follows:

```
class M
{
protected:
int m;
public:
void get_m(int);
};
class N
{
protected:
int n;
public:
void get_n(int);
};
void N::get_n(int y)
{
n = y;
}
```

The derived class **P**, as declared above, would, in effect, contain all the members of **M** and **N** in addition to its own members as shown below:

```
class P
{
protected:
int m;              //from M
int n;              //from N
publc:
void get_m(int);          //from M
void get_n(int);          //from N
void display(void);       //own member
};
```

The member function display() can be defined as follows:

```
void P:: display(void)
{
cout<<"m = "<<m<<"\n";
cout<<"n = "<<n<<"\n";
cout<<"m*n = "<<m*n<<"\n";
}
```

The main function which provides the user-interface may be written as follows:

```
main()
{
P p;
p.get_m(10);
p.get_n(20);
p.display();
}
```

The following program illustrates how all three classes are implemented in multiple inheritance.

```
/* multiple inheritance */

#include<iostream>
#include<iomanip>

using namespace std;

class M
{
protected:
int m;
public:
void get_m(void);
};
```

```
class N
{
protected:
int n;
public:
void get_n(void);
};
class D:public M, public N
{
public:
void display(void);
};
void M :: get_m()
{
cout<<"\n Enter the value of m";
cin>>m;
}
void N :: get_n()
{
cout<<"\n Enter the value of n";
cin>>n;
}
void D :: display()
{
cout<<"\n m="<<setw(5)<<m;
cout<<"\n n="<<setw(5)<<n;
cout<<"\n m * n =  "<<setw(5)<<m*n;
}
int main()
{
D d;
d.get_m();
d.get_n();
```

```
d.display();
return 0;
}
```

The output of the program would be

```
Enter the value of m
10
Enter the value of n
20
m =      10
n =      20
m*n=  200
```

**9.7 Hierarchical Inheritance**

We have discussed so far how inheritance can be used to modify a class when it did not satisfy the requirements of a particular problem on hand. Additional members are added through inheritance to extend the capabilities of a class. Another interesting application of inheritance is to use it as a support to the hierarchical design of a program. Many programming problems can be cast into a hierarchy where certain features of one level are shared by many others below that level.

As an example. Fig. 9.5 shows a hierarchical classification at students in a university. Another example could be the classification of account in a commercial bank as shown in Fig. 9.6. All the students have certain things in common and, similarly, all the accounts possess certain common features.

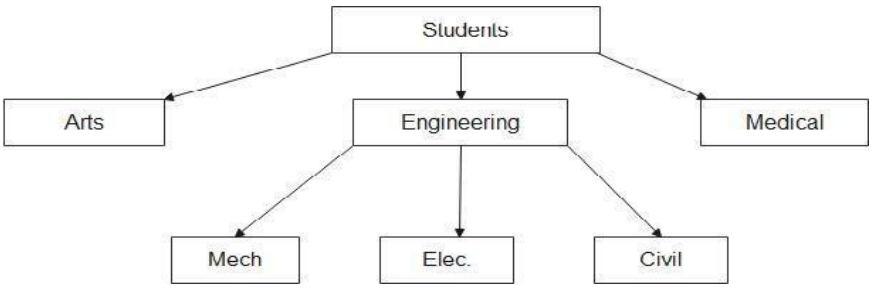



Fig. 9.5 Hierarchical Classification of students

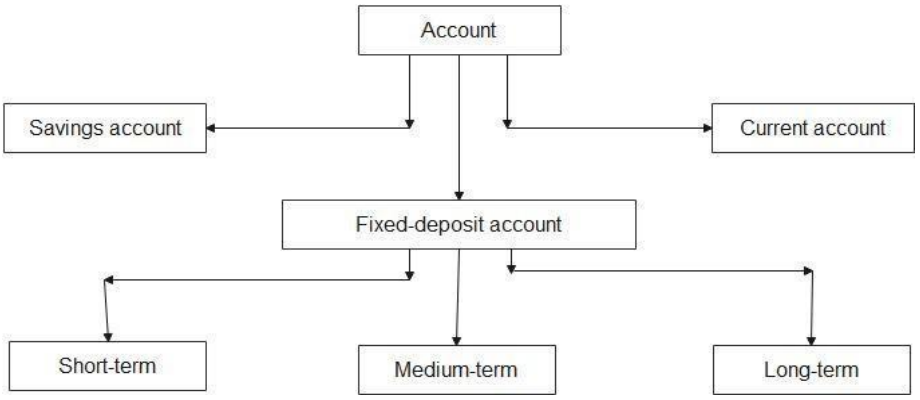



Fig. 9.6 Classification of bank accounts

In C++, such problems can be easily converted into class hierarchies. The base class will include all the features that are common to the sub classes. A subclass can be constructed by inheriting the properties of the base class. A subclass can serve as a base class for the lower level classes and so on.

### 9.8 Hybrid Inheritance

There could be situations where we need to apply two or more types of inheritance to design a program. For instance, consider the case of processing the student results discussed in See. 9.5, Assume that we have to give weightage for sports before finalizing the results. The weightage for

sports is stored in a separate class called sports. The new inheritance relationship between the various classes would be as shown in Fig. 9.7.

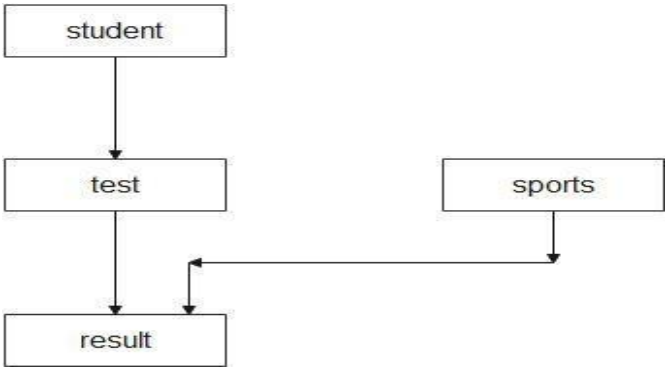



Fig. 9.7 Multilevel, multiple inheritance

The **sports class** might look like:

```
class sports
{
protected:
float score;

public:
void get_score(float);
void put_score(void);
};
```

The result will have both the multilevel and multiple inheritance and its declaration would be as follows:

```
class result: public test, public sports
{
……
……
};
```

where test itself is a derived class from student. That is,

```
class test : public student
{
…….
…….
};
```

Program 9.5 illustrates the implementation of both multilevel and multiple inheritance.

```
/*Hybrid inheritance */

#include<iostream>
#include<iomanip>

using namespace std;
class student
{
protected:
int roll_number;
public:
void get_number()
        {
        cout<<"\n Enter roll number";
        cin>>roll_number;
        }
void put_number()
        {
        cout<<"\n Roll number="<<setw(5)<<roll_number;
        }
};
class test : public student
{
protected:
float sub1,sub2;
public:
```

```
void get_marks()
        {
        cout<<"\n Enter marks in two subjects";
        cin>>sub1>>sub2;
        }

void put_marks()
        {
        cout<<"\n Mark obtained";
        cout<<"\n sub1="<<setw(15)<<sub1;
        cout<<"\n sub2="<<setw(15)<<sub2;
        }
};
class sports
{
protected:
float score;
public:
void get_score()
        {
        cout<<"Enter the score in sports";
        cin>>score;
        }
void put_score()
        { cout<<"\n score in sports="<<setw(5)<<score;
        }
};
class result : public test, public sports
{
float total;
public:
void display();
};
```

```
void result :: display()
        {
        total = sub1 + sub2 + score;
        put_number();
        put_marks();
        put_score();
        cout<<"\n Total score"<<setw(5)<<total;
        }
int main()
{
int i, n;
result student[10];
cout<<"\n How many students?";
cin>>n;
for (i=0;i<n;i++)
{
student[i].get_number();
student[i].get_marks();
student[i].get_score();
}
cout<<"\n Result\n";
for(i=0;i<n;i++)
student[i].display();
return 0;
}
```

The output of the program would be:

```
How many students?
2
Enter Roll number
111
```

```
Enter marks in two subjects
98
100
Enter the score in sports
90

Enter Roll number
222
Enter marks in two subjects
80
90
Enter the score in sports
95


Roll number    111
Marks obtained
Sub1 = 98
Sub2 = 100
score in sports = 90
Total score = 288

Roll number    222
Marks obtained
Sub1 = 80
Sub2 = 90
score in sports = 95
Total score = 265
```

### 9.9 Virtual Base Classes

We have just discussed a situation which would require the use of both the multiple and multilevel inheritance. Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance,

are involved. This is illustrated, in Fig. 9.8. The 'child' has two direct base classes 'parent1' and 'parents2' which themselves have a common base class 'grandparent'. The 'child' inherits the traits of 'grandparent' via two separate paths, It can also inherit directly as shown by the broke line. The 'grandparent' is sometimes referred to as indirect base c!ass.

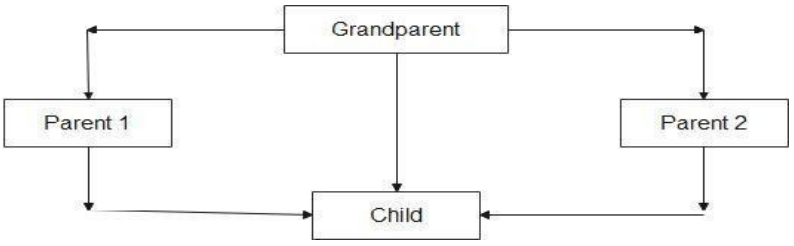



Fig. 9.8 Multipath inheritance

Inheritance by the 'child' as shown in Fig. 9.8 might pose some problems. All the public and protected members of 'grandparent' are inherited into 'child' twice, first via 'parent1' and again via 'parent2'. This means, 'child' would have duplicate sets of the members inherited from 'grandparent'. This Introduces ambiguity and should be avoided.

The duplication of inherited members due to these multiple paths can be avoided by making the common base class (ancestor class) as virtual base class while declaring the direct or intermediate base classes as shown below:

```
class A          // grandparent
{
……..
……..
};
class B1: virtual public A        //parent1
{
…….
…….
};
```

```
class B2: virtual public A        parent 2
{
……..
…….
};
class C : public B1, public B2        //child
{
……        //only one copy of A
……        //will be inherited
};
```

When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.

For example, consider again the student results processing system discussed in Sec. 9.8. Assume that the class sports derives the roll_number from the class student. Then, the inheritance relationship will be as shown in Fig. 9.9.

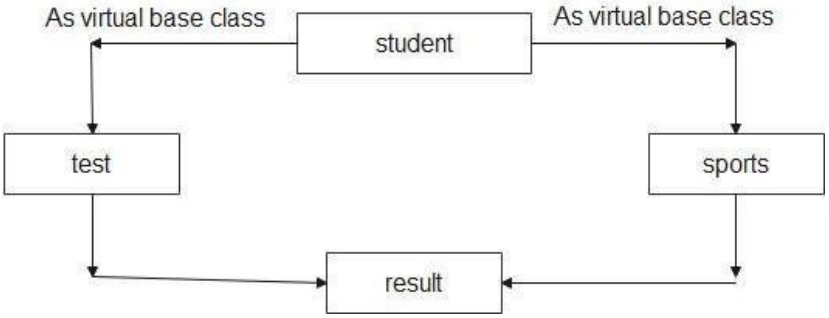



Fig. 9.9 Virtual base class

Program 9.6 illustrates the implemention of the virtual base class concept

/* program illustrating virtual base class */

```
#include<iostream>
#include<iomanip>

using namespace std;
class student
{
protected:
int roll_number;
public:
void get_number()
        {
        cout<<"\n Enter roll number";
        cin>>roll_number;
        }
void put_number()
        {
        cout<<"\n Roll number="<<setw(5)<<roll_number;
        }
};
class test:public virtual student
{
protected:
float sub1,sub2;
public:
void get_marks()
        {
        cout<<"\n Enter marks in two subjects";
        cin>>sub1>>sub2;
        }
```

```
void put_marks()
        {
        cout<<"\n Mark obtained";
        cout<<"\n sub1="<<setw(15)<<sub1;
        cout<<"\n sub2="<<setw(15)<<sub2;
        }
};
class sports:public virtual student
{
protected:
float score;
public:
void get_score()
        {
        cout<<"Enter the score in sports";
        cin>>score;
        }
void put_score()
        {
        cout<<"\n score in sports="<<setw(4)<<score;
        }
};
class result:public test,public sports
{
float total;
public:
void display();
};
void result :: display()
        {
        total=sub1+sub2+score;
        put_number();
        put_marks();
```

```
        put_score();
        cout<<"\n Total score"<<setw(11)<<total;
        }
int main()
{
int i,n;
result student[10];

cout<<"\n How many students?";
cin>>n;

for(i=0;i<n;i++)
        {
        student[i].get_number();
        student[i].get_marks();
        student[i].get_score();
        }

cout<<"\n Result\n";

for(i=0;i<n;i++)
        student[i].display();
return 0;
}
```

The output of the program would be:

```
How many students?

2

Enter Roll number
111
Enter marks in two subjects

90
90
```

Enter the score in sports

90

Enter Roll number

222

Enter marks in two subjects

80

80

Enter the score in sports

80

Roll number 111

Marks obtained

Sub1 = 90

Sub2 = 90

score in sports = 90

Total score = 270

Roll number 222

Marks obtained

Sub1 = 80

Sub2 = 80

score in sports = 80

Total score = 240

### 9.10 Abstract Classes

An abstract class is one that is not used to create objects. An abstract class is designed only to act as a base class (to be inherited by other classes).

It is a design concept in program development and provides a base upon which other classes may be built. In the previous example, the student class is an abstract class since it was not used to create any objects.

### 9.11 Constructors in Derived Classes

As we know, the constructors play an important role in initializing objects. We did not use them earlier in the derived classes for the sake of simplicity. One important thing to note here is that, as long as no base class constructor takes any arguments, the derived class need not have a constructor function. However, if any base class contains a constructor with one or more arguments, then it is *mandatory* for the derived class to have a constructor and pass the arguments to the base class constructors. Remember, while applying inheritance we usually create objects using the derived class. Thus, it makes sense for the derived class to pass arguments to the base class constructor. When both the derived and base classes contain constructors, the base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance, the base classes are constructed in the order in which they appear in the *declaration of the derived class*. Similarly, in a multilevel inheritance, the constructors will be executed *in the order of inheritance*.

Since the derived class takes the responsibility of supplying initial values to its base classes, we supply the initial values that are required by all the classes together, when a derived class object is declared. How are they passed to the base class constructors so that they can do their job? C++ supports a special argument passing mechanism for such situations.

The constructor of the derived class receives the entire list of values as its arguments and passes them on to the base constructors in the order in which they are declared in the derived class. The base constructors are called and executed before executing the statements in the body of the derived constructor,

The general form of defining a derived constructor is:

```
Derived-constructor (Arglist1, Arglist2, …. ArglistN, Arglist D)

Arglist1 → base1(arglist1)
Arglist2 → base2(arglist2)
……..
……
ArglistN →  baseN(arglistN)
{

Body of derived  Constructor

}
```

The header line of derived constructor function contains two parts separated by a colon(:). The first part provides the declaration of the arguments that are passed to the derived constructor and the second part lists the function calls to the base constructors.

base1(arglist1), base2(arglist2) ... are function calls to base constructors **base1(), base2()**,,… and therefore arglistl, arg!ist2, ... etc. represent the actual parameters that are passed to the base constructors. Arglist1 through ArglistN are the argument declarations for base constructors base1 through baseN. ArglistD provides the parameters that are necessary to initialize the members of the derived class.

Example:

```
D(lnt a1, int a2, float b1, float b2, int dl);
A(al, a2),        /* call to constructor A */
B(bl. b2) /* call to constructor B */
{
 d = d1;          // executes its own body
}
```

A(a1, a2) invokes the base constructor **A()** and B(bl, b2) invokes, another base constructor **B()**. The constructor **D()** supplies the values for these four arguments. In addition, it has one argument of its own. The constructor **D()** has a total of five arguments. **D()** may be invoked as follows:

……..

D objD(5, 12, 2.5. 7.54, 30);

…….

These values are assigned to various parameters by the constructor **D()** as follows:

5 → a1

12 → a2

2.5 → b1

7.54 → b2

30 → d1

The constructors for virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared. Any non-virtual bases are then constructed before the derived class constructor is executed. See Table 9.2.

Table 9.2 Execution of base class constructors

| Method of inheritance | Order of execution |
|---|---|
| class B: public A<br>{<br>}; | A() : Base constructor<br>B(): derived constructor |

| | |
|---|---|
| class A: public B, public C<br>{<br>}; | B() : Base first<br>C() : base second<br>A(): derived |
| class A: public B, virtual public C<br>{<br>}; | C() : virtual base<br>B() : ordinary base<br>A(): derived |

### 9.12 Member Classes, Nesting of Classes

Inheritance is the mechanism of deriving certain properties of one class into another. We have seen in detail how this is implemented using the concept of derived classes. C++ supports yet another way of inheriting properties of one class into another. This approach takes a view that an object can be a collection of many other objects. That is, a class can contain objects of other classes as its members as shown below:

```
class  alpha  {  };
class  beta  {  };
class gamma
{
alpha a;          // a is on object of alpha class
beta b;           // b is an object of beta  class
};
```

All objects of gamma class will contain the objects **a** and **b**. This kind of relationship is called *containership* or *nesting*. Creation of an object that contains another object is very different than the creation of an independent object. An independent object is created by its constructor when it is declared with arguments. On the other hand, a nested object is created in two stages.

First, the member objects are created using their respective constructors and then the other 'ordinary' members are created. This means, constructors of all the member objects should be called before its own constructor body is executed. This is accomplished using an initialization list in the constructor of the nested class.

Example:

```
class gamma
{
alpha a;                    // a is object of alpha
beta b;                     // b is object of beta
public:
gamma(arglist) : a(arglist1), b(arglist2)
{
        // constructor body )
}
```

arglist is the list of arguments that is to be supplied when a gamma object is defined. These parameters are used for initializing the members of **gamma**. arglist1 is the argument list for the constructor of **a** and arglist2 is the argument list for the constructor of **b**. arglist1 and arglist2 may or may not use the arguments from arglist. Remember, ***a****(arglist1)* and ***b****(arglist2)* are function cails and therefore the arguments do not contain the data types. They are simply variables or constants.

Example:

```
game(int x, int y, float z) : a(x), b(x, z)
{
Assignment section (for ordinary other members)
}
```

We can use as many member objects as are required in a class. For each member object, we add a constructor call in the initializer list. The constructors of the member objects are called in the order in which they are declared in the nested class.

**SUMMARY**

- The mechanism of deriving a new class from an old class is called inheritance. Inheritance provides the concept of reusability, The C+ + classes can be reused using inheritance.
- The derived class inherits some or all of the properties of the base class.
- A derived class with only one base class is called single inheritance.
- A class can inherit properties from more than one class which is known as multiple inheritance.
- A class can be derived from another derived class which is known as multilevel Inheritance.
- When the properties of one class are inherited by more than one class, it is called hierarchical inheritance.
- A private member of a class cannot be inherited either in public mode or in private mode.
- A protected member inherited in public mode becomes protected, whereas inherited in private mode becomes private in the derived class.
- A public member inherited in public mode becomes public, whereas inherited in private mode becomes private in the derived class.
- The friend functions and the member functions of a friend class can directly access the private and protected data.
- The member functions of a derived class can directly access only the protected and public data. However, they can access the private data through the member functions of the base class.
- Multipath inheritance may lead to duplication of inherited members from a 'grandparent' base class. This may be avoided by making the common base class a virtual base class.

- In multiple inheritance, the base classes are constructed in the order in which they appear in the declaration of the derived class.
- In multilevel inheritance, the constructors are executed in the order of inheritance,
- A class can contain objects of other classes. This is known as containership or nesting.

**Review Questions**

1. What does inheritance mean in C++?
2. What are the different forms of inheritance? Give an example for each.
3. Describe, the syntax of the single inheritance in C++.
4. We know that a private member of a base class is not inheritable. Is it anyway possible for the objects of a derived class to access the private members of the base class? If yes, how?
5. How do the properties of the following two derived classes differ?
    a. class Dl: private B({ }…);
    b. class D2 : public B({ }….);
6. When do we use the protected visibility specifier to a class member?
7. Describe the syntax of multiple inheritance. When do we use such an inheritance?
8. What are the implications of the following two definitions?
    a. class A: public B, public C{[ ]……};
    b. class A: public C. public B{[]……};
9. What is a virtual base class?
10. When do we make a class virtual?
11. What is an abstract class?
12. In what order are the class constructors called when a derived class object is created?

13. Class D is derived from class B. The class D does not contain any data members of its own. Does the class D require constructors? If yes, why?

14. What is containership? How does it differ from inheritance?

15. Describe how an object of a class that contains objects of other classes created?

## Programming Exercises

Assume that a bank maintains two kinds of accounts for customers, one called as savings account and the other as current account. The savings account provides compound interest and withdrawal facilities but no cheque book facility. The current account provides cheque book facility but no interest. Current account holders should also maintain a minimum balance and if the minimum balance falls below this level, a service charge is imposed.

Create a class account that stores customer name, account number and type of account. From this derive the classes cru_acct and sav_acct to make them more specific to their requirements. Include necessary member functions in order to achieve the following tasks:

(a) Accept deposit from a customer and update the balance.

(b) Display the balance.

(c) Compute and deposit interest

(d) Permit withdrawal and update the balance.

(e) Check for the minimum balance, impose penalty, necessary, and update the balance.

Do not use any constructors. Use member functions to initialize the class members.

# UNIT – X

## POINTERS, VIRTUAL FUNCTIONS AND POLYMORPHISM

### 10.1 Introduction

Polymorphism is one of the crucial features of OOP. It simply means 'one name, multiple forms'. We have already seen how the concept of polymorphism is implemented using the overloaded functions and operators. The overloaded member functions are 'selected' for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called *early binding* or *static binding* or *static linking*. Also known as compile time polymorphism, early binding simply means that an object is bound to its function call at compile time.

Now let us consider a situation where the function name and prototype is the same in both the base and derived classes. For example, consider the following class definitions:

```
class A
{
Int x;
public:
void show() {……..}        //show() in base class
};
class B: public A
{
int y;
public:
void show() {……}           //show() in derived class
};
```

How do we use the member function **show()** to print the values of objects of both the classes **A** and **B**?. Since the prototype of **show()** is the same in both the places, the function is not overloaded and therefore static binding does not apply. We have seen earlier that, in such situations, we may use the class resolution operator to specify the class while invoking the functions with the derived class objects.

It would be nice if the appropriate member function could be selected while the program is running. This is known as *run time polymorphism*. How could it happen? C++ supports a mechanism known as virtual function to achieve run time polymorphism.

At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as *late binding*. It is also known as *dynamic binding* because the selection of the appropriate function is done dynamically at run time.

Dynamic binding is one of the powerful features of C++. This requires the use of pointers to objects. We shall discuss in detail how the object pointers, and virtual functions are used to implement dynamic binding.

**10.2 Pointers**

Pointers is one of the key aspects of C++ language similar to that of C. As we know, pointers offer a unique approach to handle data in C and C++. We have seen some of the applications of pointers in unit 2 and unit 5. In this section, we shall discuss the rudiments of pointers and the special usage of them in C++,

We know that a pointer is a derived data type that refers to another data variable by storing the variable's memory address rather than data. A pointer variable defines where to get the value of a specific data variable instead of defining actual data.

Like C, a pointer variable can also refer to (or point to) another pointer in C++, However, it often points, to a data variable. Pointers provide an alternative approach to access other data objects.

**Declaring and Initializing Pointers**

As discussed in Unit 2, we can declare a pointer variable similar to other variables in C++. Like C, the declaration is based on the data type of the variable it points to. The declaration of a pointer variable takes the following form:

data-type *pointer-variable;

Here, pointer-variable is the name of the pointer, and the data-type refers to one of the valid C++ data types, such as int, char, float, and so an. The data-type is followed by an asterisk (*) symbol, which distinguishes a pointer variable from other variables to the compiler.

Note that we can locate asterisk (*) immediately before the pointer variable, or between the data type and the pointer variable, or immediately after the data type. It does not cause any effect in the execution process,

As we know, a pointer variable can point to any type of data available in C++. However, it is necessary to understand that a pointer is able to point to only one data type at the specific time. Let us declare a pointer variable, which points to an integer variable, as follows:

int 'ptr;

Here, **ptr** is a pointer variable and points to an integer data type. The pointer variable, ptr should contain the memory location of any integer variable. In the same manner, we can declare pointer variables for other data types also.

Like other programming languages, a variable must be initialized before using it in a C++ program. We can initialize a pointer variable as follows:

```
int *ptr, a;        // declaration
ptr = &a; // initialization
```

The pointer variable **ptr**, contains the address of the variable **a**. Like C, we use the 'address of ' operator or reference operator i,e, '&' to retrieve the address of a variable. The second statement assigns the address of the variable **a** to the pointer **ptr.**

We can also declare a pointer variable to point to another pointer, similar to that of C. That is, a pointer variable contains address of another pointer. Program 10.1 explains how to refer to a pointer's address by using a pointer in a C++ program.

```
//Example of using Pointers
# include<iostream>
using namespace std;

int main()
{
int a, 'ptr1 , **ptr2;
ptrl = &a;
ptr2 = &ptrl;
cout<<"The address of a : " « ptr1<< '\n";
cout << "The address of ptr1 : "<< ptr2;
cout<< " \n\n After incrementing the address values";
```

```
ptr1 = ptr1 + 2;
cout<<”The address of a :”<< ptr1<<”\n”;
ptr2 = ptr2 + 2;
cout << "The address of ptr1 : "<< ptr2;
}
```

The memory location in always addressed by the operating system. The output may vary depends on the system, Output of Program 10,1 would look like:

The address of a : ox8fb6fff4

The address of ptr1 : ox8fb6fff2

After incrementing the address values:

The address of a : ox8fb6fff8

The address of ptr1 : ox8fb6fff6

We can also use void pointers, known as generic pointers, which refer to variables of any data type. Before using void pointers, we must type cast the variables to the specific data types that they point to.

The pointers, which are not initialized in a program, are called Null pointers. Pointers of any data type can be assigned with one value i.e., ‘\0’ called null address.

**Manipulation of Pointers**

As discussed earlier, we can manipulate a pointer with the indirection operator, i.e. “ * ", which is also known as dereference operator. With this operator, we can indirectly access the data variable content. It takes the following general form:

*pointer-variable

As we know, dereferencing a pointer allows us to get the content of the memory location that the pointer points to. After assigning address of the variable to a pointer, we may want to change the content of the variable.

Using the dereference operator, we can change the contents of the memory location.

Let us consider an example that illustrates how to dereference a pointer variable. The value associated with the memory address is divided by 2 using the dereference operator. The division affects only the memory contents and not the memory address itself. Program 10.2 illustrates the use of dereference operator in C++.

```
//Manipulation of pointers
#include<iostream>
using namespace std;
void main()
{
int a = 10, *ptr;
ptr = &a;
cout<<"The value of a is : "<<a <<"\n";
*ptr = (*ptr)/2;
cout<< "The value of a after division by 2 is :"<<(*ptr)
}
```

The output of the program would be:

The value of a is: 10

The value of a after division by 2 is: 5

Care must be taken before dereferencing a pointer. It is essential to assign a value to the pointer. If we attempt to dereference an uninitialized pointer, it will cause runtime error by referring to any other location in memory.

**Pointer Expressions and Pointer Arithmetic**

As discussed in Unit 2, there are a substantial number of arithmetic operations that can be performed with pointers. C++ allows pointers to

perform the following arithmetic operations:

- A pointer can be incremented (++) (or) decremented (-- )
- Any integer can be added to or subtracted from a pointer
- One pointer can be subtracted from another

Example:

```
int a[6];
int *aptr;
aptr = &a[0];
```

Obviously, the pointer variable, **aptr,** refers to the base address of the variable **a**. We can increment the pointer variable, shown as follows:

```
aptr++          (or)      ++aptr
```

This statement moves the pointer to the next memory address. Similarly, we can decrement the pointer variable, as follows:

```
aptr -- (or) --aptr
```

This statement moves the pointer to the previous memory address. Also, if two pointer variables point to the same array can be subtracted from each other.

We cannot perform pointer arithmetic on variables which are not stored in contiguous memory locations, Program 10.3 illustrates the arithmetic operations that we can perform with pointers.

```
// ARITHMETIC OPERATIONS ON POINTERS
#include<iostream>
using namespace std;
int main()
{
Int num[] = {56, 75, 22, 18, 90};
Int *ptr;
```

```
int i;
cout<<”The array values are:\n”;
for (i=0; i<5; i++)
        cout<<num[i]<<”\n”;
/*initializing the base address of str to prt */
ptr = num;
/*printing the value in the array */
cout<<”\n vaule of ptr :”<<*ptr<<”\n”;
ptr++;
cout<<”\n vaule of ptr++ :”<<*ptr<<”\n”;
ptr--;
cout<<”\n vaule of ptr-- :”<<*ptr<<”\n”;
ptr = ptr + 2;
cout<<”\n vaule of ptr + 2 :”<<*ptr<<”\n”;
return 0;
}
```

Output of the program 10.3 would be:

The array values are:

56

75

22

18

90

value of ptr : 56

value of ptr++: 75

value of ptr--: 56

value of ptr + 2 : 22

**Using Pointers with Arrays and Strings**

Pointer is one of the efficient tools to access elements of an array. Pointers are useful to allocate arrays dynamically, ie, we can decide the array size at run time. To achieve this, we use the functions, namely **malloc()** and **calloc().** Accessing an array with pointers in simpler than accessing the array index.

In general, there are some differences between pointers and arrays. array refer to a block of memory space, whereas pointers do not refer to any section of memory. The memory addresses of arrays cannot be changed, whereas the content of the pointer variables, such as the memory addresses that it refer to, can be changed.

Even though there are subtle difference between pointers and arrays, they have a strong relationship between them.

Note that there is no error checking of array bounds in. C++. Suppose we declare an array of size 25. The compiler issues no warnings if we attempt to access 26th location. It is the programmer's task to cheek the array limit.

We can declare the pointer to arrays as follows:

```
int *nptr;
nptr = number[0];
        or
nptr = number
```

Here, **nptr** points to the first element of the integer array, number[0]. Also, consider the following example:

```
float *fptr;
fptr = price[0];
        or
fptr = price;
```

Here, **fptr** points to the first element of the array of float, price[0]. Let us consider an example of using pointers to access an array of numbers and

sum up the even numbers of the array. Initially, we accept the count as an input to know the number of inputs from the user. We use pointer variable, ptr to access each element of the array. The inputs are checked to identify the even numbers. Then the even numbers are added, and stored in the variable, Sum. If there Is no even number in the array, the output will be 0, Program 10.4 illustrates how to access the array contents using pointers,

```
//Pointers with Arrays

#include<iostream>
using namespace std;
int main()
{
Int numbers[50], *ptr;
Int n,i;
cout<<"\n Enter the count\n";
cin>>n;
cout<<"\nEnter the numbers one by one\n";
for (i=0; i<n; i++)
cin>>number[i];
/*assigning the base address of numbers to ptr and initializing the sum
to 0 */
ptr = numbers;
int sum = 0;
/*check out for even inputs and sum up them *'
for (i=0; i<n; i++)
        {
        if (*ptr % 2 == 0) sum = sum + *ptr;
        ptr++;
        }
cout<< "\n Sum of even numbers: "<<sum;
return 0;
}
```

Output of the program 10.4 is:

```
Enter the count
4
Enter the numbers one by one
10
15
20
40
Sum of even numbers: 70
```

**Array of Pointers**

Similar to other variables, we can create an array of pointers in C++. The array of pointers represents a collection of addresses. By declaring array of pointers, we can save a substantial amount of memory space.

An array of pointers point to an array of data items. Each element of the pointer array points to an item of the data array. Data items can be accessed either directly or by dereferencing the elements of pointer array. We can reorganize the pointer elements without affecting the data items.

We can declare an array of pointers as follows:

```
int *inarray[10];
```

This statement declares an array of 10 pointers, each of which points to an integer. The address of the first pointer is inarray[0], and the second pointer is inarray[1], and the final pointer points to inarray|9|. Before initializing, they point to some unknown values in the memory space. We can use the pointer variable to refer to some specific values, Program 10.5 explains the implementation of array of pointers.

```
//Array of pointers
#include<iostream>
#include<cstring>
#include<conio.h>
#include<ctype.h>
using namespace std;
void main()
{
int i=0;
char *ptr[10] = {“books”, “television”,”computer”,”sports”};
char str[25];
clrscr();
cout<<”\n Enter your favorite leisure pursuit ”;
cin>>str;
for (i=0; i<4; i++)
        {
        If (!strcmp(str, *ptr[i]))
                {
                cout<<”\n your favorite pursuit is available here”
                break;
                }
        }
if (i==4)
cout<<”\n Your favorite persuit is not available here”;
getch();
}
```

The output of the program 10.5 is

```
Enter your favorite leisure persuit: book

your favorite persuit is available here.
```

**Pointers and Strings**

We have seen the usage of pointers with one dimensional array elements. However, pointers are also efficient to access, two dimensional and multi-dimensional arrays in in C++. There is a definite relationship between arrays, and pointers. C++ also allows us to handle the special kind of arrays, i.e. string with pointers.

We know that a string is one dimensional array of characters, which start with the index 0 and ends with the null character '\0' in C++. A pointer variable can access a string by referring to its first character. As we know, there are two ways to assign a value to a string. We can use the character array or variable of type char *. Let us consider the following string declarations:

char num[ ] = "one";

const char *numptr = "one":

The first declaration creates an array of four characters, which contains the characters 'o', 'n', 'e', '\0', whereas the second declaration generates a pointer variable, which points to the first character,, i.e. 'o' of the string, There is numerous string handling functions available in C++. All of these functions are available in the header file <cstring>.

Program 10.6 shows how to reverse a string using printers and arrays.

```
//Accessing string using pointers and arrays

#include<iostream>
#include<cstring>
using namespace std;
void main()
{
char str[] = "Test";
int leg = strlen(str);
```

```
for (int i=0; i<len; i++)
        {
        cout<<str[i] <<i[str]<<*(str+i) <<*(i+str);
        }
cout<<endl;
//string reverse
int lenM = len/2;
len--;
for (i=0; i<lenM; i++)
        {
        str[i] = str[i] + str[len-i];
        str[len-i] = str[i] – str[len-i];
        str[i] = str[i] – str[len-i];
        }
cout<<”The string reversed :”<<str;
}
```

Output of program is:

TTTTeeeessssttttt

The string reversed : tseT

**Pointers to Functions**

Even though pointers to functions (or function pointers) are introduced in C, they are widely used in C++ for dynamic binding, and event-based applications. The concept of pointer to function acts as a base for pointers to members, which we have discussed in Unit 5.

The pointer to function is known as call back function. We can use these function pointers to refer to a function. Using function pointers, we can allow a C++ program to select a function dynamically at run time. We can also pass a function as an argument to another function. Here, the function is passed as a pointer. The function pointers cannot be dereferenced. C++ also allows us to compare two function pointers.

C++ provides two types of function pointers: function pointers that point to static member functions and function pointers that point to non-static member functions. These two function pointers are incompatible with each other. The function pointers that point to the non-static member function requires hidden argument.

Like other variables, we can declare a function pointer in C++. It takes the following:

data_type (*function_name()();

As we know, the data_type is any valid data types used in C++. The function_name is the name of a function, which must be preceded by an asterisk (*). The function_name is any valid name of the function.

Example:

int (*num_function(int x));

Note that declaring a pointer only creates a pointer. It does not create actual function. For this, we must define the task, which is to be performed by the function. The function must have the same return type and arguments. Program 10.7 explains how to declare and define function pointers in C++

```
//Pointers to functions
#include<iostream.h>
typedef void (*FunPtr)(int, int);
void add(int i, int j)
{
cout<< i<< " + "<< j << " = "<< i + j;
}
void subtract(int i, int j)
{
cout<< i<< " - "<< j << " = "<< i - j;
}
```

```
void main()
{
FunPtr ptr;
ptr = &add;
ptr(1, 2);
cout<<endl;
ptr = &subtract;
ptr(3, 2);
}
```

Output of program 10.7 is:

1 + 2 = 3

3 – 2 = 1

**10.3 Pointers to Objects**

We have already seen how to use pointers to access the class members, As stated earlier, a pointer can point to an object created by a class. Consider the following statement:

```
item x;
```

where **item** is a class and x is an object defined to be of type item. Similarly we can define a pointer it_ptr of type item as follows:

```
item *it_ptr;
```

Object pointers are useful in creating objects at run time. We can also use an object pointer to access the public members of an object. Consider a class item defined as follows:

```
class item
{
int code;
float price;
```

```
public:
void getdata (int a, float b)
        {
        code = a;
        price = b;
        }
void show(void)
        {
        cout<<"Code = "<<code<<"\n"
        <<"Price ="<<price;
        }
};
```

Let us declare an **item** variable **x** and a pointer **ptr** to **x** as follows:

```
item x;
item *ptr = &x;
```

The pointer **ptr** is initialized with the address of x.

We can refer to the member functions of **item** in two ways, one by using the dot operator and the object, and another by using the arrow operator and the object pointer. The statements

```
x.getdata(100,75.50);
x.show();
```

are equivalent to

```
ptr -> getdata(100, 76.50);
ptr -> show();
```

Since ***ptr** is on alias of **x**, we can also use the following method:

```
(*ptr).show();
```

The parentheses are necessary because the dot operator has higher precedence than the indirection operator*.

We can also create the objects using pointers and **new** operator as follows:

item *ptr = **new** item;

This statement allocates enough memory for the data members in the object structure and assigns the address of the memory space to **ptr**, Then **ptr** can be used to refer to the members as shown below:

ptr -> show();

If a class has a constructor with arguments and does not include an empty constructor, then we must supply the arguments when the object is created.

We can also create an array of objects using pointers. For example, the statement

item *ptr = new item[10]; // array of 10 objects

creates memory space for an array of 10 objects of **item**. Remember, in such cases, if the class contains constructors, it must also contain an empty constructor. Program 10.8 illustrates the use of pointers to objects.

```
//Pointer to Objects
#include <iostream>
using namespace std;
class item
{
int code;
float price;
```

```
public:
void getdata(int a, float b)
        {
        code = a; price = b;
        }
void show()
        {
        cout<<"code ="<< code <<"\n";
        cout<<"price = "<<price;
        }
};
const int size = 2;
int main()
{
item *p = new item[size];
item *d = p;
int x,i;
float y;
for (i=0; i<size; i++)
        {
        cout<<"Input code and price for item"<< i+1;
        cin>>x>>y;
        p -> getdata(x,y);
        p++;
        }
for (i=0; i<size; i++)
        {
        cout<<"Item "<< i+1<<"\n";
        d->show();
        d++;
        }
return 0;
}
```

The output of the program 10.8 would be

Input code and price for item 1 40 500

Input code and price for item 2 60 700

Item 1

code 40

price 500

Item 2

code 60

price 700

In Program 10.8 we created space dynamically for two objects of equal size. But this may not be the case always. For example, the objects of a class that contain character strings would not be of the same size. In such cases, we can define an array of pointers to objects that can be used to access the individual objects. This is illustrated in Program 10.9.

```
//Array of pointers to objects
#include<iostream>
#include<cstring>
using namespace std;
class city
{
protected:
char *name;
int len;
```

```
public:
   city()
        {
        len = 0;
        name = new char[len + 1];
        }
void getname(void)
        {
        char *s;
        s = new char[30];
        cout<<"Enter city name:";
        cin>>s;
        len = strlen(s);
        name = new char[len+1];
        strcpy(name, s);
        }
void printname(void)
        {
        cout<<name<<"\n";
        }
};
int main()
{
city *cptr[10];              //array of 10 pointers to cities
int n= 1;
int option;
do
{
cptr[n] = new city;       //create new city
cptr[n] -> getname();
n++;
cout<<"Do you want to enter one more name?\n";
cout<<"Enter 1 for yes and 0 for no";
```

```
cin>>option;
}while(option);

cout<<"\n\n";
for (int i=1; i<=n; i++)
        {
        cptr[i]->printname();
        }
return 0;
}
```

The output of program 10.9 would be

```
Enter city name: Chennai
Do you want to enter one more name?
Enter 1 for yes and 0 for no 1
Enter city name: Madurai
Do you want to enter one more name?
Enter 1 for yes and 0 for no 0
Chennai
Madurai
```

### 10.4 This Pointer

C++ uses a unique keyword called **this** to represent an object that invokes a member function. **this** is a pointer that points to the object for which *this* function was called. For example, the function call **A.max()** will set the pointer **this** to the address of the object **A**. The starting address is the same as the address of the first variable in the class structure.

This unique pointer is automatically passed to a member function when it is called. The pointer **this** acts as an *implicit* argument to all the *member* functions.

Consider the following simple example:

```
class ABC
{
int a;
…….
……
};
```

The private variable ‘a’ can be used directly inside a member function, like

```
a = 123;
```

We can also use the following statement to do the same job

```
this ->a = 123:
```

Since C++ permits the use of shorthand form **a = 123,** we have not been using the pointer **this** explicitly so far. However, we have been implicitly using the pointer **this** when overloading the operators using member function.

Recall, that, when a binary operator is overloaded using a member function, we pass only one argument to the function. The other argument is implicitly passed using the pointer **this**. One important application of the pointer **this** is to return the object it points to. For example, the statement

```
return *this;
```

inside a member function will return the object that invoked the function. This statement assumes importance when we want to compare two or more objects inside a member function and return the *invoking object* as a result,

Example:

```
person & person :: greater (person & x)
{
if x.age > age
        return x;                   //argument object
  else
        return *this;               //invoking object
}
```

Suppose we invoke this function by the call

```
max = A.greater(B);
```

The function will return the object **B** (argument object) if the age of the person **B** is greater than that of **A**, otherwise, it will return the object **A** (invoking object) using the pointer **this**. Remember, the dereference operator * produces the contents at the address contained in the pointer. A complete program to illustrate the use of this is given in Program 10.10.

```
//this POINTER
#include<iosteam>
#include<cstring>
using namespace std;
class person
{
char name[20];
float age;
```

```
public:
person(char *s, float a)
        {
        strcpy(name, s);
        age = a;
        }
person & person :: greater(person & x)
        {
        if (x.age >= age)
                return x;
        else
                return *this;
        }
void display(void)
        {
        cout<<"Name:  "<<name<<"\n"
                <<"Age: "<<age<<"\n";
        }
};
int main()
{
person P1("John",37.5), P2("Guru", 30.2), P3("Samy", 40);
person P = P1.greater(P3);
cout<<"Elder person is:";
P,display();
P = P1.greater(P2);
cout<<"Elder person is:";
P,display();

return 0;
}
```

The output of the program 10.10 would be

Elder person is:

Name: Samy

Age : 40

Elder person is:

Name: John

Age : 37.5

### 10.5 Pointers to Derived Classes

We can use pointers not only to the base objects but also to the objects of derived classes. Pointers to objects of a base class are *type-compatible* with pointers to objects of a derived class. Therefore, a single pointer variable can be made to point to objects belonging to different classes. For example, if **B** is a base class and **D** is a derived class from **B**, then a pointer declared as a pointer to **B** can also be a pointer to **D**. Consider the following declarations

```
B *cptr; // pointer to class B type variable
B b; // base object
D d;     // derived object
cptr = &b;        // cptr points to object b
```

We can make **cptr** to point to the object **d** as follows:

```
cptr = &d; // cptr points to object d
```

This is perfectly valid with C++ because **d** is an object derived from the class **B**.

However, there is a problem in using **cptr** to access the public members of the derived class **D**. Using **cptr**, we can access, only those members which are inherited from **B** and not the members that originally belong to **D**. In case a member of **D** has the same name as one of the

members of B, then any reference to that member by **cptr** will always access the base class member.

Although C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. We may have to use another pointer declared as pointer to the derived type.

Program 10.11 illustrates how pointers to a derived object are used.

```
//Pointers to Derived Objects
#include<iostream>
using namespace std;
class BC
{
Public:
int b;
void show()
        {
        cout<<"b ="<<b<<"\n";
        }
};
class DC : public BC
{
public:
int d;
void show()
        {
        cout<<"b ="<<b<<"\n"
                <<"d = "<<d<<"\n";
        }
};
```

```
int main()
{
BC *bptr;                    //base pointer
BC base;
bptr = &base;                // base address

bptr -> b= 100;                        //access BC via base pointer
cout<<"bptr points to base object \n";
bptr ->show();

//derived class
DC derived;
bptr = &derived;             //address of derived object
bptr -> b = 200
/*bptr -> d = 300; */ won't work
cout<< "bptr now points to derived object \n";

bptr ->show();               //bptr now points to derived object
/*accessing d using a pointer of type derived class DC */
DC *dptr;                    //derived type pointer
dptr = &derived;
dptr -> d = 300;
cout<<"dptr is derived type pointer\n";
dptr ->show();

cout<<"using (DC *)bptr)\n";
((DC *)bptr) ->d = 400;
((DC *)bptr) ->show();
return 0;
}
```

Program 10.11 produces the following output

```
bptr points base object
b = 100
bptr now points derived object
b = 200
dptr is derived type pointer
b = 200
d = 300
using ((DC *)bptr)
b = 200
d = 400
```

### 10.6 Virtual Functions

As mentioned earlier, polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes. This necessitates the use of a single pointer variable to refer to the objects of different classes. Here, we use the pointer to base class to refer to all the derived objects. But, we just discovered that a base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class. The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer. How do we then achieve polymorphism? It is achieved using what is known as 'virtual' functions.

When we use the same function name in both the base and derived classes, the function in. base class is declared an *virtual* using the keyword **virtual** preceding its normal declaration. When a function is made virtual, C++ determines which function to use at run time based on the type of object

pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the **virtual** function. Program 10.12 illustrates this point.

```
//virtual Functions
#include<iostream>
using namespace std;
class Base
{
public:
void display()
        {
        cout<<"\n Display Base";
        }
virtual void show()
        {
        cout<<"\n Show Base";
        }
};

class Derived : public Base
{
public:
void display()
        {
        cout<<"\n Display Derived";
        }
virtual void show()
        {
        cout<<"\n Show Derived";
        }
};
```

```
int main()
{
Base B;
Derived D;
Base *bptr;

cout<<"\n bptr points to Base\n";
bptr = &B;

bptr -> display();        //calls base version

bptr -> show();           //calls base version

cout<<"\n bptr points to derived\n";
bptr = &D;

bptr -> display();        //calls base version
bptr -> show();           //calls derived version

return 0;
}
```

The output of the program 10.12 would be

```
bptr points to Base

Display Base

Show Base

bptr points to Derived

Display Base

Show Derived
```

One important point to remember is that, we must access virtual functions through the use of a pointer declared as a pointer to the base class. Why can't we use the object name (with the dot operator) the same way as any other member function to call the virtual functions?. We can, but remember, run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

**Rules for Virtual Functions**

When virtual functions are created for implementing late binding, we should, observe some basic rules that satisfy the compiler requirements:

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored,
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object,

10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

### 10.7 Pure Virtual Functions

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a placeholder. For example, if we do not define any object of base class media then the function display() in the base class has been defined 'empty' . Such functions are called “do-nothing" functions.

A "do-nothing" function may be defined an follows:

```
virtual void display() = 0;
```

Such functions are called *pure virtual* functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or re-declare it as a pure virtual function. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called abstract base classes, The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

## SUMMARY

- Polymorphism simply means one name having multiple forms.
- There are two types of polymorphism, namely, compile time polymorphism and run time polymorphism.
- Functions and operators overloading are examples of compile time polymorphism. The overloaded member functions are selected for invoking by matching arguments, both type and number. The compiler

knows this information at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early or static binding or static linking. It means that an object is bound to its function call at compile time.

- In run time polymorphism, an appropriate member function is selected while the program is running. C++ supports run time polymorphism with the help of virtual functions. It is called late or dynamic binding because the appropriate function is selected dynamically at run time. Dynamic binding requires use of pointers to objects and is one of the powerful features of C++.
- Object pointers are useful in creating objects at run time. It can be used to access the public members of an object, along with an arrow operator.
- A **this** pointer refers to an object that currently invokes a member function. For example, the function call a.show() will set the pointer 'this' to the address of the object '**a**'.
- Pointers to objects of a base class type are compatible with pointers to objects of a derived class. Therefore, we can use a single pointer variable to point to objects of base class as well as derived classes.
- When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer rather than the type of the pointer. By making the base pointer to point to different objects, we can execute different versions of the virtual function.
- Run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class. It cannot be achieved using object name along with the dot operator to access virtual function,
- We can have virtual destructors but not virtual constructors.
- If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, the respective calls will invoke the base class function.
- A virtual function, equated to zero is called a pure virtual function. It is a function declared in a base class that has no definition relative to the base class. A class containing such pure function is called an abstract class.

**Review Questions**

1. What does polymorphism mean in C++ language?
2. How is polymorphism achieved at (a) compile time, and (b) run time?
3. Discuss the different ways by which we can access public member functions of an object
4. Explain, with an example, how you would create space for an array of objects using pointers.
5. What does this pointer point to?
6. What are the applications of thin pointer?
7. What is a virtual function?
8. Why do we need virtual functions ?
9. When do we make a virtual function "pure"? What are the implications of making a function a pure virtual function?

**Programming Exercises**

Create a base class called **shape**. Use this class to store two double type values that could be used to compute the area of figures. Derive two specific classes called **triangle** and **rectangle** from the base shape. Add to the base class, a member function **get_data()** to initialize base class data members and another member function **display_area()** to compute and display the area of figures. Make **display_area()** as a virtual function and redefine this function in the derived classes to suit their requirements. Using these three classes, design a program that will accept dimensions of a triangle or a rectangle interactively, and display the area. Remember the two values given as input will be treated an lengths of two sides in the case of rectangles, and as base and height in the case of triangles, and used as follows: Area of rectangle - x * y Area of triangle - 1/2 * x * y.