



GOVERNMENT ARTS AND SCIENCE COLLEGE

NAGERCOIL – 629 004

[Affiliated to Manonmaniam Sundaranar University, Tirunelveli – 12]

DEPARTMENT OF PHYSICS

COURSE MATERIAL

NAME OF THE SUBJECT : COMPUTER PROGRAMMING IN C++

SUBJECT CODE : SMPH52

YEAR : III B.Sc. PHYSICS

SEMESTER : V

STAFF IN-CHARGE : Dr. G.M.CARMEL VIGILA BAI
Head, Department of Physics
Government Arts and Science College
NAGERCOIL.

Email : gmcarmelgasc@gmail.com

Mobile No.: 9487819696

PAPER HANDLED BY : Dr. G.M.CARMEL VIGILA BAI

BOOK FOR REFERENCE : Object Oriented Programming with C++ -
E.Balagurusamy, Tata Mc Graw-Hill publishing company Ltd. New Delhi.

ACKNOWLEDGEMENT

I express my thanks to the staff members, students and my family members who helped me in some way to make this course material.

Also, I express my sincere thanks to the authors of various books, specially to Prof. E.Balagurusamy, Author of Object Oriented Programming with C⁺⁺ - Tata Mc Graw-Hill publishing company Ltd., New Delhi.

(Dr. G.M.CARMEL VIGILA BAI)

CONTENTS

S.No.	Topic	Page No.
1	M.S.University Syllabus	i
2	UNIT I	1
3	UNIT II	17
4	UNIT III	34
5	UNIT IV	86
6	UNIT V	112
7	Unit Test, Internal Test, Model Exam Question Papers	137
8	M.S. University previous year Question Papers	143

PAPER VIII**COMPUTER PROGRAMMING IN C++**

L	T	P	C
4	0	0	4

Preamble: Objective of the course is to provide knowledge about the basics of Computer programming in C++ and to solve problems by writing programs. The paper does not need any special prerequisite and the learners are expected to come out with the ability to apply the computer language C++ to solve problems .

UNIT-I: WHAT IS C++

Introduction - tokens - keywords - identifiers and constants - declaration of variables - basic data types - user defined data types-derived data types - symbolic constants - operators in C++ -expressions and their type-hierarchy of arithmetic operators- scope resolution operator – declaring, initializing and modifying variables-special assignment operators - all control structures-structure of a simple

C ++ program (11L)

UNIT-II: ARRAYS AND FUNCTIONS IN C++

Introduction - one dimensional and two dimensional arrays-initialization of arrays-array of strings

Functions-introduction-function with no argument and no return values-function with no argument but return value - function with argument and no return values- function with argument and return values- call by reference-return by reference- function prototyping - inline functions - local, -global and static variables- -function overloading - virtual functions-main function-math library

functions. (13L)

UNIT-III: CLASSES AND OBJECTS

Introduction - specifying a class - defining member functions-C⁺⁺ program with class - nesting of member functions - private member functions - objects as function arguments - arrays within a class-array of objects-static class membersfriend functions-constructors - parameterized constructors-multiple constructors - constructors with default arguments - copy constructor. (15L)

UNIT-IV: OPERATOR OVERLOADING, INHERITANCE AND POINTERS

Introduction -defining operator overloading - overloading unary operators - binary operators.

Inheritance - single inheritance - multiple inheritance - multilevel inheritance - hybrid inheritance - hierarchial inheritance-virtual base class-abstract class

Pointers- definition-declaration- arithmetic operations. (12L)

UNIT-V: MANAGING CONSOLE I/O OPERATIONS

Introduction - C⁺⁺ stream - C⁺⁺ stream classes - unformatted I/O Operations formatted console I/O operations - working with files - classes for file steam operations - opening and closing a file - file pointers and their manipulations. (9L)

Books for study

1. Object oriented Programming with C⁺⁺ - E.Balagurusamy, Tata Mc Graw-Hill publishing company Ltd. New Delhi

Books for reference

1. Programming with C⁺⁺ - D.Ravichandran, Tata Mc Graw-Hill publishing company Ltd. New Delhi .
2. Object oriented Programming in C⁺⁺ 4th Edn.Robert Lafore-Macmilan publishing company Ltd.
3. Fundamentals of Programming with C⁺⁺ -Richardl.Halterman

**DEPARTMENT OF PHYSICS,
GOVERNMENT ARTS AND SCIENCE COLLEGE,
NAGERCOIL**

SMPH52 - COMPUTER PROGRAMMING IN C++

-Dr.G.M.CARMEL VIGILA BAI

UNIT-I:

WHAT IS C++: Introduction - tokens - keywords - identifiers and constants - declaration of variables - basic data types - user defined data types-derived data types - symbolic constants - operators in C++ -expressions and their type-hierarchy of arithmetic operators- scope resolution operator – declaring, initializing and modifying variables-special assignment operators - all control structures-structure of a simple C ++ program.

Introduction to C++

C++ is an object oriented programming language. It was developed by Bjarne Stroustrup in 1979 at Bell Laboratories in Murray Hill, New Jersey. He initially called the new language "C with Classes." However, in 1983 the name was changed to C++.

C++ is a superset of C. Stroustrup built C++ on the foundation of C, including all of C's features, attributes, and benefits. Most of the features that Stroustrup added to C were designed to support object-oriented programming .These features comprise of classes, inheritance, function overloading and operator overloading. C++ has many other new features as well, including an improved approach to input/output (I/O) and a new way to write comments.

C++ is used for developing applications such as editors, databases, personal file systems, networking utilities, and communication programs. Because C++ shares C's efficiency, much high-performance systems software is constructed using C++.

C++ Tokens

Smallest individual units in a program are known as Tokens. Tokens of C++ are keyword, identifiers, constants, strings, and operators.

C++ keywords

When a language is defined, one has to design a set of instructions to be used for communicating with the computer to carry out specific operations. The set of instructions which are used in programming, are called keywords. These are also known as reserved words of the language. They have a specific meaning for the C++ compiler and should be used for giving specific instructions to the computer. These words cannot be used for any other purpose, such as naming a variable. C++ is a case-sensitive language, and it requires that all keywords be in lowercase. C++ keywords are:

asm	insert an assembly instruction
auto	declare a local variable
bool	declare a boolean variable
break	break out of a loop
case	a block of code in a switch statement
catch	handles exceptions from throw
char	declare a character variable
class	declare a class
const	declare immutable data or functions that do not change data
const_cast	cast from const variables
continue	bypass iterations of a loop
default	default handler in a case statement
delete	make memory available
do	looping construct
double	declare a double precision floating-point variable
dynamic_cast	perform runtime casts
else	alternate case for an if statement
enum	create enumeration types
explicit	only use constructors when they exactly match
export	allows template definitions to be separated from their declarations
extern	tell the compiler about variables defined elsewhere
false	the boolean value of false
float	declare a floating-point variable
for	looping construct
friend	grant non-member function access to private data
goto	jump to a different part of the program
if	execute code based off of the result of a test
inline	optimize calls to short functions
int	declare a integer variable
long	declare a long integer variable
mutable	override a const variable
namespace	partition the global namespace by defining a scope
new	allocate dynamic memory for a new variable
operator	create overloaded operator functions
private	declare private members of a class
protected	declare protected members of a class
public	declare public members of a class
register	request that a variable be optimized for speed
reinterpret_cast	change the type of a variable
return	return from a function
short	declare a short integer variable
signed	modify variable type declarations
sizeof	return the size of a variable or type
static	create permanent storage for a variable
static_cast	perform a nonpolymorphic cast
struct	define a new structure

switch	execute code based off of different possible values for a variable
template	create generic functions
this	a pointer to the current object
throw	throws an exception
true	the boolean value of true
try	execute code that can throw an exception
typedef	create a new type name from an existing type
typeid	describes an object
typename	declare a class or undefined type
union	a structure that assigns multiple variables to the same memory location
unsigned	declare an unsigned integer variable
using	import complete or partial namespaces into the current scope
virtual	create a function that can be overridden by a derived class
void	declare functions or data with no associated data type
volatile	warn the compiler about variables that can be modified unexpectedly
wchar_t	declare a wide-character variable
while	looping construct

Identifiers

An identifier is a name assigned to a function, variable, or any other user-defined item. Identifiers can be from one to several characters long. An identifier is a name for a variable, constant, function, etc. It consists of a letter followed by any sequence of letters, digits, and underscores.

Examples of valid identifiers: First_name, age, y2000, y2k

Examples of invalid identifiers: 2000y

Identifiers cannot have special characters in them. For example: X=Y, J-20, ~Ricky,*Michael are invalid identifiers.

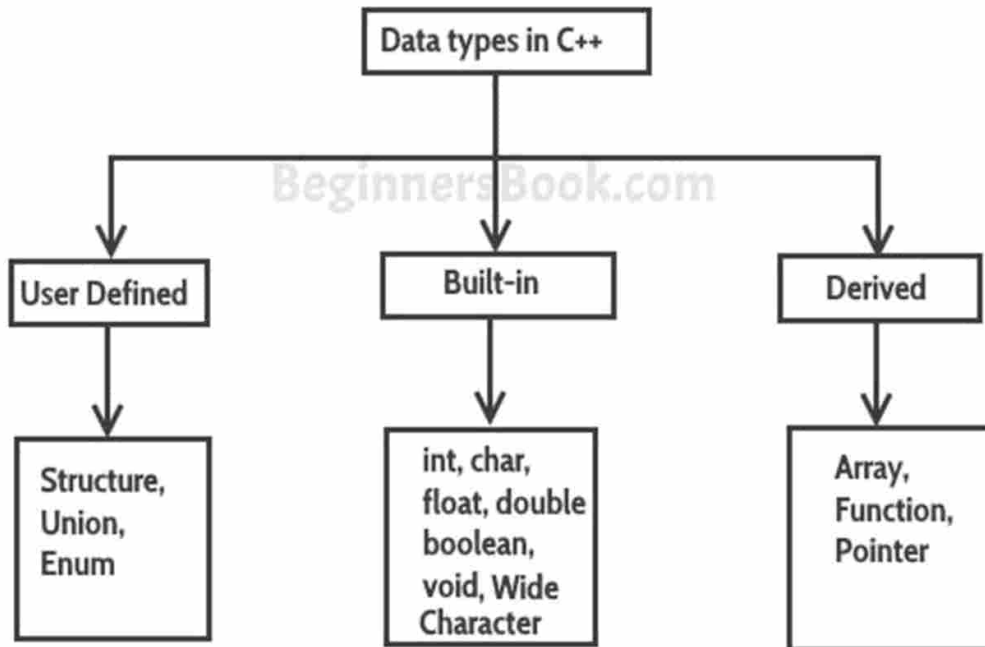
Rules for naming identifiers:

- Variable names can start with any letter of the alphabet or an underscore. Next comes a letter, a digit, or an underscore.
- Uppercase and lowercase are distinct.
- C++ keywords cannot be used as identifier.

Data types

Data type defines size and type of values that a variable can store along with the set of operations that can be performed on that variable. Data types are classified as

- User Defined data types
- Built in data types
- Derived data types



C++ provides built-in data types that correspond to Structure, Union and Enumeration.

C++ provides built-in data types that correspond to Array, Function and Pointer.

C++ provides built-in data types that correspond to integers, characters, floating-point values, and Boolean values. There are the seven basic data types in C++ as shown below:

Type Meaning

- Char (character) holds 8-bit ASCII characters
- wchar_t (Wide character) holds characters that are part of large character sets
- int (Integer) represent integer numbers having no fractional part
- float (floating point) stores real numbers in the range of about 3.4×10^{-38} to 3.4×10^{38} , with a precision of seven digits.
- Double (Double floating point) Stores real numbers in the range from 1.7×10^{-308} to 1.7×10^{308} with a precision of 15 digits.
- Bool (Boolean) can have only two possible values: true and false.
- Void Valueless

C++ allows certain of the basic types to have modifiers preceding them. A modifier alters the

meaning of the base type so that it more precisely fits the needs of various situations. The data type modifiers are: signed, unsigned, long and short

Table 3.2 *Size and range of C++ basic data types*

Type	Bytes	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-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

This Table 3.2 shows all combinations of the basic data types and modifiers along with their size and range for a 16-bit word machine

Variable

A variable is a named area in memory used to store values during program execution. Variables are run time entities. A variable has a symbolic name and can be given a variety of values. When a variable is given a value, that value is actually placed in the memory space assigned to the variable. All variables must be declared before they can be used.

The general form of a declaration is:

```
type variable_list;
```

Here, type must be a valid data type plus any modifiers, and variable_list may consist of one or more identifier names separated by commas. Here are some declarations:

```
int i,j,l;
```

```
short int si;
```

```
unsigned int ui;
```

```
double balance, profit, loss;
```

Constants

Constants refer to fixed values that the program cannot alter. Constants can be of any of the basic data types. The way each constant is represented depends upon its type. Constants are also called literals. We can use keyword `const` prefix to declare constants with a specific type as follows:

```
const type variableName = value;
```

e.g,

```
const int LENGTH = 10;
```

Enumerated Types An enumerated type declares an optional type name and a set of zero or more identifiers that can be used as values of the type. Each enumerator is a constant whose type is the enumeration. Creating an enumeration requires the use of the keyword `enum`. The general form of an enumeration type is:

```
enum enum-name { list of names } var-list;
```

Here, the `enum-name` is the enumeration's type name. The list of names is comma separated.

For example, the following code defines an enumeration of colors called `color` and the variable `c` of type `color`. Finally, `c` is assigned the value "blue".

```
enum color { red, green, blue } =c;
```

By default, the value of the first name is 0, the second name has the value 1 and the third has the value 2, and so on. But you can give a name, a specific value by adding an initializer. For example, in the following enumeration, `green` will have the value 5.

```
enum color { red, green=5, blue };
```

Here, `blue` will have a value of 6 because each name will be one greater than the one that precedes it.

Operator

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators. Generally, there are six type of operators: Arithmetical operators, Relational operators, Logical operators, Assignment operators, Conditional operators, Comma operator.

Arithmetical operators

Arithmetical operators +, -, *, /, and % are used to performs an arithmetic (numeric) operation.

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus

You can use the operators +, -, *, and / with both integral and floating-point data types. Modulus or remainder % operator is used only with the integral data type.

Relational operators

The relational operators are used to test the relation between two values. All relational operators are binary operators and therefore require two operands. A relational expression returns zero when the relation is false and a non-zero when it is true. The following table shows the relational operators.

Relational	Operators Meaning
<	Less than
<=	Less than or equal to
==	Equal to
>	Greater than
>=	Greater than or equal to
!=	Not equal to

Logical operators

The logical operators are used to combine one or more relational expression. The logical operators are

Operators	Meaning
	OR
&&	AND

! **NOT**

Assignment operator

The assignment operator '=' is used for assigning a variable to a value. This operator takes the expression on its right-hand-side and places it into the variable on its left-hand-side. For example:

```
m = 5;
```

The operator takes the expression on the right, 5, and stores it in the variable on the left, m.

```
x = y = z = 32;
```

This code stores the value 32 in each of the three variables x, y, and z. In addition to standard assignment operator shown above, C++ also support compound assignment operators.

Compound Assignment Operators

Operator	Example	Equivalent to
+=	A += 2	A = A + 2
-=	A -= 2	A = A - 2
%=	A %= 2	A = A % 2
/=	A /= 2	A = A / 2
*=	A *= 2	A = A * 2

Increment and Decrement Operators

C++ provides two special operators viz '++' and '--' for incrementing and decrementing the value of a variable by 1. The increment/decrement operator can be used with any type of variable but it cannot be used with any constant. Increment and decrement operators each have two forms, pre and post.

The syntax of the increment operator is: Pre-increment: ++variable Post-increment: variable++ The syntax of the decrement operator is:

Pre-decrement: --variable Post-decrement: variable--

In Prefix form first variable is first incremented/decremented, then evaluated In Postfix form first variable is first evaluated, then incremented / decremented.

Conditional operator

The conditional operator ?: is called ternary operator as it requires three operands. The format of the conditional operator is :

```
Conditional_ expression ? expression1 : expression2;
```

If the value of conditional expression is true then the expression1 is evaluated, otherwise expression2 is evaluated.

```
int a = 5, b = 6;
```

```
big = (a > b) ? a : b;
```

The condition evaluates to false, therefore big gets the value from b and it becomes 6.

The comma operator

The comma operator gives left to right evaluation of expressions. When the set of expressions has to be evaluated for a value, only the rightmost expression is considered.

```
int a = 1, b = 2, c = 3, i; // comma acts as separator, not as an operator
```

`i = (a, b);` // stores b into i would first assign the value of a to i, and then assign value of b to variable i. So, at the end, variable i would contain the value 2.

The sizeof operator

The sizeof operator can be used to find how many bytes are required for an object to store in memory. For example

```
sizeof (char) returns 1
```

```
sizeof (float) returns 4
```

Typecasting:

Typecasting is the concept of converting the value of one type into another type. For example, you might have a float that you need to use in a function that requires an integer.

Implicit conversion:

Almost every compiler makes use of what is called automatic typecasting. It automatically converts one type into another type. If the compiler converts a type it will normally give a warning. For

example this warning: conversion from 'double' to 'int', possible loss of data. The problem with this is, that you get a warning (normally you want to compile without warnings and errors) and you are not in control. With control we mean, you did not decide to convert to another type, the compiler did. Also the possible loss of data could be unwanted.

Explicit conversion:

The C++ language have ways to give you back control. This can be done with what is called an explicit conversion.

Four typecast operators

The C++ language has four typecast operators:

- static_cast
- reinterpret_cast
- const_cast
- dynamic_cast

Type Conversion

The Type Conversion is that which automatically converts the one data type into another but remember we can store a large data type into the other. For example we can't store a float into int because a float is greater than int.

When a user can convert the one data type into another, then it is called as the **type casting**. The type Conversion is performed by the compiler but a casting is done by the user for example converting a float into int. When we use the Type Conversion then it is called the promotion. In the type casting when we convert a large data type into another then it is called as the demotion. When we use the type casting then we can lose some data.

Control Structures

Control structures allow to control the flow of program's execution based on certain conditions. C++ supports following basic control structures:

1) Selection Control structure

2) Loop Control structure

1) Selection Control structure: Selection Control structures allow to control the flow of program's execution depending upon the state of a particular condition being true or false. C++ supports two types of selection statements: if and switch. Condition operator (?:) can also be used as

an alternative to the if statement.

A.1) If Statement:

The syntax of an if statement in C++ is:

```
if(condition)

{

// statement(s) will execute if the condition is true

}
```

If the condition evaluates to true, then the block of code inside the if statement will be executed. If it evaluates to false, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

A.2) The if...else Statement

The syntax is shown as:

```
if(condition)

{

// statement(s) will execute if the condition is true

}

else

{

// statement(s) will execute if condition is false

}
```

If the condition evaluates to true, then the if block of code will be executed, otherwise else block of code will be executed.

A.3) if...else if...else Statement

An if statement can be followed by an optional else if...else statement, which is very useful to test various conditions using single if...else if statement.

The Syntax is shown as:

```

if(condition 1)
{
// Executes when the condition 1 is true
} else if(condition 2)
{
// Executes when the condition 2 is true
}
else if(condition 3)
{
// Executes when the condition 3 is true
}
else
{
// executes when the none of the above condition is true.
}

```

A.4) Nested if Statement

It is always legal to nest if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

The syntax for a nested if statement is as follows:

```

if( condition 1)
{
// Executes when the condition 1 is true
if(condition 2)

```

```

    {

        // Executes when the condition 2 is true

    }

}

```

B) Switch

C++ has a built-in multiple-branch selection statement, called switch, which successively tests the value of an expression against a list of integer or character constants. When a match is found, the statements associated with that constant are executed. The general form of the switch statement is:

```

switch (expression)

{

case constant1:

statement sequence

break;

case constant2:

statement sequence

break;

case constant3:

statement sequence

break;

    default

statement sequence

}

```

The expression must evaluate to a character or integer value. Floating-point expressions, for example, are not allowed. The value of expression is tested, in order, against the values of the constants specified in the case statements. When a match is found, the statement sequence

associated with that case is executed until the break statement or the end of the switch statement is reached. The default statement is executed if no matches are found. The default is optional and, if it is not present, no action takes place if all matches fail.

The break statement is one of C++'s jump statements. You can use it in loops as well as in the switch statement. When break is encountered in a switch, program execution "jumps" to the line of code following the switch statement.

2) Loop control structures A loop statement allows us to execute a statement or group of statements multiple times. Loops or iterative statements tell the program to repeat a fragment of code several times or as long as a certain condition holds. C++ provides three convenient iterative statements: while, for, and do-while.

while loop A while loop statement repeatedly executes a target statement as long as a given condition is true. It is an entry-controlled loop.

The syntax of a while loop in C++ is:

```
while(condition)

{

statement(s);

}
```

Here, statement(s) may be a single statement or a block of statements. The condition may be any expression, and true is any non-zero value. The loop iterates while the condition is true. After each execution of the loop, the value of test expression is changed. When the condition becomes false, program control passes to the line immediately following the loop.

The do-while Loop The do-while loop differs from the while loop in that the condition is tested after the body of the loop. This assures that the program goes through the iteration at least once. It is an exit-controlled loop.

```
do

{

statement(s);

}

while( condition );
```

The conditional expression appears at the end of the loop, so the statement(s) in the loop execute

once before the condition is tested. If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

for Loop: A for loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times. The syntax of a for loop in C++ is:

```
for ( init; condition; increment )  
  
{  
  
statement(s);  
  
}
```

Here is the flow of control in a for loop:

1. The init step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
2. Next, the condition is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
3. After the body of the for loop executes, the flow of control jumps back up to the increment statement. This statement allows you to update any C++ loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
4. The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

Structure of A Simple C++ Program

```
#include<iostream.h>  
  
#include<conio.h>  
  
int main()  
  
{  
  
cout<< "Simple C++ program without using class";  
  
return 0;
```

}

Lines beginning with a hash sign (#) are directives read and interpreted by what is known as the preprocessor. They are special lines interpreted before the compilation of the program itself begins. In this case, the directive `#include <iostream.h>`, instructs the preprocessor to include a section of standard C++ code, known as header `iostream` that allows to perform standard input and output operations, such as writing the output of this program to the screen.

The function named `main` is a special function in all C++ programs; it is the function called when the program is run. The execution of all C++ programs begins with the `main` function, regardless of where the function is actually located within the code.

The open brace (`{`) indicates the beginning of `main`'s function definition, and the closing brace (`}`) indicates its end.

The statement :

```
cout<< "Simple C++ program without using class";
```

causes the string in quotation marks to be displayed on the screen. The identifier `cout` (pronounced as c out) denotes an object. It points to the standard output device namely the console monitor. The operator `<<` is called insertion operator. It directs the string on its right to the object on its left.

The program ends with this statement:

```
return 0;
```

This causes zero to be returned to the calling process (which is usually the operating system). Returning zero indicates that the program terminated normally. Abnormal program termination should be signaled by returning a nonzero value.

The general structure of C++ program with classes is shown as :

1. Documentation Section
2. Preprocessor Directives or Compiler Directives Section
 - (i) Link Section
 - (ii) Definition Section
3. Global Declaration Section
4. Class declaration or definition
5. Main C++ program function called `main ()`

UNIT-II: ARRAYS AND FUNCTIONS IN C++:

ARRAYS

Introduction:

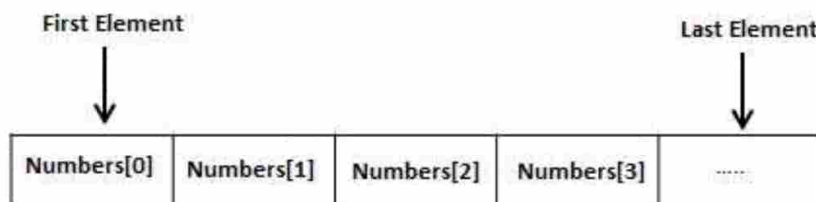
An **array** is a collection of data items, all of the same type, accessed using a common name. A one-dimensional **array** is like a list; A two dimensional **array** is like a table

An **array** is a collection of one or more values of the same **type**. Each value is called an element of the **array**. The elements of the **array** share the same variable name but each element has **its** own unique index number (also known as a subscript).

An **array** can be of any **type**, For example: int , float , char etc. An array is a variable that can store multiple values. For example, if we want to store 100 integers, we can create an array for it.

```
int data[100]
```

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays:

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows –

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type.

For example, to declare a 10-element array called **balance** of type double, use this statement –

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays:

We can initialize an array in C either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces { } cannot be larger than the number of elements that we declare for the array between square brackets [].

If we omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if we write –

```
double balance[ ] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

We will create exactly the same array as we did in the previous example. Following is an example to assign a single element of the array –

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0.

All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1.

Shown below is the pictorial representation of the array we discussed above –

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

```
#include <stdio.h>

int main ()
{
    int n[ 10 ]; /* n is an array of 10 integers */
    int i,j;

    /* initialize elements of array n to 0 */
    for ( i = 0; i < 10; i++ )
    {
        n[ i ] = i + 100; /* set element at location i to i + 100 */
    }
}
```

```

/* output each array element's value */
for (j = 0; j < 10; j++ )
{
    printf("Element[%d] = %d\n", j, n[j] );
}

return 0;

```

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

```

Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109

```

Two Dimensional Arrays

C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.

An array of arrays is known as 2D array. The two dimensional (2D) array in C programming is also known as matrix. A matrix can be represented as a table of rows and columns.

This program demonstrates how to store the elements entered by user in a 2d array and how to display the elements of a two dimensional array.

```

#include<stdio.h>
int main()
{
    /* 2D array declaration*/
    int disp[2][3];
    /*Counter variables for the loop*/
    int i, j;
    for(i=0; i<2; i++)
    {
        for(j=0;j<3;j++)
        {
            printf("Enter value for disp[%d][%d]:", i, j);

```

```

        scanf("%d", &disp[i][j]);
    }
}
//Displaying array elements
printf("Two Dimensional array elements:\n");
for(i=0; i<2; i++)
{
    for(j=0;j<3;j++)
    {
        printf("%d ", disp[i][j]);
        if(j==2)
        {
            printf("\n");
        }
    }
}
return 0;
}

```

Output:

```

Enter value for disp[0][0]:1
Enter value for disp[0][1]:2
Enter value for disp[0][2]:3
Enter value for disp[1][0]:4
Enter value for disp[1][1]:5
Enter value for disp[1][2]:6
Two Dimensional array elements:
1 2 3
4 5 6

```

Initialization of 2D Array

There are two ways to initialize a two Dimensional arrays during declaration.

```

int disp[2][4] =
{
    { 10, 11, 12, 13},
    { 14, 15, 16, 17}
};

```


OR

```
int disp[2][4] = { 10, 11, 12, 13, 14, 15, 16, 17};
```

Although both the above declarations are valid, better to use the first method as it is more readable, because we can visualize the rows and columns of 2D array in this method.

Things that you must consider while initializing a 2D array:

We already know, when we initialize a normal array (or one dimensional array) during declaration, we need not to specify the size of it. However that's not the case with 2D array, you must always specify the second dimension even if we are specifying elements during the declaration. Let's understand this with the help of few examples –

```
/* Valid declaration*/
int abc[2][2] = { 1, 2, 3 ,4 }
/* Valid declaration*/
int abc[ ][2] = { 1, 2, 3 ,4 }
/* Invalid declaration – you must specify second dimension*/
int abc[ ][ ] = { 1, 2, 3 ,4 }
/* Invalid because of the same reason mentioned above*/
int abc[2][ ] = { 1, 2, 3 ,4 }
```

Arrays of Strings

Strings are actually one-dimensional array of characters terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows –

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C/C++ –

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string –

```
#include <stdio.h>

int main ()
{
    char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
    printf("Greeting message: %s\n", greeting );
    return 0;
}
```

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

Greeting message: Hello

Functions

Introduction:

A function groups a number of program statements into a unit and gives it a name. This unit can then be invoked (called) from other parts of the program. The function's code is stored in only one place in memory, even though the function is executed many times in the course of the program's execution. Functions help to reduce the program size when same set of instructions are to be executed again and again.

Thus, a function is a block of statements that performs a specific task. If we are writing a C++ program and we need to perform a same task in that program more than once. In such case we have two options:

- a) Use the same set of statements every time we want to perform the task
- b) Create a function to perform that task, and just call it every time we need to perform that task. Using option (b) is a good practice and a good programmer always uses functions while writing code in C++.

Why we need functions in C++:

Functions are used because of following reasons –

- a) To improve the readability of code.
- b) Improves the reusability of the code, same function can be used in any program rather than writing the same code from scratch.
- c) Debugging of the code would be easier if you use functions, as errors are easy to be traced.
- d) Reduces the size of the code, duplicate set of statements are replaced by function calls.

Types of functions:

1) Predefined standard library functions

Standard library functions are also known as built-in functions. Functions such as puts(), gets(), printf(), scanf() etc are standard library functions. These functions are already defined in header files (files with .h extensions are called header files such as stdio.h), so we just call them whenever there is a need to use them.

For example, printf() function is defined in <stdio.h> header file so in order to use the printf() function, we need to include the <stdio.h> header file in our program using #include <stdio.h>.

2) User Defined functions

The functions that we create in a program are known as user defined functions or in other words you can say that a function created by user is known as user defined function.

User Defined functions:

Now we will learn how to create user defined functions and how to use them in C++ Programming.

A general function consists of three parts, namely, function declaration (or prototype), function definition and function call.

Function declaration — prototype:

A function has to be declared before using it, in a manner similar to variables and constants. A function declaration tells the compiler about a function's name, return type, and parameters and how to call the function.

The general form of a C++ function declaration is as follows:

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

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

Function definition:

The function definition is the actual body of the function. The function definition consists of two parts namely, function header and function body.

The general form of a C++ function definition is as follows:

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

{ body of the function }
```

Here

return-type: A function may return a value. The return-type is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the return-type is the keyword void.

Function Name:

This is the actual name of the function.

Argument or Parameters:

A parameter is like a placeholder. When a function is invoked, we pass a value

to the parameter. This value is referred to as **actual parameter or argument**. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.

Function Body:

The function body contains a collection of statements that define what the function does.

Calling a Function :

To use a function, we will have to call or invoke that function. To call a function, we simply need to pass the required parameters along with function name, and if function returns a value, then we can store returned value.

Type of User-defined Functions in C++:

There can be 4 different types of user-defined functions, they are:

1. Function with no arguments and no return value
2. Function with no arguments and a return value
3. Function with arguments and no return value
4. Function with arguments and a return value

Below, we will discuss about all these types, along with program examples.

1.Function with no arguments and no return value:

Such functions can be used to display information or they are completely dependent on user inputs. Below is an example of a function, which takes 2 numbers as input from user, and display which is the greater number.

```
#include<stdio.h>
void greatNum();    // function declaration
int main()
{
    greatNum();    // function call
    return 0;
}
void greatNum()    // function definition
{
    int i, j;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    if(i > j)
    {
        printf("The greater number is: %d", i);
    }
}
```

```

    }
    else
    {
        printf("The greater number is: %d", j);
    }
}

```

2.Function with no arguments and a return value:

We have modified the above example to make the function greatNum() return the number which is greater amongst the 2 input numbers.

```

#include<stdio.h>
int greatNum();    // function declaration
int main()
{
    int result;
    result = greatNum();    // function call
    printf("The greater number is: %d", result);
    return 0;
}
int greatNum()    // function definition
{
    int i, j, greaterNum;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    if(i > j)
    {
        greaterNum = i;
    }
    else
    {
        greaterNum = j;
    }
    // returning the result
    return greaterNum;
}

```

3.Function with arguments and no return value:

We are using the same function as example again and again, to demonstrate that to solve a problem there can be many different ways.

This time, we have modified the above example to make the function greatNum() take two int values as arguments, but it will not be returning anything.

```
#include<stdio.h>
void greatNum(int a, int b);    // function declaration
int main()
{
    int i, j;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    greatNum(i, j);    // function call
    return 0;
}

void greatNum(int x, int y)    // function definition
{
    if(x > y)
    {
        printf("The greater number is: %d", x);
    }
    else {
        printf("The greater number is: %d", y);
    }
}
```

4.Function with arguments and a return value:

This is the best type, as this makes the function completely independent of inputs and outputs, and only the logic is defined inside the function body.

```
#include<stdio.h>
int greatNum(int a, int b);    // function declaration
int main()
{
    int i, j, result;
    printf("Enter 2 numbers that you want to compare...");
    scanf("%d%d", &i, &j);
    result = greatNum(i, j);    // function call
    printf("The greater number is: %d", result);
    return 0;
}
```



```

int greatNum(int x, int y)    // function definition
{
    if(x > y)
    {
        return x;
    }
    else {
        return y;
    }
}

```

1) Function – Call by value method – In the call by value method the actual arguments are copied to the formal arguments, hence any operation performed by function on arguments doesn't affect actual parameters.

2) Function – Call by reference method – Unlike call by value, in this method, address of actual arguments (or parameters) is passed to the formal parameters, which means any operation performed on formal parameters affects the value of actual parameters.

Function call by value is the default way of calling a function in C programming.

Actual parameters: The parameters that appear in function calls.

Formal parameters: The parameters that appear in function declarations.

Inline Functions

An inline function is a function that is expanded inline at the point at which it is invoked, instead of actually being called. The reason that inline functions are an important addition to C++ is that they allow us to create very efficient code. Each time a normal function is called, a significant amount of overhead is generated by the calling and return mechanism.

A function can be defined as an inline function by prefixing the keyword inline to the function header. If a function cannot be inlined, it will simply be called as a normal function.

One of the advantages of using function is to save memory space by making common block for the code we need to execute many times. When compiler invoke / call a function, it takes extra time to execute such as jumping to the function definition, saving registers, passing value to argument and returning value to calling function. This extra time can be avoidable for large functions but for small functions we use inline function to save extra time.

Example:

```
#include<iostream.h>
inline int Add(int x,int y)
{
    return x+y;
}
void main()
{
    cout<<"\n\tThe Sum is : " << Add(10,20);
    cout<<"\n\tThe Sum is : " << Add(45,83);
    cout<<"\n\tThe Sum is : " << Add(27,48);
}
```

Output :

```
The Sum is : 30
The Sum is : 98
The Sum is : 75
```

Reference variable

A reference variable is an **alias**, that is, *another name* for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable. To declare a reference variable or parameter, precede the variable's name with the &.

The syntax for declaring a reference variable is:

```
datatype &Ref = variable name;
```

Example:

```
int main()
{
    int var1=10;                                //declaring simple variable
    int &var2=var1;                              //declaring reference variable
    cout<<"\n value of var2 =" << var2;
    return 0;
}
```

var2 is a reference variable to var1. Hence, var2 is an alternate name to var1. This code prints the value of var2 exactly as that of var1.

Call by reference

Arguments can be passed to functions in one of two ways: using call-by-value or call-by-reference. When using call-by-value, a copy of the argument is passed to the function. Call-by-reference passes the address of the argument to the function. By default, C++ uses call-by-value.

Provision of the reference variables in c++ permits us to pass parameter 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.

Example

```
#include
<iostream.h>
#include<conio.h>
void swap(int &x, int &y);           // function declaration
int main ()
{
    int a=10, b=20;
    cout << "Before swapping"<<endl;
    cout<< "value of a : " << a << " value of b : " << b << endl;
    swap(a, b);                     //calling a function to swap the values.
    cout << "After swapping"<<endl;
    cout<< "value of a : " << a << "value of b : " << b << endl;
    return 0;
}
void swap(int &x, int &y)           //function definition to swap the values.
{
    int temp;
    temp = x;
    x = y;
    y = temp;
}
```

Output:

Before swapping value of a:10 value of b:20
After swapping value of a:20 value of b:10

Function Overloading

Function overloading is the process of using the same name for two or more functions. Each redefinition of the function must use either different types of parameters or a different number of parameters. It is only through these differences that the compiler knows which function to call in any given situation.

Overloaded functions can help to reduce the complexity of a program by allowing related operations to be referred to by the same name. To overload a function, simply declare and define all required versions. The compiler will automatically select the correct version based upon the number and/or type of the arguments used to call the function. Two functions differing only in their return types cannot be overloaded.

Example

```
#include<iostream.h>
#include<conio.h>
int sum(int p,int q,int r);
double sum(int l,double m);
float sum(float p,float q)
int main()
{
cout<<"sum="<< sum(11,22,33);           //calls func1
cout<<"sum="<< sum(10,15.5);           //calls func2
cout<<"sum="<< sum(13.5,12.5);         //calls func3
return 0;
}
int sum(int p,int q,int r)              //func1
{
    return(a+b+c)
;
}
double sum(int l,double m)              //func2
{
    return(l+m);
}
float sum(float p,float q)              //func3
{
    return(p+q);
}
```

Default arguments

C++ allows a function to assign a parameter as default value when no argument

corresponding to that parameter is specified in a call to that function. The default value is specified in a manner syntactically similar to a variable initialization. All default parameters must be to the right of any parameters that don't have defaults. We cannot provide a default value to a particular argument in the middle of an argument list. When we create a function that has one or more default arguments, those arguments must be specified only once: either in the function's prototype or in the function's definition if the definition precedes the function's first use.

Default arguments are useful if we don't want to go to the trouble of writing arguments that, for example, almost always have the same value. They are also useful in cases where, after a program is written, the programmer decides to increase the capability of a function by adding another argument. Using default arguments means that the existing function calls can continue to use the old number of arguments, while new function calls can use more.

Example

```
#include
<iostream.h>
#include<conio.h>
int sum(int a, int b=20)
{
    return( a + b);
}
int main ()
{
    int a = 100, b=200, result;
    result = sum(a, b);                //here a=100 , b=200
    cout << "Total value is :" << result << endl;
    result = sum(a);                  //here a=100 , b=20(using default value)
    cout << "Total value is :" << result << endl;
    return 0;
}
```

4.11 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 in Table 4.1.

Table 4.1 Commonly used math library functions

Function	Purposes
ceil(x)	Rounds x to the smallest integer not less than x $\text{ceil}(8.1) = 9.0$ and $\text{ceil}(-8.8) = -8.0$
cos(x)	Trigonometric cosine of x (x in radians)
exp(x)	Exponential function e^x .
fabs(x)	Absolute value of x . If $x > 0$ then $\text{abs}(x)$ is x If $x = 0$ then $\text{abs}(x)$ is 0.0 If $x < 0$ then $\text{abs}(x)$ is $-x$
floor(x)	Rounds x to the largest integer not greater than x $\text{floor}(8.2) = 8.0$ and $\text{floor}(-8.8) = -9.0$
log(x)	Natural logarithm of x (base e)
log10(x)	Logarithm of x (base 10)
pow(x,y)	x raised to power y (x^y)
sin(x)	Trigonometric sine of x (x in radians)
sqrt(x)	Square root of x
tan(x)	Trigonometric tangent of x (x in radians)

note

The argument variables x and y are of type **double** and all the 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++.

Classes and Objects

5

Key Concepts

Using structures | Creating a class | Defining member functions | Creating objects | Using objects | Inline member functions | Nested member functions | Private member functions | Arrays as class members | Storage of objects | Static data members | Static member functions | Using arrays of objects | Passing objects as parameters | Making functions friendly to classes | Functions returning objects | **const** member functions | Pointers to members | Using dereferencing operators | Local classes

5.1

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. We shall discuss, in this chapter, the concept of class by first reviewing the traditional structures found in C and then the ways in which classes can be designed, implemented and applied.

5.2

C Structures Revisited

We know that one of the unique features of the C language is structures. They provide a method for 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* or *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
```


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

5.3

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 declaration describes the type and scope of its members. The class function definitions 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 declaration;
};
```

The **class** declaration is similar to a **struct** declaration. 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. This is illustrated in Fig. 5.1. The binding of data and functions together into a single class-type variable is referred to as *encapsulation*.

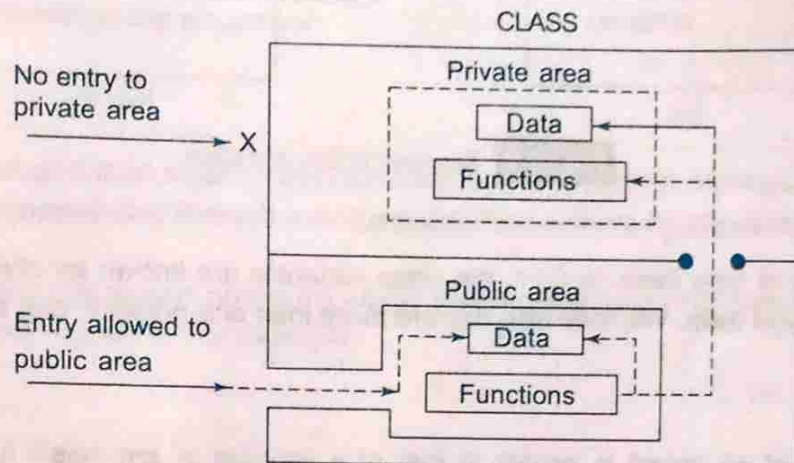


Fig. 5.1 Data hiding in classes

A Simple Class Example

A typical class declaration would look like:

```
class item
{
    int number;                // variables 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 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**. Figure 5.2 shows two different notations used by the OOP analysts to represent a class.

Creating Objects

Remember that the declaration 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,

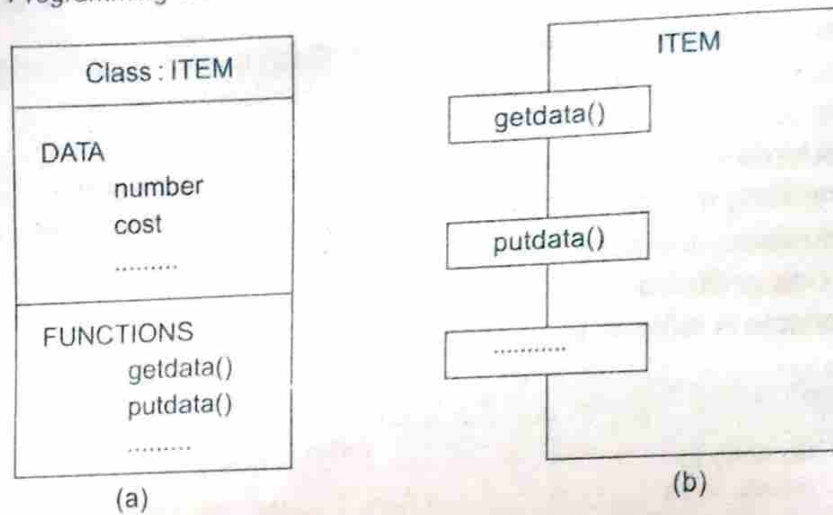


Fig. 5.2 Representation of a class

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

creates 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 closing brace, 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. Please refer Sec. 5.4 for further details.

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();
```

sends 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 z;
};

.....
.....

xyz p;
p.x = 0;           // error, x is private
p.z = 10           // OK, z is public
.....
.....
```



NOTE: The use of data in this manner defeats the very idea of data hiding and therefore should be avoided.

5.4

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. This 'label' tells the compiler 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. They 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 functions 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 functions can access the private data of the class. A nonmember function cannot do so. (However, an exception to this rule is a *friend* function discussed later.)
- 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 << "\n";
        cout << 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.

5.5

A C++ Program with Class

All the details discussed so far are implemented in Program 5.1.

Program 5.1 Class Implementation

```
#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
```

(Contd.)

```

{
    number = a; // private variables
    cost = b;   // directly used
}
//.....Main Program.....

int main()
{
    item x; // create object x

    cout << "\nobject x " << "\n";

    x.getdata(100, 299.95); // call member function
    x.putdata();           // call member function

    item y; // create another object

    cout << "\nobject y" << "\n";

    y.getdata(200, 175.50);
    y.putdata();

    return 0;
}

```

This program features the class **item**. This class contains 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 statements 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 Program 5.1:

```

object x
number  :100
cost    :299.95

object y
number  :200
cost    :175.5

```


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

5.6 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 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;
}
```

5.7 Nesting of Member Functions

We just discussed that 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*. Program 5.2 illustrates this feature.

Program 5.2 Nesting of Member Functions

```
#include<iostream>
#include<conio.h>
#include<string>
using namespace std;

class binary
{
    string s;

    public:
    void read(void)
    {
        cout<<"Enter a binary number:";
        cin>>s;
    }
}
```

(Contd.)


```

void chk_bin(void)
{
    for(int i=0;i<s.length();i++)
    {
        if(s.at(i)!='0' && s.at(i)!='1')
        {
            cout<<"\nIncorrect binary number format...the program will
            quit";
            getch();
            exit(0);
        }
    }
}

void ones(void)
{
    chk_bin(); //calling member function
    for(int i=0;i<s.length();i++)
    {
        if(s.at(i)=='0')
            s.at(i)='1';
        else
            s.at(i)='0';
    }
}

void displayones(void)
{
    ones(); //calling member function
    cout<<"\nThe 1's compliment of the above binary number is:
    "<<s;
}

};

int main()
{
    binary b;
    b.read();
    b.displayones();
    getch();
    return 0;
}

```

The output of Program 5.2 would be:

Output 1:

```

Enter a binary number: 110101
The 1's compliment of the above binary number is: 001010

```

Output 2:

```
Enter a binary number: 1101210
Incorrect binary number format...the program will quit
```

NOTE: The above program uses the built-in class, `string` for storing the binary number. We will explore the `string` class in more detail in Chapter 15.

5.8

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 m;
    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
}
```

5.9

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.

Let us consider a shopping list of items for which we place an order with a dealer every month. The list includes details such as the code number and price of each item. We would like to perform operations such as adding an item to the list, deleting an item from the list and printing the total value of the order. Program 5.3 shows how these operations are implemented using a class with arrays as data members.

Program 5.3 Processing Shopping List

```
#include <iostream>

using namespace std;

const m=50;

class ITEMS
{
    int itemCode[m];
    float itemPrice[m];
    int count;

public:
    void CNT(void){count = 0;} // initializes count to 0
    void getitem(void);
    void displaySum(void);
    void remove(void);
    void displayItems(void);
};

//=====
void ITEMS :: getitem(void)                // assign values to data
                                           // members of item
{
    cout << "Enter item code :";
    cin >> itemCode[count];
    cout << "Enter item cost :";
    cin >> itemPrice[count];
    count++;
}

void ITEMS :: displaySum(void)             // display total value of
                                           // all items
{
    float sum = 0;
    for(int i=0; i<count; i++)
        sum = sum + itemPrice[i];
    cout << "\nTotal value : " << sum << "\n";
}
```

(Contd.)


```

}
void ITEMS :: remove(void) // delete a specified item
{
    int a;
    cout << "Enter item code :";
    cin >> a;

    for(int i=0; i<count; i++)
        if(itemCode[i] == a)
            itemPrice[i] = 0;
}

void ITEMS :: displayItems(void) // displaying items
{
    cout << "\nCode Price\n";

    for(int i=0; i<count; i++)
    {
        cout << "\n" << itemCode[i];
        cout << " " << itemPrice[i];
    }
    cout << "\n";
}

//=====

int main()
{
    ITEMS order;
    order.CNT();
    int x;
    do // do....while loop
    {
        cout << "\nYou can do the following;"
            << "Enter appropriate number \n";
        cout << "\n1 : Add an item ";
        cout << "\n2 : Display total value";
        cout << "\n3 : Delete an item";
        cout << "\n4 : Display all items";
        cout << "\n5 : Quit";
        cout << "\n\nWhat is your option?";

        cin >> x;

        switch(x)
        {
            case 1 : order.getItem(); break;
            case 2 : order.displaySum(); break;
            case 3 : order.remove(); break;
            case 4 : order.displayItems(); break;
            case 5 : break;
            default : cout << "Error in input; try again\n";
        }
    } while(x != 5); // do...while ends

    return 0;
}

```

The output of Program 5.3 would be:

```
You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit
```

```
What is your option?1
Enter item code :111
Enter item cost :100
```

```
You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit
```

```
What is your option?1
Enter item code :222
Enter item cost :200
```

```
You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit
```

```
What is your option?1
Enter item code :333
Enter item cost :300
```

```
You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit
```

```
What is your option?2
Total value :600
```

```
You can do the following; Enter appropriate number
1 : Add an item
2 : Display total value
3 : Delete an item
4 : Display all items
5 : Quit
```

```
What is your option?3
Enter item code :222
```

You can do the following; Enter appropriate number

- 1 : Add an item
- 2 : Display total value
- 3 : Delete an item
- 4 : Display all items
- 5 : Quit

What is your option?4

Code	Price
111	100
222	0
333	300

You can do the following; Enter appropriate number

- 1 : Add an item
- 2 : Display total value
- 3 : Delete an item
- 4 : Display all items
- 5 : Quit

What is your option?5



NOTE: The program uses two arrays, namely `itemCode[]` to hold the code number of items and `itemPrice[]` to hold the prices. A third data member `count` is used to keep a record of items in the list. The program uses a total of four functions to implement the operations to be performed on the list. The statement

```
const int m = 50;
```

defines the size of the array members.

The first function `CNT()` simply sets the variable `count` to zero. The second function `getitem()` gets the item code and the item price interactively and assigns them to the array members `itemCode[count]` and `itemPrice[count]`. Note that inside this function `count` is incremented after the assignment operation is over. The function `displaySum()` first evaluates the total value of the order and then prints the value. The fourth function `remove()` deletes a given item from the list. It uses the item code to locate it in the list and sets the price to zero indicating that the item is not 'active' in the list. Lastly, the function `displayItems()` displays all the items in the list.

The program implements all the tasks using a menu-based user interface.

5.10

Memory Allocation for Objects

We have stated that the 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. This is shown in Fig. 5.3.

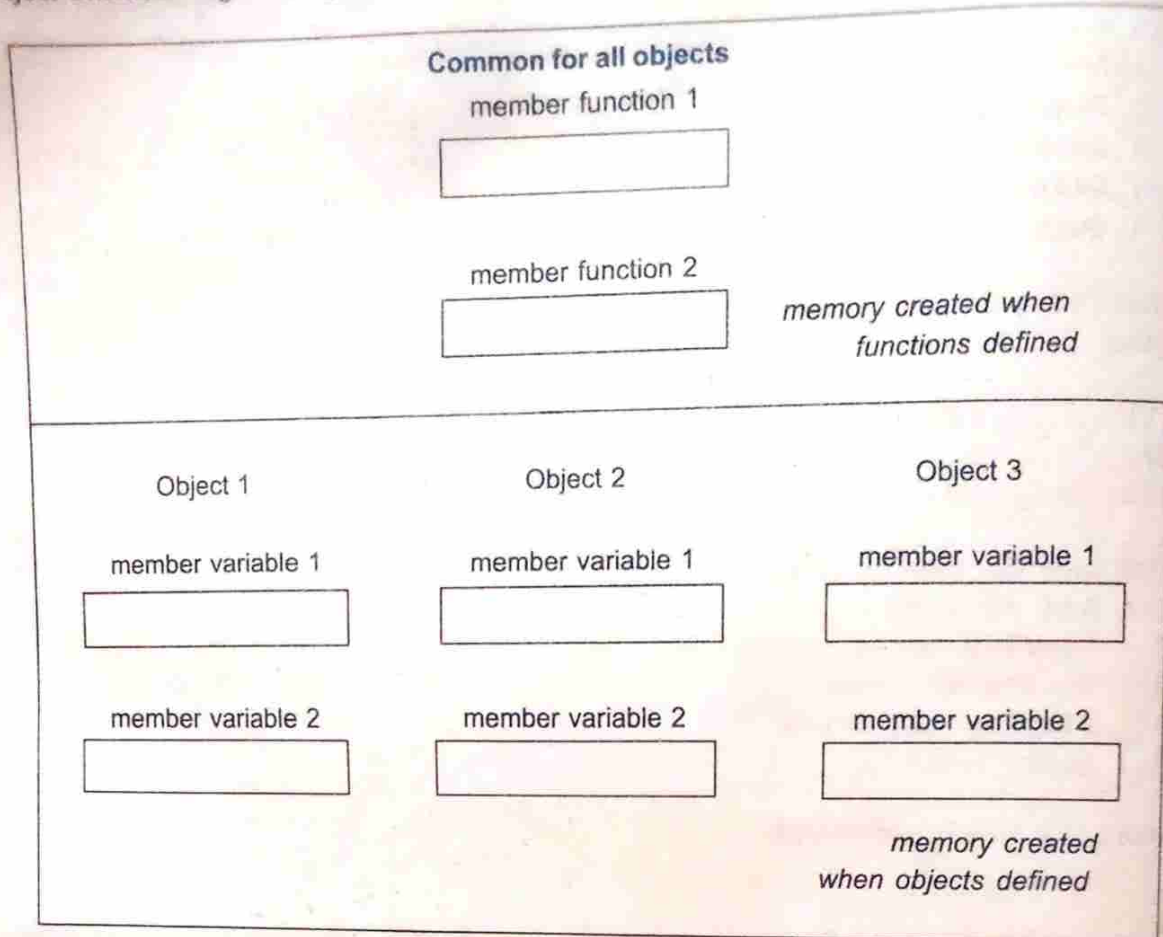


Fig. 5.3 Object of memory

5.11

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. Program 5.4 illustrates the use of a static data member.

Program 5.4 Static Class Member

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

(Contd.)

```

    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 data into object a
    b.getdata(200);         // getting data into object b
    c.getdata(300);         // getting data into object c

    cout << "After reading data" << "\n";

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

```

The output of the Program 5.4 would be:

```

count: 0
count: 0
count: 0
After reading data
count: 3
count: 3
count: 3

```

NOTE: 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. Figure 5.4 shows how a static variable is used by the objects.

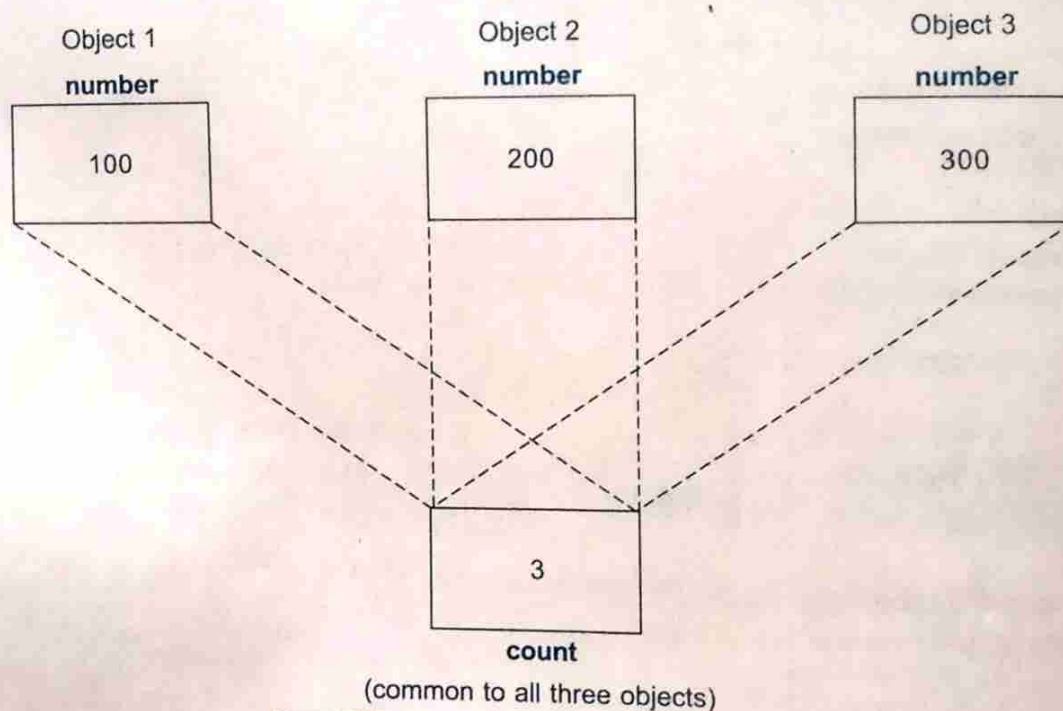


Fig. 5.4 Sharing of a static data member

Static variables are like non-inline member 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;
```

5.12

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;
```

Program 5.5 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.

Program 5.5 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 5.5 would be:

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

NOTE: 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.

Remember, the following function definition will not work:

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

5.13

Arrays 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 (foremen) 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 methods to access individual elements, and then the dot member operator to access the member functions. For example, the statement

```
manager[1].putdata();
```


will display the data of the i th element of the array **manager**. That is, this statement requests 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. The array **manager** is represented in Fig. 5.5. 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.

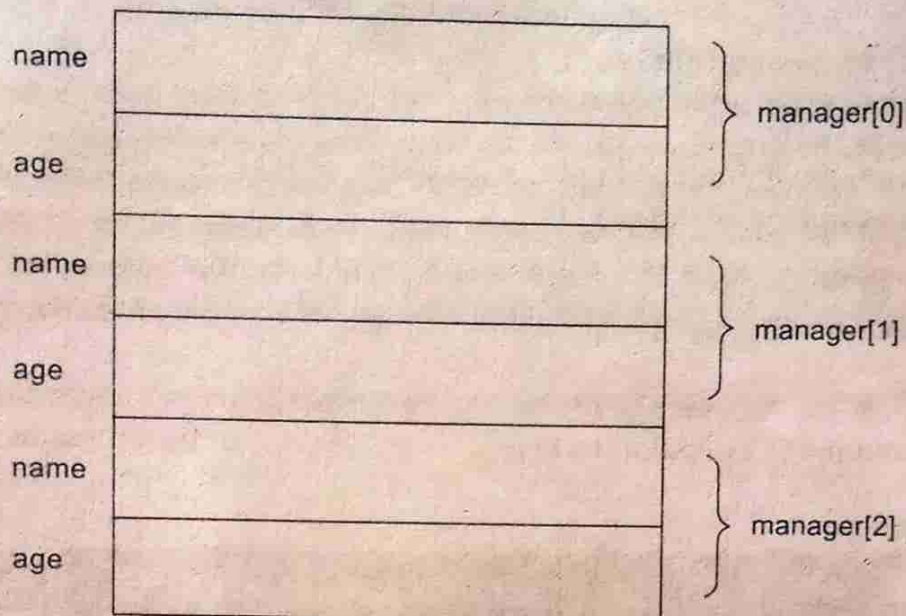


Fig. 5.5 Storage of data items of an object array

Program 5.6 illustrates the use of object arrays.

Program 5.6 Arrays of Objects

```
#include <iostream>

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;
}
```

(Contd.)

```

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 << "\nDetails of manager" << i+1 << "\n";
        manager[i].getdata();
    }
    cout << "\n";
    for(i=0; i<size; i++)
    {
        cout << "\nManager" << 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 manager1

Enter name: xxx

Enter age: 45

Details of manager2

Enter name: yyy

Enter age: 37

Details of manager3

Enter name: zzz

Enter age: 50

Program output

Manager1

Name: xxx

Age: 45

Manager2

Name: yyy

Age: 37

Manager3

Name: zzz

Age: 50

5.14

Objects as Function Arguments

Like any other data type, an object may be used as a 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.

Program 5.7 illustrates the use of objects as function arguments. It performs the addition of time in the hour and minutes format.

Program 5.7 Objects as Arguments

```
#include <iostream>

using namespace std;

class time
{
    int hours;
    int minutes;
public:
    void gettime(int h, int m)
    { hours = 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;
}
```

(Contd.)


```

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 Program 5.7 would be:

```

T1 = 2 hours and 45 minutes
T2 = 3 hours and 30 minutes
T3 = 6 hours and 15 minutes

```

NOTE: 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 members 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` refer to `T1`, and `T2.hours` and `T2.minutes` refer to `T2`.

Figure 5.6 illustrates how the members are accessed inside the function `sum()`.

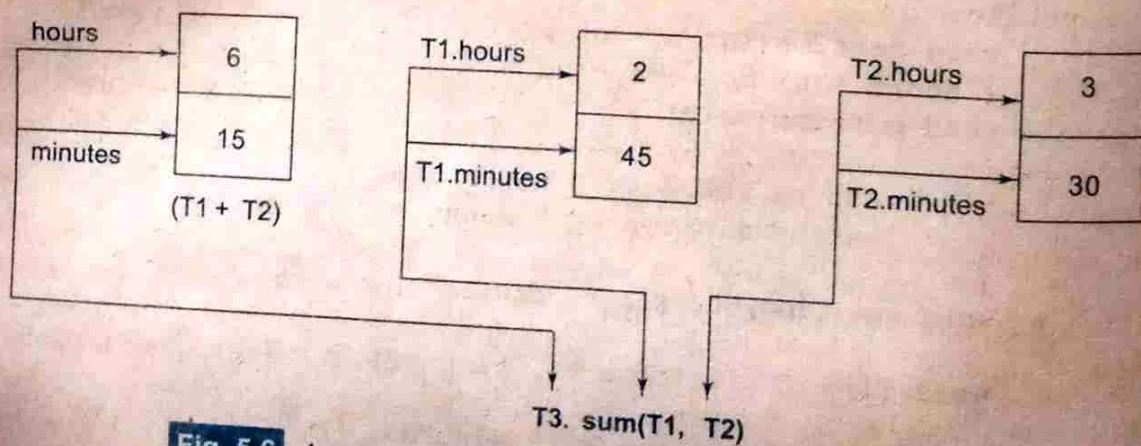


Fig. 5.6 Accessing members of objects within a called function

An object can also be passed as an argument to a nonmember 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.

5.15

Friendly Functions

We have been emphasizing throughout this chapter that the private members cannot be accessed from outside the class. That is, a nonmember 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, 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 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.

The friend functions are often used in operator overloading which will be discussed later.

Program 5.8 illustrates the use of a friend function.

Program 5.8 Friend Function

```
#include <iostream>

using namespace std;
```

(Contd.)


```

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 Program 5.8 would be:

Mean value = 32.5

 **NOTE:** The friend function accesses 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** functions 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();
    .....
};

// fun1() of X
// is 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 5.9 demonstrates how friend functions work as a bridge between the classes.

Program 5.9 A Function Friendly to 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) // Definition of friend
{
    if(m.x >= n.a)
        cout << m.x;
    else
        cout << n.a;
}
// ----- //
int main()
{
    ABC abc;
    abc.setvalue(10);
    XYZ xyz;
    xyz.setvalue(20);
    max(xyz, abc);

    return 0;
}
```

The output of Program 5.9 would be:

20

NOTE: 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. Remember, altering the values of private members is against the basic principles of data hiding. It should be used only when absolutely necessary.

Program 5.10 shows how to use a common friend function to exchange the private values of two classes. The function is called by reference.

Program 5.10 Swapping Private Data of Classes

```
#include <iostream>

using namespace std;

class class_2;

class class_1
{
    int value1;
public:
    void indata(int a) {value1 = a;}
    void display(void) {cout << value1 << "\n";}
    friend void exchange(class_1 &, class_2 &);
};

class class_2
{
    int value2;
public:
    void indata(int a) {value2 = a;}
    void display(void) {cout << value2 << "\n";}
    friend void exchange(class_1 &, class_2 &);
};

void exchange(class_1 & x, class_2 & y)
```

(Contd.)

```

    int temp = x.value1;
    x.value1 = y.value2;
    y.value2 = temp;
}

int main()
{
    class_1 C1;
    class_2 C2;

    C1.indata(100);
    C2.indata(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 **class_1** and **class_2**.

The output of Program 5.10 would be:

```

Values before exchange
100
200
Values after exchange
200
100

```

5.16

Returning Objects

A function cannot only receive objects as arguments but also can return them. The example in Program 5.11 illustrates how an object can be created (within a function) and returned to another function.

Program 5.11 Returning Objects

```

#include<iostream>

#include<conio.h>

using namespace std;

class matrix
{
    int m[3][3];

public:
    void read(void)
    {
        cout<<"Enter the elements of the 3X3 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 matrix trans(matrix);
};

matrix trans(matrix m1)
{
    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<<"\nYou entered the following matrix:";
    mat1.display();

    mat2 = trans(mat1);
    cout<<"\nTransposed matrix:";
    mat2.display();

    getch();
    return 0;
}

```

Upon execution, Program 5.11 would generate the following output:

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 matrix:

```

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 × 3 matrix and stores it in a new matrix object. The display member function displays the matrix elements.

5.17

const Member Functions

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.

5.18

Pointers 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".

Remember, 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;
cout << ap -> *ip;       // ap is pointer to object a
cout << ap -> m;         // display 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 which, 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 5.12 illustrates the use of dereferencing operators to access the class members.

Program 5.12 Dereferencing Operators

```
#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 Program 5.12 would be:

```
sum = 30
sum = 70
```

5.19

Local Classes

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

```
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.

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.

(Contd.)

- ☐ 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

Key Terms

abstract data type | arrays of objects | **class** | class declaration | class members | class variables | **const** member functions | data hiding | data members | dereferencing operator | dot operator | elements | encapsulation | **friend** functions | inheritance | **inline** functions | local class | member functions | nesting of member functions | objects | pass-by-reference | pass-by-value | period operator | **private** | prototype | **public** | scope operator | scope resolution | static data members | static member functions | static variables | **struct** | structure | structure members | structure name | structure tag | template

Review Questions

- 5.1 How do structures in C and C++ differ?
- 5.2 What is a class? How does it accomplish data hiding?
- 5.3 How does a C++ structure differ from a C++ class?
- 5.4 What are objects? How are they created?
- 5.5 How is a member function of a class defined?

- 5.6 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.
- 5.7 Describe the mechanism of accessing data members and member functions in the following cases:
- Inside the **main** program.
 - Inside a member function of the same class.
 - Inside a member function of another class.
- 5.8 When do we declare a member of a class **static**?
- 5.9 What is a friend function? What are the merits and demerits of using friend functions?
- 5.10 Can we pass class objects as function arguments? Explain with the help of an example.
- 5.11 State whether the following statements are TRUE or FALSE.
- Data items in a class must always be private.
 - A function designed as private is accessible only to member functions of that class.
 - A function designed as public can be accessed like any other ordinary functions.
 - Member functions defined inside a class specifier become inline functions by default.
 - Classes can bring together all aspects of an entity in one place.
 - Class members are public by default.
 - Friend functions have access to only public members of a class.
 - An entire class can be made a friend of another class.
 - Functions cannot return class objects.
 - Data members can be initialized inside class specifier.

Debugging Exercises

- 5.1 Identify the error in the following program.

```
#include <iostream.h>
struct Room
{
    int width;
    int length;
    void setValue(int w, int l)
    {
        width = w;
        length = l;
    }
};

void main()
{
    Room objRoom;
    objRoom.setValue(12, 1, 4);
}
```

- 5.2 Identify the error in the following program.

```
#include <iostream.h>
class Room
{
```

```

    int width, height;
    void setValue(int w, int h)
    {
        width = w;
        height = h;
    }
};
void main()
{
    Room objRoom;
    objRoom.width = 12;
}

```

5.3 Identify the error in the following program.

```

#include <iostream.h>
class Item
{
private:
    static int count;
public:
    Item()
    {
        count++;
    }
    int getCount()
    {
        return count;
    }
    int* getCountAddress()
    {
        return count;
    }
};
int Item::count = 0;

void main()
{
    Item objItem1;
    Item objItem2;

    cout << objItem1.getCount() << '\n';
    cout << objItem2.getCount() << '\n';

    cout << objItem1.getCountAddress() << '\n';
    cout << objItem2.getCountAddress() << '\n';
}

```

5.4 Identify the error in the following program.

```

#include <iostream.h>
class staticFunction
{

```

```

        static int count;
    public:
        static void setCount()
        {
            count++;
        }
        void displayCount()
        {
            cout << count;
        }
};
int staticFunction::count = 10;
void main()
{
    staticFunction obj1;
    obj1.setCount(5);
    staticFunction::setCount();
    obj1.displayCount();
}

```

5.5 Identify the error in the following program.

```

#include <iostream.h>
class Length
{
    int feet;
    float inches;
public:
    Length()
    {
        feet = 5;
        inches = 6.0;
    }
    Length(int f, float in)
    {
        feet = f;
        inches=in;
    }
    Length addLength(Length l)
    {
        l.inches += this->inches;
        l.feet += this->feet;
        if(l.inches>12)
        {
            l.inches-=12;
            l.feet++;
        }
        return l;
    }
    int getFeet()
    {
        return feet;
    }
}

```



```

float getInches()
{
    return inches;
}

};

void main()
{
    Length objLength1;
    Length objLength1(5, 6.5);
    objLength1 = objLength1.addLength(objLength2);
    cout << objLength1.getFeet() << ' ';
    cout << objLength1.getInches() << ' ';
}

```

5.6 Identify the error in the following program.

```

#include <iostream.h>
class Room;
void Area()
{
    int width, height;
    class Room
    {
        int width, height;
    public:
        void setValue(int w, int h)
        {
            width = w;
            height = h;
        }
        void displayValues()
        {
            cout << (float)width << ' ' << (float)height;
        }
    };
    Room objRoom1;
    objRoom1.setValue(12, 8);
    objRoom1.displayValues();
}

void main()
{
    Area();
    Room objRoom2;
}

```

Programming Exercises

5.1 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. **WEB**

- 5.2 Write a class to represent a vector (a series of float values). Include member functions to perform the following tasks:

- (a) To create the vector
- (b) To modify the value of a given element
- (c) To multiply by a scalar value
- (d) To display the vector in the form (10, 20, 30, ...)

Write a program to test your class.

- 5.3 Modify the class and the program of Exercise 5.11 for handling 10 customers. **WEB**
- 5.4 Modify the class and program of Exercise 5.12 such that the program would be able to add two vectors and display the resultant vector. (Note that we can pass objects as function arguments.)
- 5.5 Create two classes **DM** and **DB** which store the value of distances. **DM** stores distances in metres and centimetres 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**. **WEB**
- 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 metres and centimetres depending on the object on display.
- 5.6 Refer Program 5.11 and write a function that receives two matrix objects as arguments and returns a new matrix object containing their multiplication result. **WEB**

itself when it is created. This member function called the *destructor* that destroys

6.2

Constructors

A constructor is a 'special' member function whose task is to initialize 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**. (Meaning of virtual will be discussed later in Chapter 9.)
- We cannot refer to their addresses.
- 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.

Remember, 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 i1=0;
```

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 object int1 and passes the values 0 and 100 to it. The same 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.

Remember, when the constructor is parameterized, we must provide appropriate arguments to the constructor. Program 6.1 demonstrates the passing of arguments to the constructor functions.

Program 6.1 Class with Constructors

```
#include <iostream>

using namespace std;

class integer
{
    int m, n;
public:
    integer(int, int);           // constructor declared

    void display(void)
    {
        cout << " m = " << m << "\n";
        cout << " n = " << n << "\n";
    }
};

integer i: 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
    int1.display();
    cout << "\nOBJECT2" << "\n";
    int2.display();
    return 0;
}
```


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)           // constructor 2
    {m = a; n = b;}
    integer(integer & i)             // constructor 3
    {m = i.m; n = i.n;}
};
```

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*. We learned in Chapter 4 that 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 6.2 shows the use of overloaded constructors.

Program 6.2 Overloaded Constructors

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

```
using namespace std;
```

(Contd.)

not do anything and is defined just to comply

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 the **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

✓ Copy Constructor

6.7

We briefly mentioned about the copy constructor in Sec. 6.3. We used the copy constructor

`Integer(Integer &i);` → reference variable.
→ Define.

- Sec 6.4 as one of the overloaded constructors.

As stated earlier, a copy constructor is used to declare and initialize an object from another object.
For example, the statement

`Integer i2(i1);` → Function call
→ main function.

will 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; copying P_1 obj to P_2 .

The process of initializing through a copy constructor is known as copy initialization. Remember the statement

$i2 = i1$ is Not a copy constructor. (Assign)
 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(=). We shall see more about this later.

A copy constructor takes a reference to an object of the same class as itself as an argument. Let us consider a simple example of constructing and using a copy constructor as shown in Program 6.4.

Program 6.4 Copy Constructor

```
#include <iostream>
```

UNIT IV – OPERATOR OVERLOADING, INHERITANCE AND POINTERS

OPERATOR OVERLOADING

Introduction:

We can redefine or overload most of the built-in operators in C++.

Overloaded operators are functions with special function name, with keyword operator followed by the symbol for the operator already exist

.Defining Operator overloading

The general form of an operator function is :

Return type class name :: operator op (arg list)

```
{  
  
    Function body  
  
}
```

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

Operator functions must be either member functions 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 .

Arguments may be passed either by value or by reference.

Operator functions are declared in the class using prototypes 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);             //comparision  
friend int operator==(vector, vector) //comparision
```

The process of overloading involve 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 for unary operators -x

and $x \text{ op } y$ for binary operators $x-y$
op x would be interpreted as operator op (x) for friend functions.

Overloading unary operators:

Operators that operates with single operand (values or expressions) are unary operators.

eg: +, -

Let us consider the unary minus operator. A minus operator when used as a unary, takes just one operand.. This operator changes the sign of an operand when applied to a basic data item. The unary minus when applied to an object should change the sign of each of its data items.

Program for Overloading unary minus

```
#include<iostream.h>
class space
{
int x;
int y;
int z;
public:
void getdata(int a, int b, int c);
void display();
void operator – ();    //overload unary minus
};

void space :: getdata(int a, int b, int c)
{
x = a;
y = b;
z = c;
}
void space :: display( )
{
cout<< x << “ ”;
cout<< y << “ ”;
cout<< z << “ \n” ;
}
void space :: operator – ( )
{
x = -x;
y = -y;
z = -z;
}
int main( )
{
space S;
S. getdata( 10, -20, 30);
```

```

cout<< "S : ";
S. display( );
- S;           // activates operator - () function
cout<< "-S : ";
S . display( );
return 0;
}

```

Output

S : 10, -20, 30

S : -10, 20, -30

Overloading Binary Operators:

The functional notation can be expressed as

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

by overloading the + operator using an operator+() function.

Overloading + operator

```

#include<iostream.h>
class complex
{
float x;      // real part
float y;      // imaginary part
public:
complex()     // constructor 1
{
}

complex (float real, float imag)    // constructor 2
{
x = real;
y = imag;
}
complex operator +(complex );
void display();
};
complex complex :: operator +(complex c)
{
complex temp;           // temporary
temp. x = x+c.x;        //float additions
temp. y = y+c.y;
return (temp);
}

```

```

void complex :: display()
{
    cout<< x << " +j" << y << "\n";
}

int main()
{
    complex C1,C2,C3;           //invokes constructor 1
    C1 = complex(2.5, 3.5);     //invokes constructor 2
    C2 = complex(1.6, 2.7);
    C3 = C1 + C2;
    cout<< "C1 = ";
    C1. display();
    C2. display();
    C3. display();
    return 0;
}

```

Output

C1 = 2.5+j3.5

C2 = 1.6+j2.7

C3 = 4.1+j6.2

The function is expected to add two complex values and return a complex value as the result but receives only one value as argument.

The statement

C3 = C1+C2;

invokes the operator function.

In the operator function, the data members of **C1** are accessed directly and the data members of **C2** are accessed using the dot operator.

In the statement,

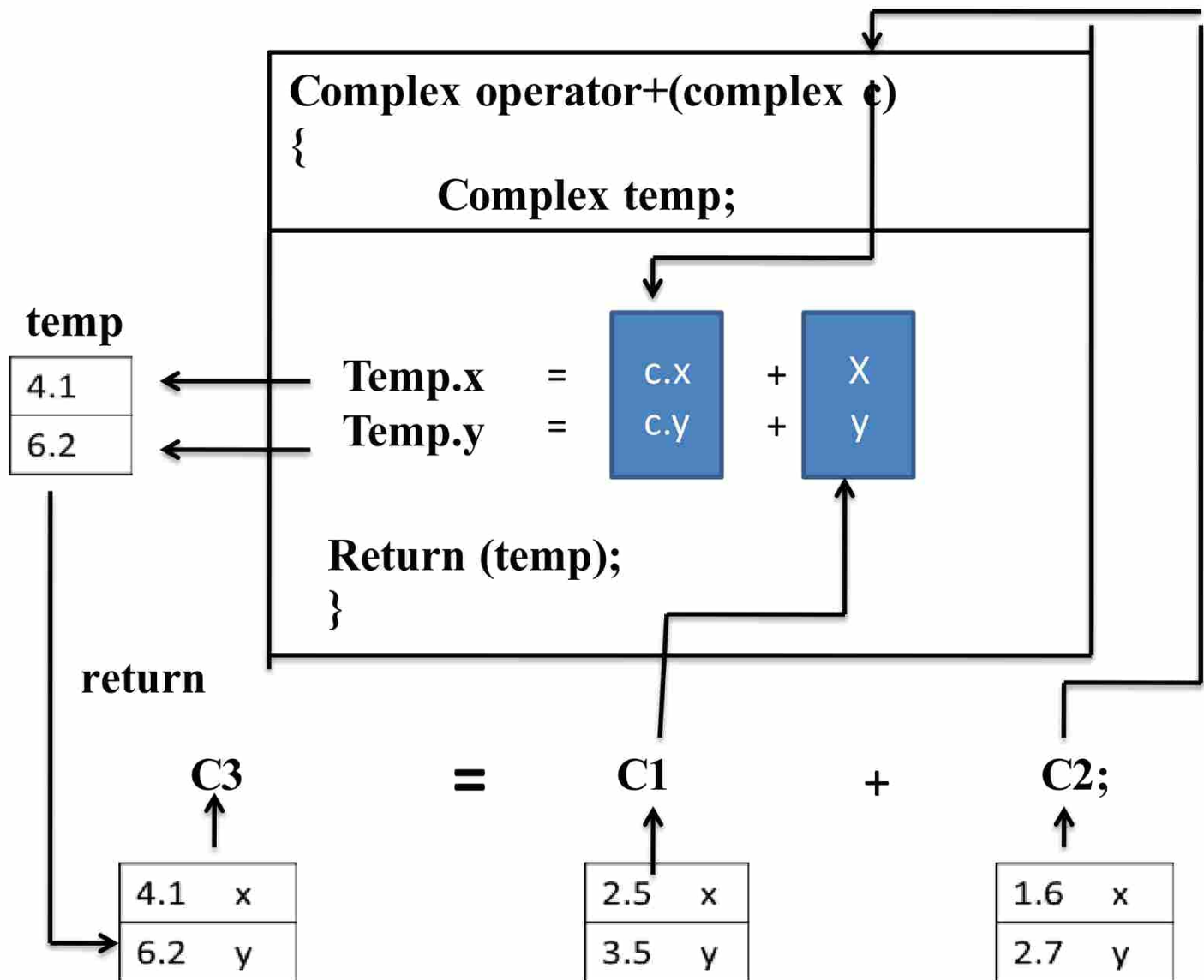
temp.x = x + c.x;

c.x refers to the object **C2** and **x** refers to the object **C1**.

temp.x is the real part of **temp** that has been created specially to hold the results of addition of **C1** and **C2**.

The function returns the complex **temp** to be assigned to **C3**.

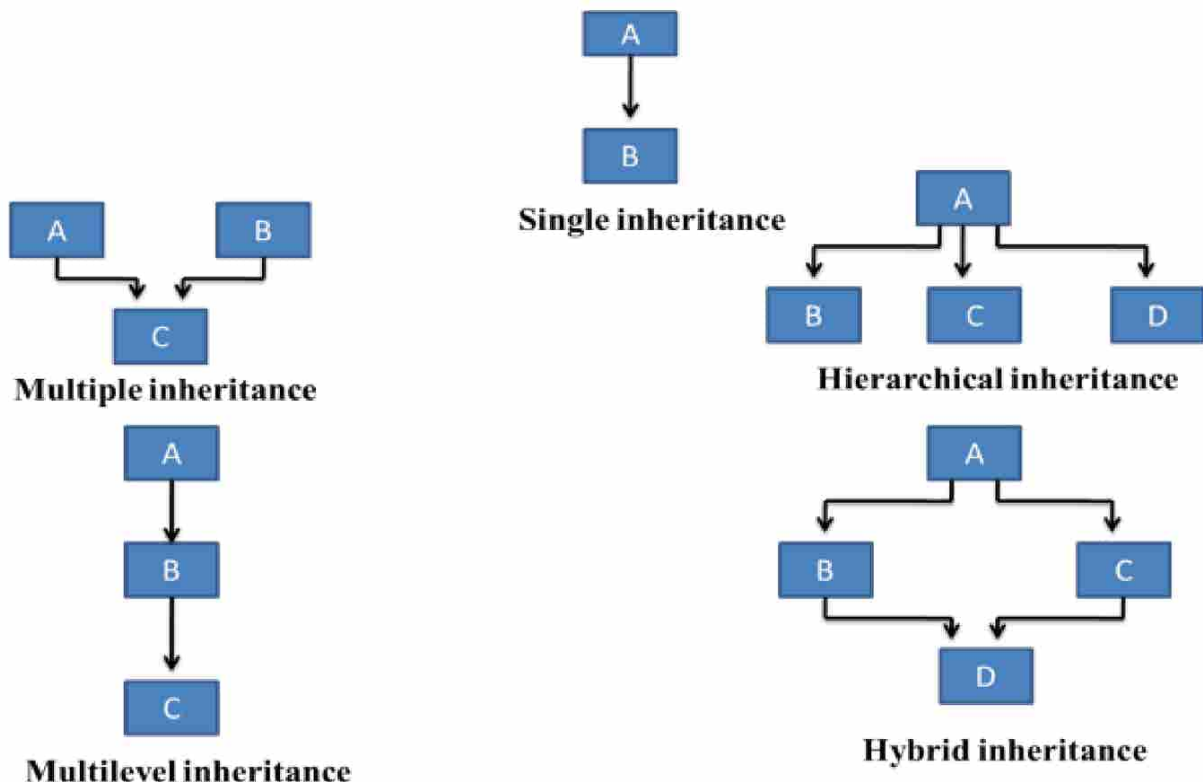
This is explained in the following diagram.



INHERITANCE

Introduction

- Inheritance is the mechanism by which one class can inherit the properties of another.
- The old class is referred to as the base class and the new one is called the derived class or sub class.
- 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.
- 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.



Forms of inheritance

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 : vis 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:

1. **class ABC :private XYZ** // private derivation
{
members of ABC
};
2. **class ABC :public XYZ** // public derivation
{
members of ABC
};
3. **class ABC :XYZ** //private derivation by default
{
members of ABC
};

ie., The default visibility mode is private.

When a base class is **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. A public member of a class can be accessed by its own objects using the dot operator.

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.

Single inheritance:

Program shows a **base class B** and a **derived class D**. The **class B** contains one private data member, and one public data member and three public member functions.

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

```
class B
```

```
{
    int a;                //private, not inheritance
public:
    int b;                // public, ready for inheritance
    void set_ab();
    int get_a(void);
    void show_a(void);
};
```

```
class D : public B        //public derivation
```

```
{
    int c;
public:
    void mul(void);
    void display(void);
};
```

```
void B :: set_ab(void)
```

```
{
    a = 5;
    b = 10;
}
```

```
int B :: get_a()
```

```
{
    return a;
}
```

```
void B :: show_a()
```

```
{
    cout<<"a = "<<a<<"\n";
}
```

```
void D :: mul()
```

```
{
    c = b*get_a();
}
```

```
void D :: display()
```

```
{
```

```

    cout<<"a="<<get_a()<<"\n";
    cout<<"b="<<b<<"\n";
    cout<<"c="<<c<<"\n\n";
}

int main()
{
    D d;
    d.et_ab();
    d.mul();
    d.show_a();
    d.display();
    d.b=20;
    d.mul();
    d.display();
    return 0;
}

```

Output

```

a=5
a=5
b=10
c=50

```

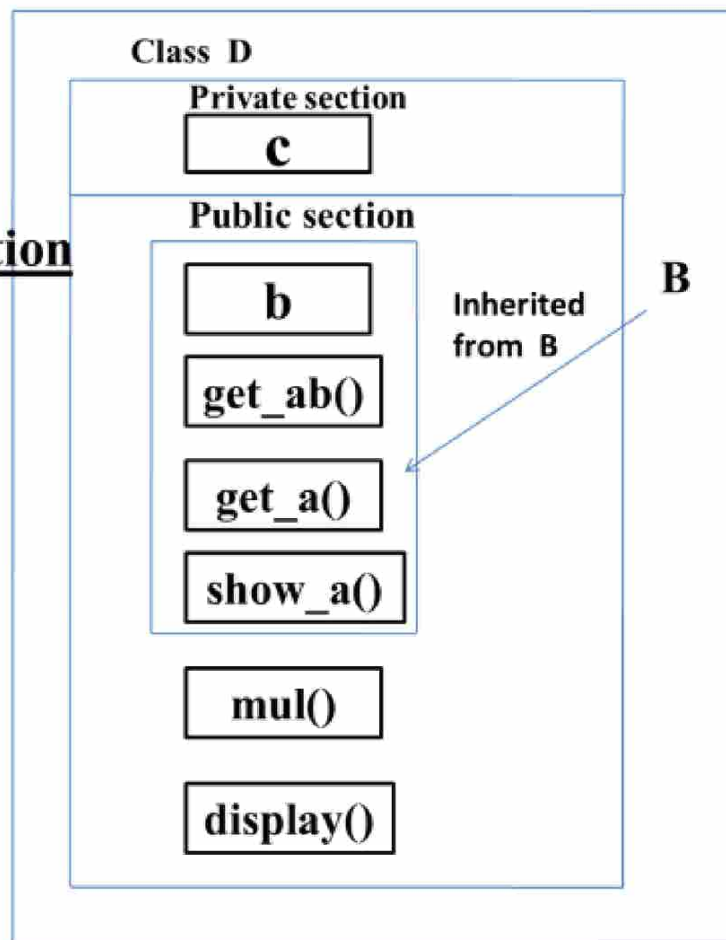
```

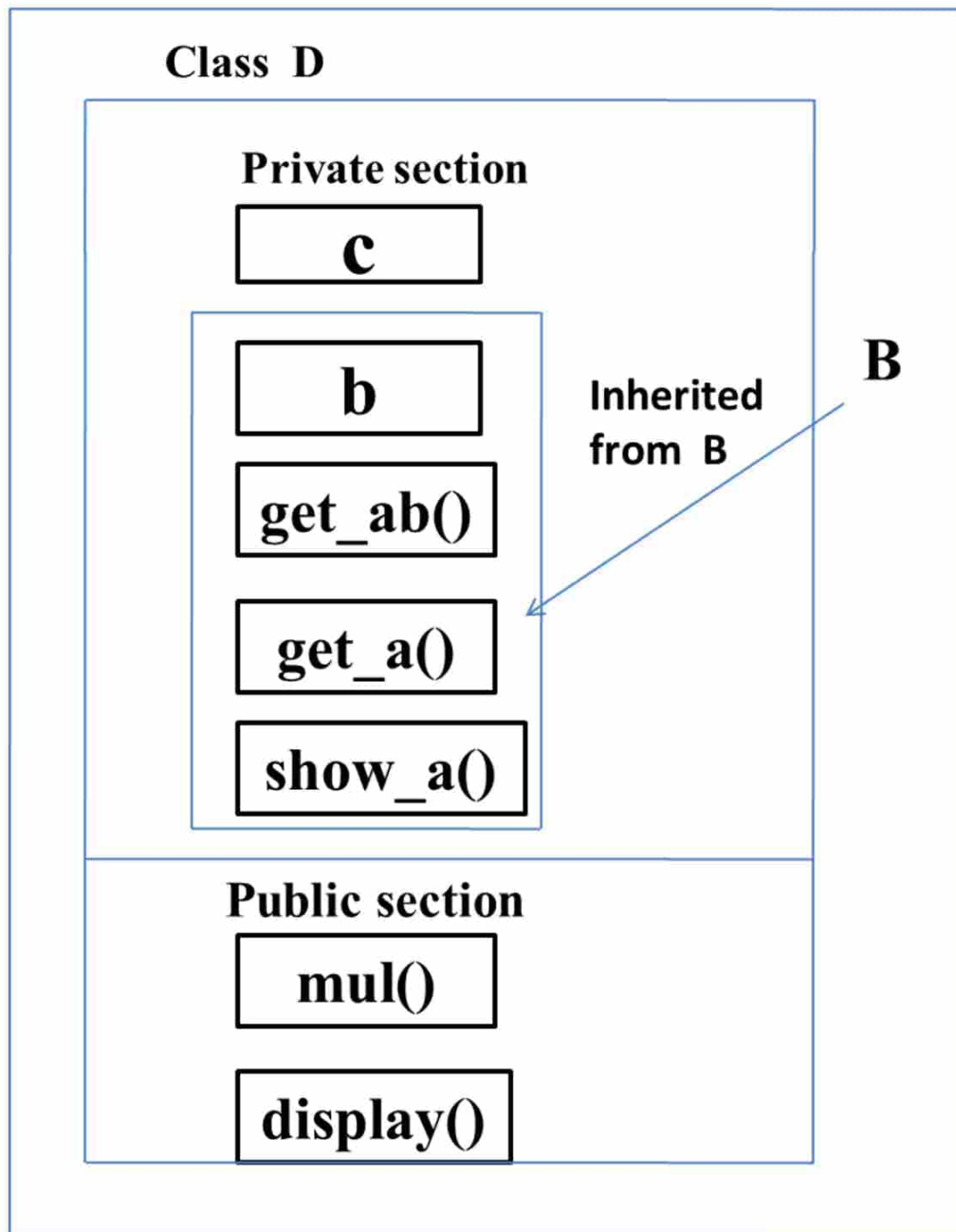
a=5
b=20
c=100

```

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.

Public derivation





Private derivation

In private derivation, the public members of the base class become private members of the derived class. Therefore, the objects of D cannot have direct access to the public member functions of B.

```
#include<iostream.h>
class B
{
    int a;        //private, not inheritance
public:
    int b;        // public, ready for inheritance
```



```

    void get_ab();
    int get_a(void);
    void show_a(void);
};

class D : private B           //private inheritance
{
    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 :: show_a()
{
    cout<<"a = "<<a<<"\n";
}

void D :: mul()
{
    get_ab();
    c = b*get_a();    //'a' cannot be used directly
}

void D :: display()
{
    show_a();          // outputs value of 'a'
    cout<<"b="<<b<<"\n";
    cout<<"c="<<c<<"\n\n";
}

int main()
{
    D d;
    // d.get_ab();      won't work
    d.mul();
    //d.show_a();       won't work
    d.display();
}

```

```

    //d.b=20;          won't work,
    d.mul();
    d.display();
    return 0;
}

```

Enter values for a and b: 5 8

a = 5

b = 8

c = 40

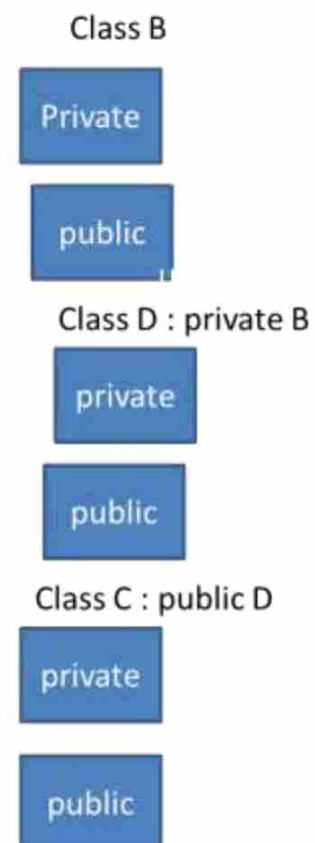
Enter values for a and b: 10 20

a = 10

b = 20

c = 200

Single Inheritance



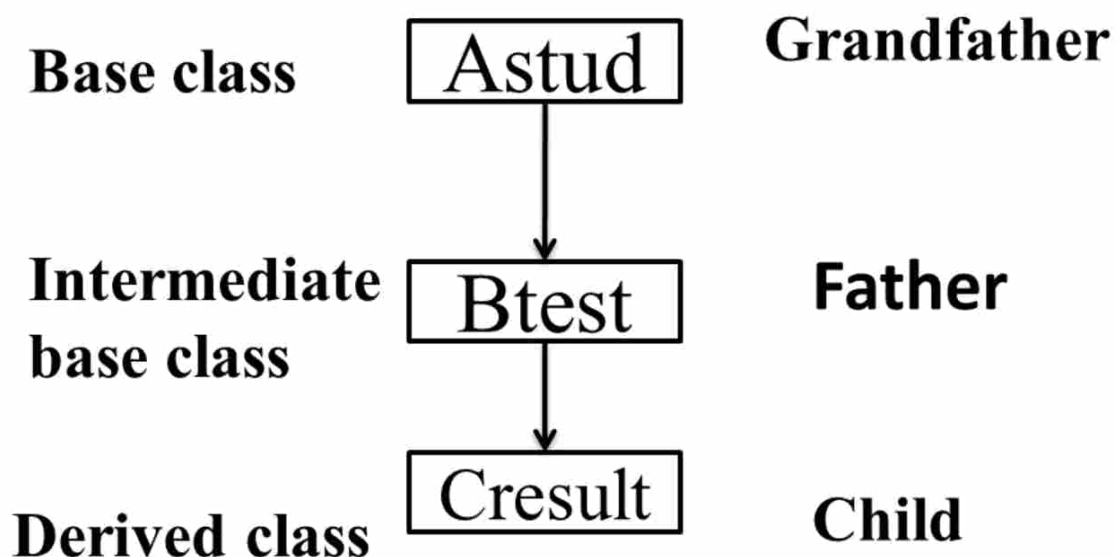
Multilevel Inheritance

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.

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.



Multilevel Inheritance

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.

```
#include<iostream.h>
class student
{
protected:
    int rollnumber;
public:
    void getnumber(int);
    void putnumber();
};
void student :: getnumber(int a)
{
    rollnumber = a;
}

void student :: putnumber()
{
    cout<<"Roll Number: " <<rollnumber<<"\n";
}
class test : public student //first level derivation
{
protected:
    float sub1;
    float sub2;
public:
    void getmarks(float, float);
    void putmarks();
};

void test :: getmarks(float x, float y)
{
    sub1 = x;
    sub2 = y;
}
void test :: putmarks()
{
    cout<<"marks in sub1 = " <<sub1<<"\n";
    cout<<"marks in sub2 = " <<sub2<<"\n";
}
class result : public test    //second level derivation
{
    float total;
public:
```

```

void display();
};

void result :: display()
{
total = sub1+sub2;
putnumber();
putmarks();
cout<<"Total = "<< total<<"\n";
}

int main()
{
result student1;          // obj student1 created
student1.getnumber(10011);
student1.getmarks(75.0, 68.0);
student1.display();
return 0;
}

```

Output

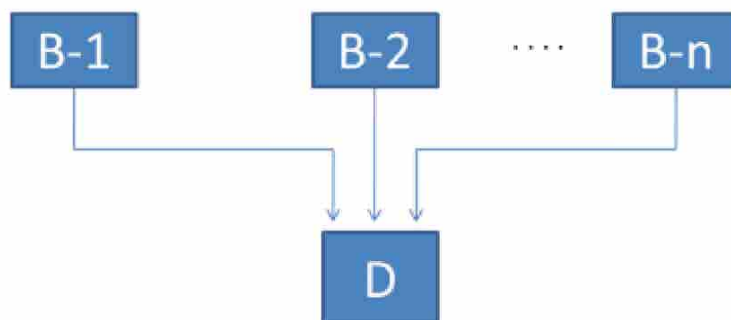
```

Roll Number : 10011
Marks in sub1 = 75.0
Marks in sub2 = 68.0
Total = 143.0

```

Multiple Inheritance

Multiple inheritance allows to combine the features of several existing classes as starting pointing for defining new classes. A class can inherit the attribution of two or more classes as shown in Fig. This is known as **multiple inheritance**.



Multiple Inheritance

The syntax of a derived class with multiple base classes is

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.

Program to Multiply two numbers:

```
#include<iostream.h>  
class M  
{  
protected:  
    int m;  
public:  
    void getm(int);  
};  
class N  
{  
protected:  
    int n;  
public:  
    void getn(int);  
};  
  
class P : public M, public N  
{  
public:  
    void display();  
}  
void M :: getm(int x)  
{  
    m = x;  
}  
void N :: getn(int y)  
{  
    n = y;  
}
```

```

Void P :: display()
{
cout<<"m="<<m<<"\n";
cout<<"n="<<n<<"\n";
cout<<"m x n="<<m*n<<"\n";
}
in 0;
int main()
{
P p;
p.getm(50);
p.getn(10);
p.display();
return 0;
}

```

Output

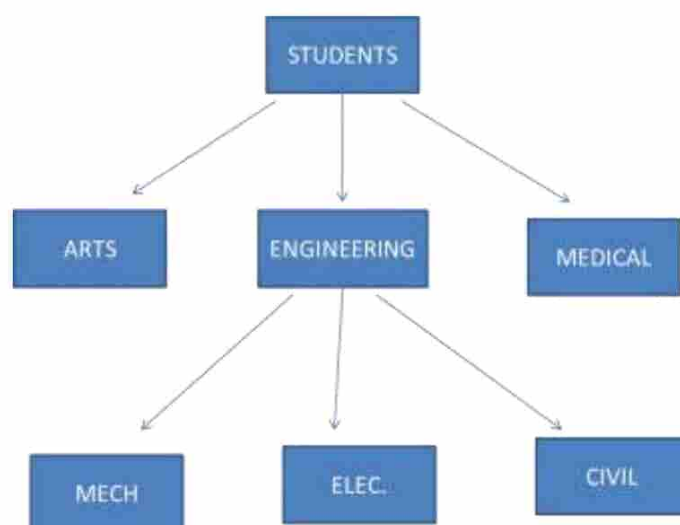
```

m = 50
n = 10
m x n = 500

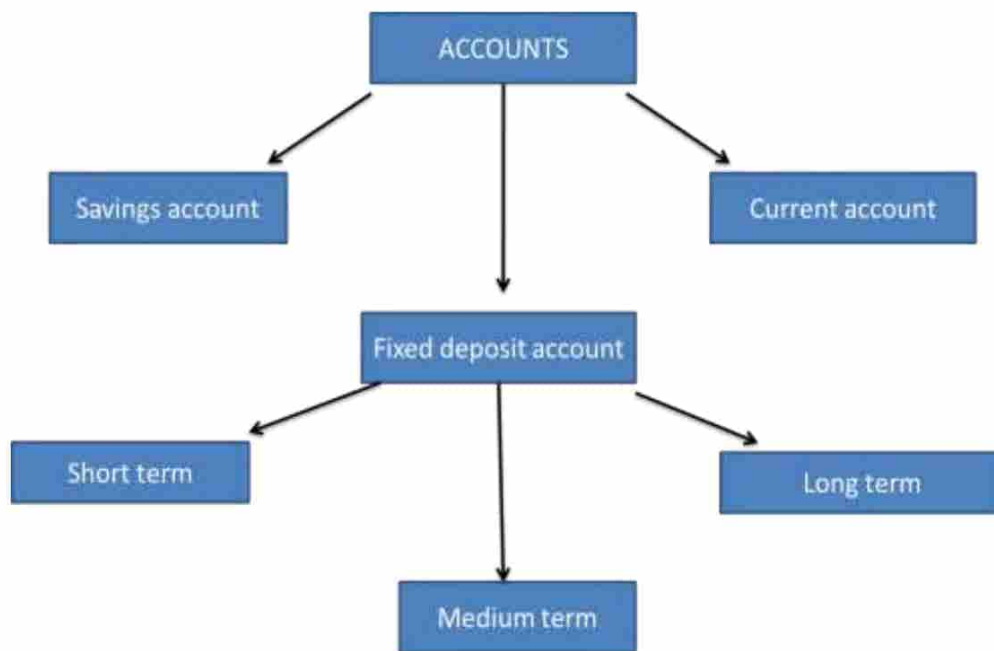
```

Hierarchical Inheritance

In Hierarchical Inheritance, certain features of one level are shared by many others below that level. As an example, Fig shows a Hierarchical classification of students in a university. Another example could be the classification of accounts in a commercial bank as shown in Fig. All the students have certain things in common and, similarly, all the accounts possess certain common features.



Hierarchical classification of students

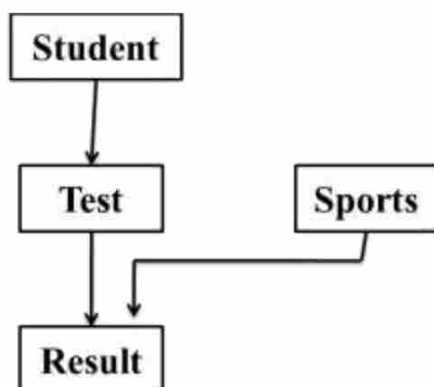


Classification of bank accounts

The base class will include all the features that are common to the subclasses. 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.

Hybrid Inheritance

Hybrid inheritance is a combination of multiple **inheritance** and **multilevel inheritance**. A class Result is derived from two classes Test, Sports as in multiple **inheritance**.



Multilevel, Multiple Inheritance

```

#include<iostream.h>
class student
{
protected:
    int rollnumber;
public:
    void getnumber(int);
    void putnumber();
};
void student :: getnumber(int a)
{
    rollnumber = a;
}

void student :: putnumber()
{
    cout<<"Roll Number: "<<rollnumber<<"\n";
}
class test : public student
{
protected:
    float sub1;
    float sub2;
public:
    void getmarks(float, float);
    void putmarks();
};

void test :: getmarks(float x, float y)
{
    sub1 = x;
    sub2 = y;
}
void test :: putmarks()
{
    cout<<"marks in sub1 = "<<sub1<<"\n";
    cout<<"marks in sub2 = "<<sub2<<"\n";
}
class sports
{
protected:
    float score;
public:

```

```

void getscore(float s)
{
    score = s;
}
void putscore()
{
    cout<<"Sports wt: "<<score<<"\n";
}
};
class result : public test, public sports
{
    float total;
public:
    void display();
};

void result :: display()
{
    total = sub1+sub2+score;
    putnumber();
    putmarks();
    putscore();
    cout<<"Total = "<< total<<"\n";
}
int main()
{
    result student1;
    student1.getnumber(10011);
    student1.getmarks(75.0, 59.5);
    student1.getscore(6.0);
    student1.display();
    return 0;
}

```

Output

```

Roll Number : 10011
Marks in sub1 = 75.0
Marks in sub2 = 59.5
Sports wt: 6.0
Total = 140.5

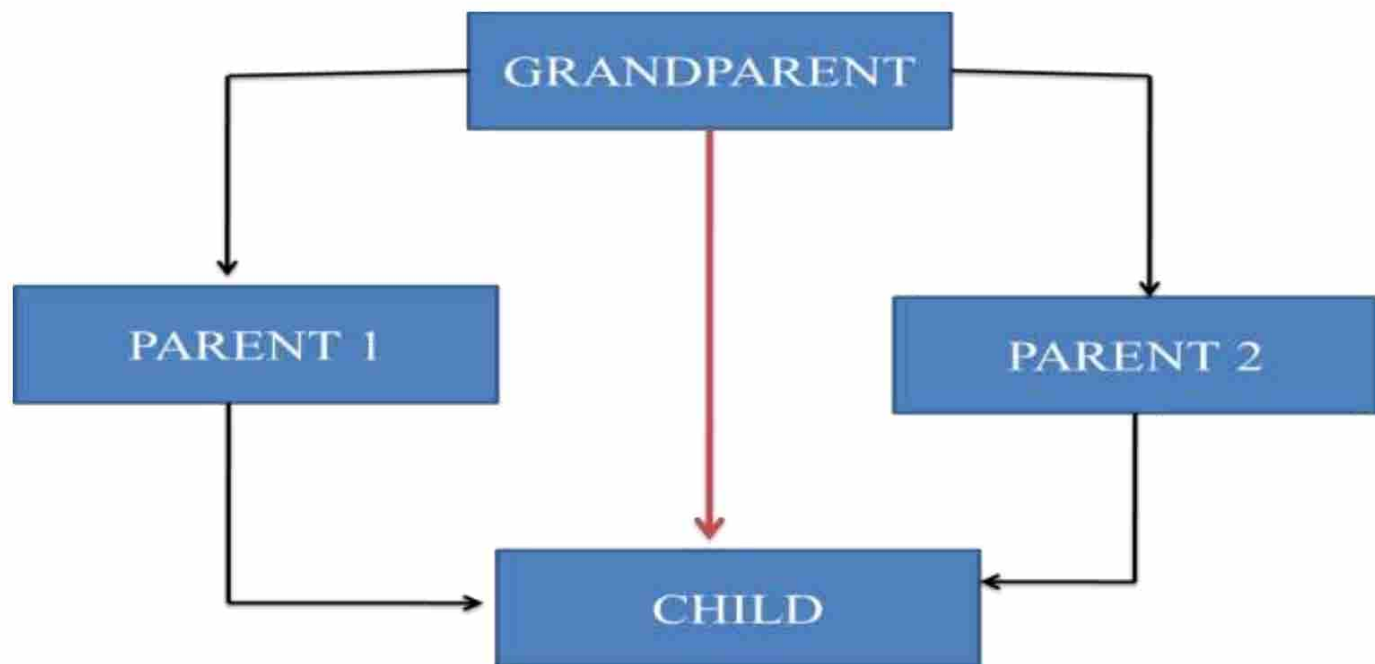
```

Virtual Base Class

Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved. This is illustrated in Fig. The child has

two direct base classes parent 1 and parent 2 which themselves have a common base class grandparent. The child inherits the traits(quality or character) of grandparent via two separate paths. It can also inherit directly as shown by the vertical line. The grandparent is sometimes referred to as indirect base class.

Multipath Inheritance



All the public and protected members of grandparent are inherited into child twice, first via parent 1 and again via parent 2. The duplication of inherited members due to these multiple paths can be avoided by making the common base class as virtual base class while declaring the direct or intermediate base classes which is shown as follow:

```

class A                                     //grandparent
{
.....
};
class B1: virtual public A                 // parent1
{
.....
};
Class B2 : public virtual A               //parent2
{
.....
};
class C : public B1, public B2             //child
{
.....                                     //only one copy of A

```

```

..... // will be inherited
};

```

For example, consider the student results processing system. The class sports derives the rollnumber from the class student.

```

#include<iostream.h>
class student
{
protected:
    int rollnumber;
public:
    void getnumber(int);
    void putnumber();
};
void student :: getnumber(int a)
{
    rollnumber = a;
}

void student :: putnumber()
{
    cout<<"Roll Number: "<<rollnumber<<"\n";
}
class test : virtual public student
{
protected:
    float sub1;
    float sub2;
public:
    void getmarks(float, float);
    void putmarks();
};

void test :: getmarks(float x, float y)
{
    sub1 = x;
    sub2 = y;
}

void test :: putmarks()
{
    cout<<"marks in sub1 = "<<sub1<<"\n";
    cout<<"marks in sub2 = "<<sub2<<"\n";
}

class sports :: public virtual student
{

```

```

protected:
float score;
public:

void getscore(float s)
{
score = s;
}
void putscore()
{
cout<<"Sports wt: "<<score<<"\n";
}
};

class result : public test, public sports
{
float total;
public:
void display();
};

void result :: display()
{
total = sub1+sub2+score;
putnumber();
putmarks();
putscore();
cout<<"Total = "<< total<<"\n";
}

int main()
{
result student1;
student1.getnumber(10011);
student1.getmarks(75.0, 59.5);
student1.getscore(6.0);
student1.display();
return 0;
}

```

Output

```

Roll Number : 10011
Marks in sub1 = 75.0
Marks in sub2 = 59.5
Sports wt: 6.0
Total = 140.5

```

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. It is a design concept in program development and provides a base upon which other classes may be built. The **student** class is an abstract class since it was not used to create any objects.

POINTERS

What Is A Pointer?

A pointer is a variable that holds the address of a memory location. All the variables we declare, have a specific address in memory. We declare a pointer variable to point to these addresses in memory.

The general syntax for declaring a pointer variable is:

```
datatype * variable_name;
```

For Example, the declaration

```
int* ptr;
```

This means ptr is a pointer that points to a variable of type int. Hence a pointer variable always contains a memory location or address.

Consider we have the following declarations:

```
int p, *ptr; //declare variable p and pointer  
            variable ptr
```

```
p = 4;           //assign value 4 to variable p  
ptr = &p;        //assign address of p to pointer  
                variable ptr
```

As ptr has an address of variable p, *ptr will give the value of variable p.

In memory, these declarations will be represented as follows:



This is the internal representation of pointer in memory. When we assign the address variable to the pointer variable, it points to the variable as shown in the representation above.

Pointer Arithmetic

A pointer variable always points to the address in memory. Among the operations that we can perform, the following arithmetic operations that are carried out on pointers.

Increment operator (++)

Decrement operator (--)

Addition (+)

Subtraction (-)

```
#include <iostream.h>
#include <string.h>
int main()
{
    int myarray[5] = {2, 4, 6, 8, 10};
    int* myptr;
    myptr = myarray;
    cout<<"First element in the array:" <<*myptr <<endl;
    myptr ++;

    cout<<"next element in the array :"<<*myptr<<endl;
    myptr +=1;
    cout<<"next element in the array :"<<*myptr<<endl;
    myptr--;
    cout<<"next element in the array :"<<*myptr<<endl;
    myptr -= 1;
    cout<<"next element in the array :"<<*myptr<<endl;
    return 0;
}
```

Output:

```
First element in the array :2
next element in the array :4
next element in the array :6
next element in the array :4
next element in the array :2
```

The increment operator ++ increments the pointer and points to the next element in the array. Similarly, the decrement operator decrements the pointer variable by 1 so that it points to the previous element in the array.

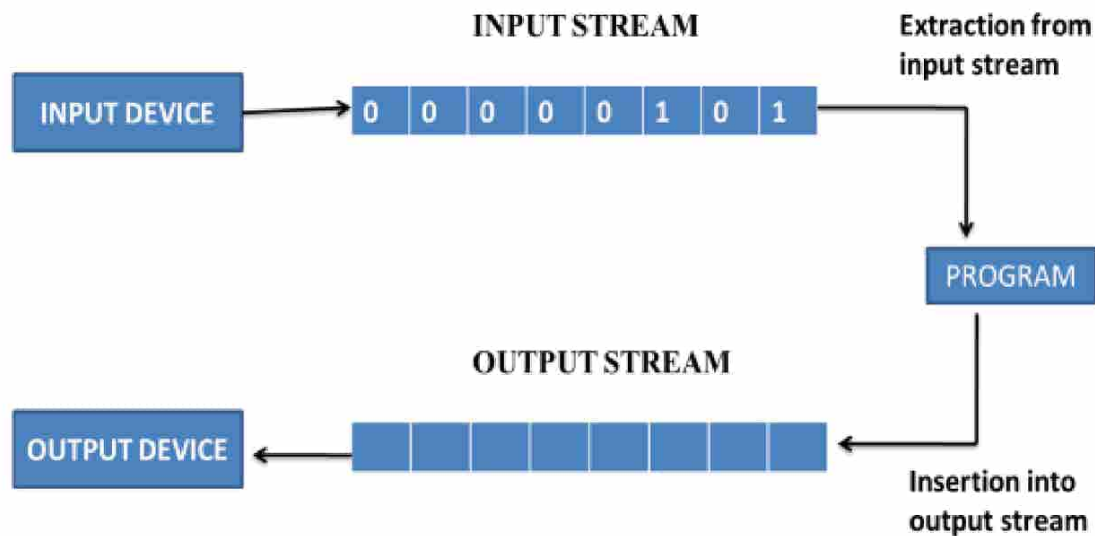
We also use + and – operators. First, we have added 1 to the pointer variable. The result shows that it points to the next element in the array. Similarly, – operator makes the pointer variable to point to the previous element in the array.

Apart from these arithmetic operators, we can also use comparison operators like ==, < and >.

UNIT V – MANAGING CONSOLE I/O OPERATIONS

C++ Streams

A stream is a sequence of bytes. It acts either as a source from which the input data can be obtained or as a destination to which the output data can be sent. The source stream that provides data to the program is called the input stream and the destination stream that receives output from the program is called the output stream. A program extracts the bytes from an input stream and inserts bytes into an output stream as illustrated in Fig.



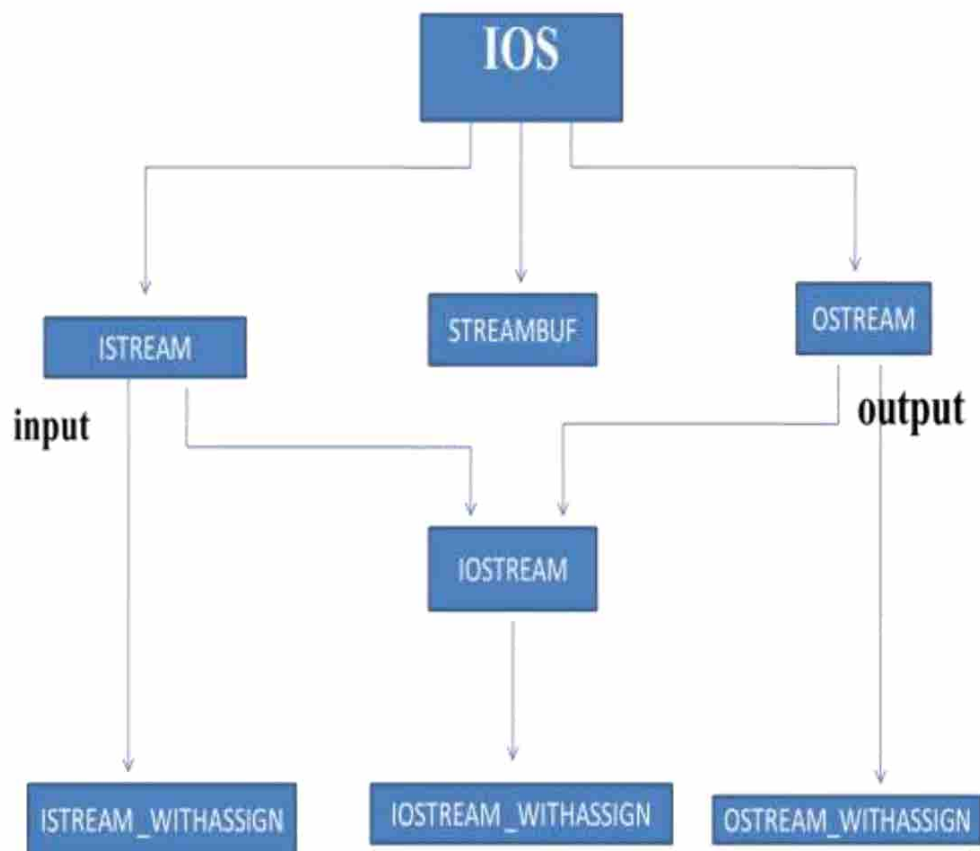
Data streams

The data in the **input stream** can come from the **keyboard** or any other storage device. Similarly, the data in the **output stream** can go to the **screen** or any other storage device. A **stream** acts as an **interface** between the program and the input/ output device. **Cin** represents the **input stream** connected to the standard input device(keyboard) and **cout** represents the output stream connected to the standard output device(screen)

C++ Stream classes

The C++ I/O system contains a hierarchy of classes that are used to define various streams to deal with both the console and disk files. These classes are called stream classes. Fig shows the hierarchy of the stream classes used for input and output operations with the console unit. These classes are declared in the header file `iostream`. This file should be included in all the programs that communicate with the console unit

Stream classes for console I/O operations



IOS is the base class for istream(input stream) and ostream(output stream) which are in turn, base classes for iostream(input/output stream). The class IOS is declared as the virtual base class so that only one copy of its members are inherited by the stream.

Unformatted I/O operations

Overloaded operators >> and <<

The >> operator is overloaded in the istream class and << is overloaded in the ostream class. The following is the general format for reading data from the keyboard.

cin>>variable1>>variable2>>....>>variableN;

variable1, variable2.... are valid C++ variable names that have been declared already. This statement will cause the computer to stop the execution and look for input data from the keyboard. The input data for this statement would be
data1 data2datan

The input data are separated by white spaces and should match the type of variable in the cin list.

The operator >> reads the data character by character and assigns it to the indicated location. The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type. For eg.

```
Int code;  
cin>>code;
```

Suppose the following data is given as input:

4258D

The operator will read the characters upto 8 and the value 4258 is assigned to **code**. The character **D** remains in the input stream and will be input to the next cin statement.

The general form for displaying data on the screen is

```
cout<< item1<<item2<<.....<<itemN;
```

The items item1 through itemN may be variables or constants of any basic type.

put() and get() functions

The classes **istream** and **ostream** define two member functions **get()** and **put()** respectively to handle the single character input/output operations. There are two types of **get()** functions. We can use both **get(char*)** and **get()** prototypes to fetch a character including the blank space, tab and the newline character. The **get(char*)** version assigns the input character to its argument and the **get()** version returns the input character.

Since these functions are members of the input/output stream classes, we must invoke these functions using an appropriate object.

Example

```
char c;  
cin.get(c);    //get a character from keyboard  
               //and assign it to c  
while(c!='\n')  
{  
    cout<<c;    //display the character on screen  
    cin.get(c); //get another character  
}
```

This code reads and displays a line of text(terminated by a newline character).

The **get()** version is used as follows:

```
.....  
char c;  
c=cin.get(); // for cin.get(c)  
.....  
.....
```

The value returned by the function **get()** is assigned to the variable **c**.

The function **put()**, a member of **ostream** class, can be used to output a line of text, character by character.

For example

```
cout.put('x');  
displays the character x and  
cout.put(ch);  
displays the value of variable ch.
```

```
#include(iostream.h>  
int main()  
{  
int count=0;  
char c;  
cout<<"INPUT TEXT\n";  
cin.get(c);  
  
while(c!='\n')  
{  
cout.put(c);    not equal to  
count++;  
cin.get(c);  
}  
cout<<"\nNumber of characters="<<count<<"\n";  
return 0;  
}  
input  
INPUT TEXT  
Object Oriented Programming  
Output  
Object Oriented Programming  
Number of characters=27
```

getline() and write() Functions

The **getline()** function reads a whole line of text that ends with a newline character. This function can be invoked by using the object as follows:

```
cin.getline(line, size);
```

This function call invokes the function **getline()** which reads character input into the variable **line**. The reading is terminated as soon as either the newline character '**\n**' is encountered or size-1 characters are read. The newline character is read but not saved. Consider the following code

```
char name[20];  
cin.getline(name, 20);
```

Assume that we have given the following input through the keyboard:

Bjarne Stroustrup <press RETURN>

This input will be read correctly and assigned to the character array name. Let us suppose the input is as follows.

Object Oriented Programming

<press RETURN>

In this case, the input will be terminated after reading the following 19 characters.

Object Oriented Pro

The two blank spaces contained in the string are also taken into account.

```
#include<iostream.h>
int main()
{
int size=20;
char city[20];
cout<<"Enter city name: \n";
cin>>city;           New Delhi
cout<<"city name:"<<city<<"\n\n";   New
cout<<"Enter city name again:\n";
cin.getline(city, size);
cout<<"city name now:"<<city<<"\n\n"; Delhi
cout<<"Enter another city name:\n";
cin.getline(city,size);   Greater Bombay
cout<<"New city name:"<<city<<"\n\n"; Greater
                           Bombay

return 0;
}
```

Output

```
Enter city name:
New Delhi
City name: New
Enter city name again:
city name now: Delhi
Enter another city name:
Greater Bombay
New city name: Greater Bombay
```

The **write()** function displays an entire line and has the following form:

cout.write(line, size);

The first argument **line** represents the name of the string to be displayed and the second argument **size** indicates the number of characters to display. If the size is greater than the length of line, then it displays beyond the bounds of line.


```

#include<iostream.h>
#include<string.h>
int main()
{
char* string1="C++";
char* string2="Programming";
int m=strlen(string1);  string length of str1 3
int n=strlen(string2);
for(int i=1; i<n; i++)
{
cout.write(string2, i);
cout<<"\n";
}

for(i=n; i>0; i--)
{
cout.write(string2,i);
cout<<"\n";
}
//concatenating strings
cout.write(string1,m).write(string2, n);
cout<<"\n";
return 0;
}

```

p
pr
pro
prog
progr
progra
program
programm
programm
programmin
programming
programmin
programmi
programm
program
progra
progr
prog
pro

Formatted console I/O operations

C++ supports a number of features that could be used for formatting the output. These features include:

- * **ios** class functions and flags
- * Manipulators
- * User-defined output functions

The **ios** class contains a large number of member functions that would help us to format the output in a number of ways.

IOs format functions

Function	Task
width()	To specify the required field size for displaying an output value
precision()	To specify the number of digits to be displayed after the decimal point of a float value
fill()	To specify a character that is used to fill the unused portion of a field
setf()	To specify format flags that can control the form of output display (such as left-justification and right-justification)
unsetf()	To clear the flags specified

Manipulators are special functions that can be included in the I/O statements to alter the format parameters of a stream. Table shows some important manipulator functions that are frequently used. To access these manipulators, the file `iomanip` should be included in the program.

Manipulators

Manipulators	Equivalent ios function
setw()	Width()
setprecision()	Precision()
setfill()	Fill()
setiosflags()	Setf()
Resetiosflags()	Unsetf()

Defining Field Width: width()

The **width()** function can be used to define the width of a field necessary for the output of an item. Since, it is a member function, an object can be used to invoke it, as shown below:

```
cout.width(w);
```

where **w** is the field width(number of columns). The output will be printed in a field of **w** characters wide at the right end of the field. The **width()** function can specify the field width for only one item. After printing one item, it will revert back to the default. For eg, the statements

```
cout.width(5);  
cout<<543<<12<<"\n";
```

will produce the following output:

		5	4	3	1	2
--	--	---	---	---	---	---

The value 543 is printed right justified in the first five columns. The specification width(5) does not retain the setting for printing the number 12.

```
cout.width(5);  
cout<<543;  
cout.width(5);  
cout<<12>>>"\n";
```

This produce the following output:

		5	4	3				1	2
--	--	---	---	---	--	--	--	---	---

C++ never truncates the values and therefore, if the specified field width is smaller than the size of the value to be printed, C++ expands the field to fit the value.

```
#include<iostream h>  
int main()  
{  
int items[4]={ 10,8,12,15};  
int cost[4]={75,100,60,99};  
cout.width(5);  
cout<< "ITEMS";  
cout.width(8);
```

```

cout<< "COST":
cout.width(15);
cout<< "TOTAL VALUE" << "\n";
int sum=0;

for(int i=0; i<4; i++)
{
cout.width(5);
cout<< items[i];
cout.width(8);
cout<< cost[i];
int value=items[i]*cost[i];
cout.width(15);
cout<<value<< "\n";
sum=sum+value;
}
cout<< "\n Grand Total=";
cout.width(2);
cout<<sum<< "\n";
return 0;
}

```

Output

I T E M S				C O S T				T O T A L V A L U E							
		1 0				7 5								7 5 0	
		8				1 0 0								8 0 0	
		1 2				6 0								7 2 0	
		1 5				9 9								1 4 8 5	

Grand Total = 3755

Output

I T E M S	C O S T	T O T A L V A L U E
1 0	7 5	7 5 0
8	1 0 0	8 0 0
1 2	6 0	7 2 0
1 5	9 9	1 4 8 5

Grand Total = 3755

Setting Precision: `precision()`

By default, the floating numbers are printed with six digits after the decimal point. We can specify the number of digits to be displayed after the decimal point while printing the floating point numbers. This can be done by using the **`precision()`** member function as follows:

```
cout.precision(d);
```

where **`d`** is the number of digits to the right of the decimal point. For example, the statements

```
cout.precision(3);  
cout<<sqrt(2)<<"\n";  
cout<<3.14159<<"\n";  
cout<<2.50032<<"\n";
```

will produce the following output:

```
1.414   (truncated)  
3.142   (rounded to the nearest cent)  
2.5     (no trailing zeros)
```

The `precision()` function retains the setting in effect until it is reset.

```
cout.precision(3);  
cout<<sqrt(2)<<"\n";  
cout.precision(3);    //Reset the precision  
cout<<3.14159<<"\n";
```

we can also combine the field specification with the precision writing. Example:

```
cout.precision(2);  
cout.width(5);  
cout<<1.2345;
```

The first two statements instruct: “print two digits after the decimal point in a field of five character width”. Thus, the output will be

	1	.	2	3
--	----------	----------	----------	----------

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

```
#include<cmath.h>
```

```
int main()
```

```
{
```

```
cout<<"Precision set to 3 digits \n\n";
```

```
cout.precision(3);
```


VALUE	S Q R T _ O F _ V A L U E
1	1
2	1 . 4 1
3	1 . 7 3
4	2
5	2 . 2 4

Precision set to 5 digits

Sqrt(10) = 3.1623

Sqrt(10) = 3.162278 default setting

Filling and Padding: fill()

The unused positions of the field are filled with white spaces, by default. We can use the **fill()** function to fill the unused positions by any desired character. It is used in the following form:

```
cout.fill(ch);
```

where **ch** represents the character which is used for filling the unused positions. Example:

```
cout.fill('*');
```

```
cout.width(10);
```

```
cout<<5250<<"\n";
```

The output would be

*	*	*	*	*	*	5	2	5	0
---	---	---	---	---	---	---	---	---	---

Financial institutions and banks use this kind of padding while printing cheques so that no one can change the amount easily.

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

```
int main()
```

```
{
```

```
cout.fill('<');
```

```

cout.precision(3);

for(int n=1; n<=6; n++)
{
    cout.width(5);
    cout<<n;
    cout.width(10);
    cout<<1.0/float(n)<<"\n";
    if(n==3)
        cout.fill('>');
}
cout<<"\nPadding changed \n\n";
cout.fill('#');    //fill() reset
cout.width(15);
cout<<12.145678<<"\n";
return 0;
}

```

The output of the program

<	<	<	<	1	<	<	<	<	<	<	<	<	<	1
<	<	<	<	2	<	<	<	<	<	<	<	0	.	5
>	>	>	>	3	>	>	>	>	>	0	.	3	3	3
>	>	>	>	4	>	>	>	>	>	>	0	.	2	5
>	>	>	>	5	>	>	>	>	>	>	>	0	.	2
>	>	>	>	6	>	>	>	>	>	0	.	1	6	7

Padding changed

#	#	#	#	#	#	#	#	#	#	#	1	2	.	3
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

The output of the program

```
< < < < 1 < < < < < < < < 1
< < < < 2 < < < < < < < 0 . 5
> > > > 3 > > > > > 0 . 3 3 3
> > > > 4 > > > > > > 0 . 2 5
> > > > 5 > > > > > > > 0 . 2
> > > > 6 > > > > > 0 . 1 6 7
```

Padding changed

```
# # # # # # # # # # # 1 2 . 3
```

Formatting Flags, Bit-fields and setf()

When the function **width()** is used, the value is printed right-justified in the field width created. The **setf()**, a member function of the **ios** class, can be used to print the text left-justified . The **setf()** (set flags) function can be used as follows:

```
cout.setf(arg1, arg2);
```

The **arg1** is one of the **formatting flags** defined in the class **ios**. The formatting flag specifies the format action required for the output. Another **ios** constant, **arg2**, known as **bit field** specifies the group to which the formatting flag belongs. **3004.5732**
/3.0045732x10^3 /3.0045732e+03

Flags and bit fields for setf() function

Format required	Flag(arg1)	Bit field(arg 2)
Left justified output	ios :: left	ios :: adjustfield
Right justified output	ios :: right	ios :: adjustfield
Padding after sign(++20)	ios :: internal	ios :: adjustfield
Scientific notation	ios :: scientific	ios :: floatfield
Fixed point notation	ios :: fixed	ios :: floatfield
Decimal base	ios :: dec	ios :: basefield
Octal base	ios :: oct	ios :: basefield
Hexadecimal base	ios :: hex	ios :: basefield

Table shows the bit fields, flags and their format actions. There are three bit fields and each has a group of format flags.

```
cout.setf(ios :: left, ios :: adjustfield);
cout.setf(ios :: scientific, ios :: floatfield);
```

Consider the following segment of code:

```
cout.fill('*');
cout.setf(ios :: left, ios :: adjustfield);
cout.width(15);
cout<<"TABLE 1"<<"\N";
```

This will produce the following output:

T	A	B	L	E		1	*	*	*	*	*	*	*	*
---	---	---	---	---	--	---	---	---	---	---	---	---	---	---

The statements

```
cout.fill('*');
cout.precision(3);
cout.setf(ios :: internal, ios :: adjustfield);
cout.setf(ios :: scientific, ios :: floatfield);
cout.width(15);
cout<<-12.34567<<"\n";
```

will produce the following output:

-	*	*	*	*	*	1	.	2	3	5	e	+	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

The sign is left-justified and the value is right justified. The space between them is padded with stars. The value is printed accurate to three decimal places in the scientific notation.

Displaying Trailing zeros and plus sign

If we print the numbers 10.75, 25.00 and 15.50 using a field width of eight positions, with two digits precision, then the output will be as follows:

			1	0	.	7	5
						2	5
				1	5	.	5

The trailing zeros in the second and third items have been truncated. Certain situations, such as a list of prices of items or the salary statement of employees, require trailing zeros to be shown. The above output would look better if they are printed as follows:

10.75

25.00

15.50

The `setf()` can be used with the flag `ios :: showpoint` as a single argument to achieve this form of output. For example,

```
cout.setf(ios :: showpoint); //display trailing zeros
```

would cause `cout` to display trailing zeros and trailing decimal point.

Similarly, a plus sign can be printed before a positive number using the following statement:

```
cout.setf(ios :: showpos);
```

For example, the statements

```
cout.setf(ios :: showpoint);
```

```
cout.setf(ios :: showpos);
```

```
cout.precision(3);
```

```
cout.setf(ios :: fixed, ios :: floatfield);
```

```
cout.setf(ios :: internal, ios :: adjustfield);
```

```
cout.width(10);
```

```
cout<<275.5<<"\n";
```

will produce the following output:

+			2	7	5	.	5	0	0
---	--	--	---	---	---	---	---	---	---

The flags such as **showpoint** and **showpos** do not have any bitfields and are used as single arguments in **setf()**.

Formatting with flags in setf()

```
#include<iostream.h>
#include<cmath.h>
int main()
{
    cout.fill('*');
    cout.setf(ios :: left, ios :: adjustfield);
    cout.width(10);
    cout<<"VALUE":
    cout.setf(ios :: right, ios :: adjustfield);
    cout.width(15);
    cout<<"SQRT OF VALUE"<<"\n";
    cout.fill('.');
    cout.precision(4);
    cout.setf(ios :: showpoint);
    cout.setf(ios :: showpos);
    cout.setf(ios :: fixed, ios :: floatfield);
    for(int n=1; n<=10; n++)
    {
        cout.setf(ios :: internal, ios :: adjustfield);
        cout.width(5);
        cout<<n;
        cout.setf(ios :: right, ios :: adjustfield);
        cout.width(20);
        cout<<sqrt(n)<<"\n";
    }
    cout.setf(ios :: scientific, ios :: floatfield);
    cout<<"\nSQRT(100) = "<<sqrt(100)<<"\n";
    return 0;
}
```

The output of program would be

V	A	L	U	E	*	*	*	*	*	*	*	S	Q	R	T		O	F		V	A	L	U	E
+	.	.	.	1	+	1	.	0	0	0	0
+	.	.	.	2	+	1	.	4	1	4	2
+	.	.	.	3	+	1	.	7	3	2	1
+	.	.	.	4	+	2	.	0	0	0	0
+	.	.	.	5	+	2	.	2	3	6	1
+	.	.	.	6	+	2	.	4	4	9	5
+	.	.	.	7	+	2	.	6	4	5	8
+	.	.	.	8	+	2	.	8	2	8	4
+	.	.	.	9	+	3	.	0	0	0	0
+	.	.	1	0	+	3	-	1	6	2	3

SQRT(100) = +1.0000E+001

Working with files

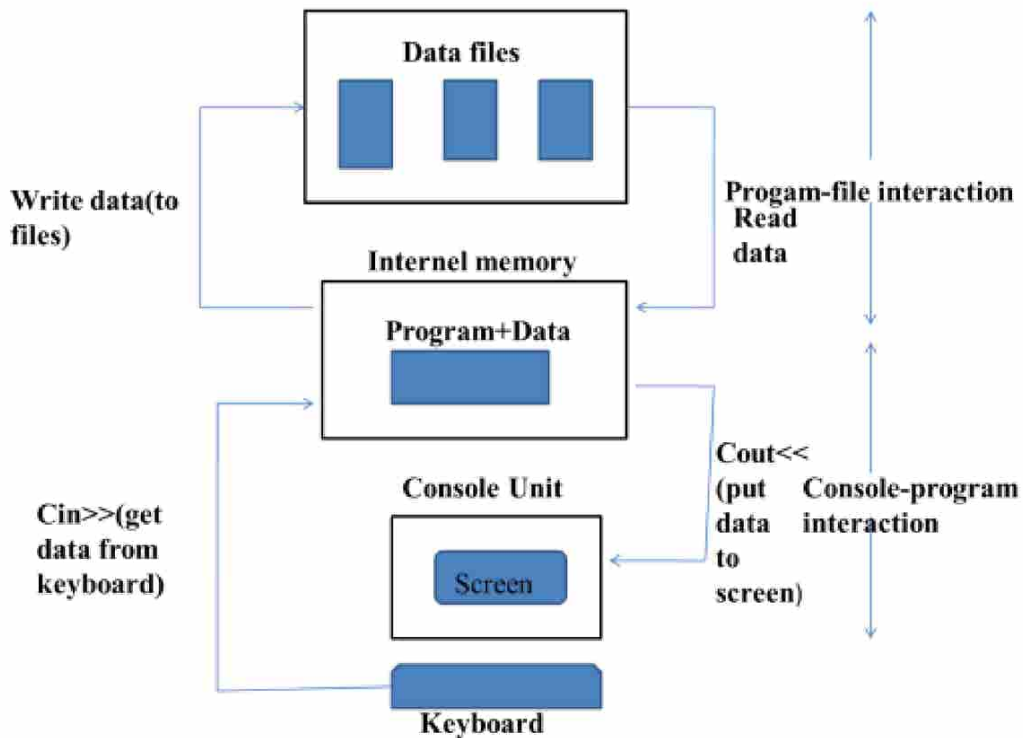
Introduction

The data is stored in some devices such as floppy disk or hard disk, using the concept of files. A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on these files.

A program typically involves either or both of the following kinds of data communication:

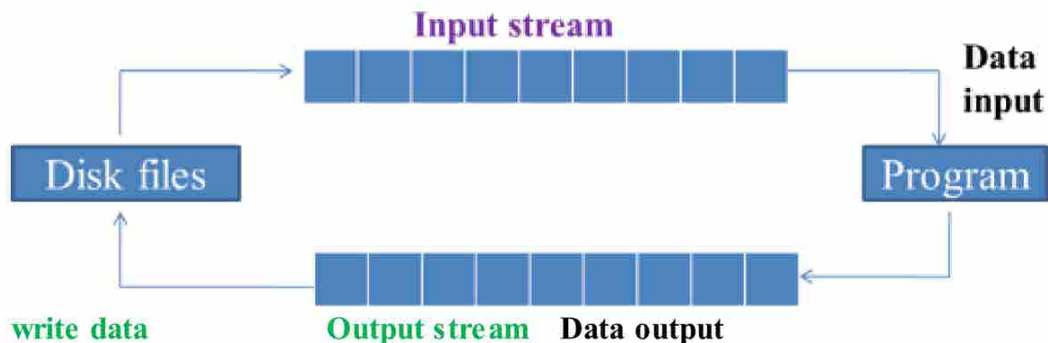
1. Data transfer between the console unit and the program
2. Data transfer between the program and a disk file.

External Memory



The I/O system of C++ handles file operations which are very much similar to the console input and output operations. It uses file streams as an interface between the **programs** and the **files**. The stream that **supplies data** to the program is known as **input stream** and the one that **receives data** from the program is known as **output stream**. The input stream extracts data from the file and the output stream inserts data to the file.

Read data

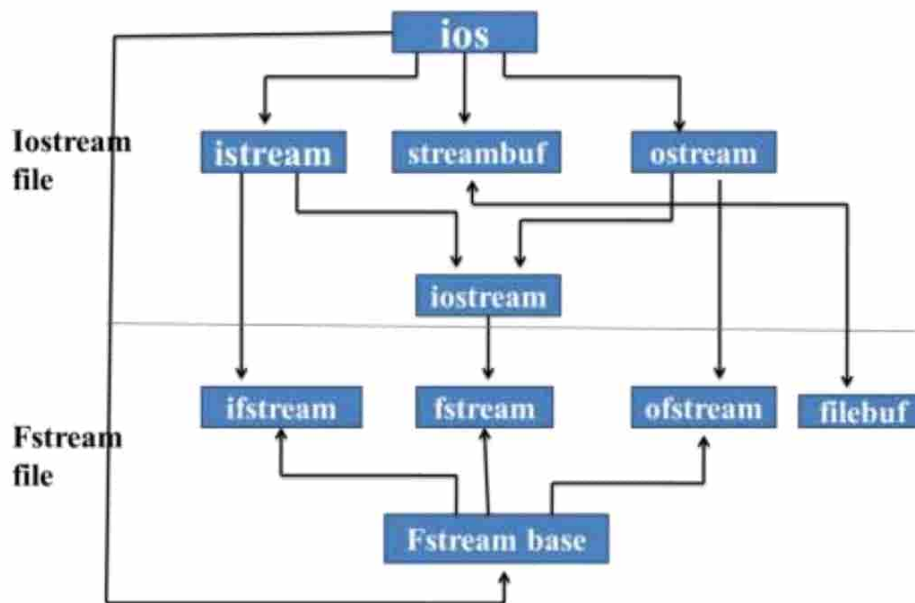


The input operation involves the creation of an **input stream** and linking it with the **program** and the **input file**. Similarly, the output operation involves establishing an **output stream** with the necessary links with the **program** and the **output file**.

Classes for file stream operations

The I/O stream of C++ contains a set of classes that define the file handling methods. These include **ifstream**, **ofstream** and **fstream**. These classes are derived from **fstreambase** and from the corresponding **istream** class as shown in Fig. These classes, designed to manage the disk files, are declared in **fstream** and include this file in any program that uses files.

Stream classes for file operations



Details of file stream classes

Class	Contents
fstreambase	Provides operations common to the file streams. Serves as a base for fstream , ifstream and ofstream class. Contains open() and close() functions
ifstream	Provides input operations. Contains open() with default input mode. Inherits the functions get() , getline() , read() , seekg() and tellg() functions from istream .
ofstream	Provides output operations. Contains open() with default output mode. Inherits the functions put() , seekp() and tellp() and write() functions from ostream .
fstream	Provides support for simultaneous input and output operations. Contains open() with default input mode. Inherits all the functions from istream and ostream classes through iostream .

Opening and closing a file

If we want to use disk file, we need to describe the following things about the file:

1. Suitable name for the file.
2. Data type and structure
3. Purpose
4. Opening method

The file name is a string of characters that make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with extension.

Input. Data	student
Test. Doc	salary
INVENT.ORY	OUTPUT

For opening a file, we must first create a file stream and then link it to the filename. A file stream can be defined using the classes `ifstream`, `ofstream`, and `fstream` that are contained in the header file `fstream`. The class to be used depends upon the purpose, that is, whether we want to read data from the file or write data to it. A file can be opened in two ways:

1. Using the constructor function of the class
2. Using the member function `open()` of the class

The first method is useful when we use only one file in the stream. The second method is used when we want to manage multiple files using one stream.

Opening files using constructor

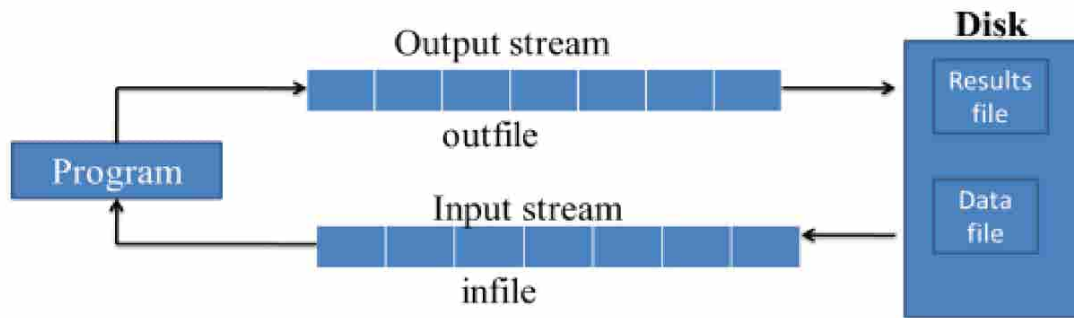
A constructor is used to initialize an object, while it is being created. Here, a filename is used to initialize the file stream object. This involves the following steps:

1. Create a file stream object to manage the stream using the appropriate class. That is, the class `ofstream` is used to create the output stream and the class `ifstream` to create the input stream.
2. Initialize the file object with the desired filename.

For example, the following statement opens a file named “results” for output:

```
ofstream outfile(“results”); // output only
```

This creates `outfile` as an `ofstream` object that manages the output stream. This object can be any valid C++ name such as `o_file`, `myfile` or `fout`. This statement also opens the file `results` and attaches it to the output stream `outfile`. This is illustrated in Fig.



Two file streams working on separate files

Similarly, the following statement declares infile as an ifstream object and attaches it to the file data for reading:

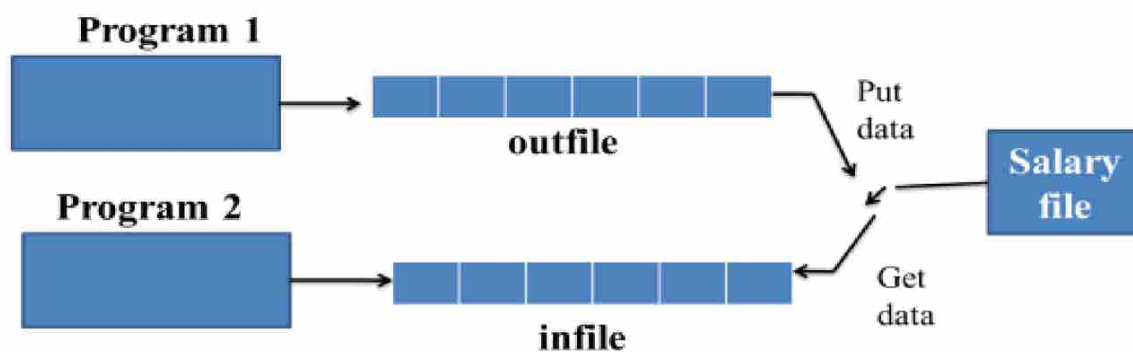
```
ifstream infile("data");    //input only
```

The program may contain statements like:

```
outfile<<"Total";
outfile<<sum;
infile>>number;
infile>>string;
```

We can also use the same file for both reading and writing data as shown in Fig. The Programs would contain the following statements:

Two file streams working on one file



Program 1

.....

.....

```
ofstream outfile("salary");    //creates outfile and
```

```
.....                          //connects "salary" to it
```

.....

Program 2

```
.....  
.....  
ifstream infile("salary"); // creates infile and  
..... // connects "salary" to it  
.....
```

The connection with a file is closed automatically when the stream object expires (when the program terminates). In the above statement, when the **program 1** is terminated, the **salary** file is disconnected from the **outfile** stream. Similar action takes place when the **program 2** terminates.

Instead of using two programs, one for writing data (output) and another for reading data (input), we can use a single program to do both the operations on a file. Example, ofstream outfile("salary");

```
.....  
    outfile.close(); //disconnect salary from  
    ifstream infile("salary"); //outfile and connect  
    ..... // to infile  
    .....  
infile.close(); //disconnect salary from infile
```

We created two file stream objects, **outfile** (to put data to the file) and **infile** (to get data from the file). The statement **outfile.close()**; disconnects the file **salary** from the output stream **outfile** and connecting the **salary** file to **infile** stream to read data.

Following program uses a single file for both writing and reading the data. First, it takes data from the keyboard and writes it to the file. After the writing is completed, the file is closed. The program again opens the same file, reads the information already written to it and displays the same on the screen.

// Creating files with constructor function

```
#include<iostream.h>  
#include<fstream.h>  
int main()  
{  
    ofstream outf("ITEM"); //connect ITEM file to outf  
    cout<<"Enter item name:";  
    char name[30];  
    cin>>name; //get name from keyboard and  
    Outf<<name<<"\n"; // write to file ITEM  
    cout<<" Enter item cost:";  
    float cost;  
    cin>>cost; //get cost from keyboard and  
    outf<<cost<<"\n"; //write to file ITEM  
    outf.close();  
    ifstream inf("ITEM"); //connect ITEM file to inf
```

```

inf>>name;           //read name from file ITEM
inf>>cost;            //read cost from file ITEM
cout<<"\n";
cout<<"Item name:"<<name<<"\n";
cout<<"Item cost:"<<cost<<"\n";
inf.close();          // Disconnect ITEM from inf
return 0;
}

```

Output of the program

```

Enter item name : CD-ROM
Enter item cost : 250
Item name : CD-ROM
Item cost : 250

```

Opening Files using open()

The function `open()` can be used to open multiple files that use the same stream object. For example, we may want to process a set of files sequentially. In such cases, we may create a single stream object and use it to open each file in turn.

File-stream-class stream-object;

Stream-object . open("file name");

Example

```

ofstream outfile;           // Create stream (for output)
outfile .Open("DATA1");     //connect stream to DATA1
.....
.....
outfile.close();             //Disconnect stream from DATA1
outfile.open("DATA2");       //connect stream to DATA2
.....
.....
outfile.close();             //Disconnect stream from DATA2
.....
.....

```

The above program segment opens two files in sequence for writing the data. The first file is closed before opening the second one.

// Creating files with open() function

```

#include<iostream.h>
#include<fstream.h>

```

```

int main()
{
    ofstream fout;    // create output stream
    fout.open("country"); //connect country to it
    fout<<"United States of America\n";
    fout<<"United Kingdom\n";
    fout<<"South Korea\n";
    fout.close();    // disconnect "country" and

    Fout.open("capital"); //connect "capital"
    fout<<"Washington\n";
    fout<<"London\n";
    fout<<"Seoul\n";
    fout.close();    // disconnect "capital"
    // Reading the files
    const int N = 80;    // size of line
    ifstream fin;    // create input stream
    fin.open("country"); //connect "country" to it
    cout<<"contents of country file\n";
    while(fin)    // check end-of-file
    {
        fin.getline(line, N);    //read a line
        cout<<line;    //display it

    Fin.close();    //disconnect "country" and
    fin.open("capital"); //connect "capital"
    cout<<"\ncontents of capital file \n";
    while(fin)
    {
        fin.getline(line, N);
        cout<<line;
    }
    fin.close();
    return 0;
}

contents of country file
United States of America
United Kingdom
South Korea
contents of capital file
Washington
London
Seoul

```

DEPARTMENT OF PHYSICS
GOVERNMENT ARTS AND SCIENCE COLLEGE, NAGERCOIL
Internal Test 1 (12.10.2020)

SMPH52 – Computer Programming in C++

Time : 1 hour

Total: 20 marks

Submit before 10.45am

Part A(2x1=2)

Answer all the Questions

1. The first character of ----- must be an alphabet or underscore.
2. The ----- is an entry control loop.

Part B(2x5=10)

Answer all the Questions

3. Write about Type casting and its types.
4. Discuss the declaration and initialisation of a strings in Arrays.

Part C(1x8=8)

Answer any one

5. (i) What are operators. (2)
(ii) Describe the Arithmetic, Relational and Logical operators in C++ language with **examples**.(6)

[OR]

6. Explain in detail the declaration and initialisation of one dimensional Array with **one example**.

DEPARTMENT OF PHYSICS
GOVERNMENT ARTS AND SCIENCE COLLEGE, NAGERCOIL
Internal Test 1 (27.10.2020)

SMPH52 – Computer Programming in C++

Time : 1 hour

Total: 20 marks

Submit before 10.45am

Part A(2x1=2)

Answer all the Questions

7. The purpose of math library function $\log(x)$ is -----.
8. To declare a reference variable or parameter, precede the variable's name with the --
-----,
.

Part B(2x5=10)

Answer all the Questions

9. Explain inline function with example.
10. Explain function overloading with example.

Part C(1x8=8)

Answer any one

11. With an example, explain the types of user defined functions in detail.

[OR]

12. (i). Write the difference between structure and class in C++.
- (ii). Explain in detail specifying a class in C++.

DEPARTMENT OF PHYSICS
GOVERNMENT ARTS AND SCIENCE COLLEGE, NAGERCOIL

Unit Test 1 (**17.09.2020**)

Submit before 11am

Time : 1 hr

Total: 20 marks

Computer Programming in C++

Part A(2x1=2)

1. Every C++ program must include _____

header file

(a) stdio.h (b) conio.h

(c) iostream.h (d) math.h

2. What will be the output of this program

```
#include <iostream>
int main()
{
    std::cout << "Welcome!";
    return 0;
}
```

Part B(2x5=10)

3. Explain the Hierarchy of Arithmetic Operators using One example.

4. Write about variables and constants in C++.

Part C(1x8=8)

Answer Any one

5. Explain the following control statements

a. Nested if

b. do – while loop

(OR)

6. Explain the structure of a simple C++ program and write a program to add two numbers with output.

1. Identifiers are -----
 - a) user defined name
 - b) keywords
 - c) predefined name
 - d) reserve words
2. OOPs give more importance to -----
 - (a) Class
 - (b) Object
 - (c) Data
 - (d) Algorithm
3. The ----- statement reads all types of data values
 - a)scanf()
 - b) printf()
 - c)puts()
 - d)abs()
4. Default arguments are useful in situation when some argument always have----- value
 - (a) same
 - (b)smaller
 - (c) higher
 - (d) different
5. Which of the following feature of C++ is operator overloading
 - (a) polymorphism
 - (b) inheritance
 - (c) datahiding
 - (d) encapsulation
6. A ----- is defined as a block of statements which are repeatedly executed for a certain number of times
 - a) if
 - b) switch
 - c) loop
 - d) break
- 7.A Derived class with only one base class called
 - (a)single inheritance
 - (b) multiple
 - (c) multilevel
 - (d) friend function
8. The stream that supplies data to the program is known as

(OR)

(b) Describe the usage of main () function in C++ program

18.(a) Describe the concept of nesting of Member function with example (OR)

(b) Interpret the usage of multiple constructors in a class

19. (a) Illustrate the concept of multilevel inheritance (OR)

(b) Briefly explain the term arithmetic operation in pointers

20. (a) Describe unformatted and formatted I/O operations (OR)

(b) Explain opening and closing a file.

(6 pages)

Reg. No. :

Code No. : 40561 E Sub. Code : SMPH 52

B.Sc. (CBCS) DEGREE EXAMINATION,
NOVEMBER 2019.

Fifth Semester

Physics — Main

COMPUTER PROGRAMMING IN C++

(For those who joined in July 2017 onwards)

Time : Three hours

Maximum : 75 marks

PART A — (10 × 1 = 10 marks)

Answer ALL questions.

Choose the correct answer :

1. The smallest individual units in a program are known as _____.
(a) System (b) Token
(c) Array (d) Bytes
2. ANSI C++ all character in a name are _____.
(a) Significant (b) Insignificant
(c) Identical (d) None

8. A class can inherit properties from more than one class which is known as _____ inheritance.
(a) Multiple (b) Single
(c) Multilevel (d) None
9. A stream is a sequence of _____.
(a) Unit (b) System
(c) Bytes (d) Token
10. To clear specified flags related function _____.
(a) Unsetf() (b) Istream
(c) Ostream (d) Main()

PART B — (5 × 5 = 25 marks)

Answer ALL questions, choosing either (a) or (b).

Each answer should not exceed 250 words.

11. (a) Describe the various user defined data types, such as structure and classes.

Or
(b) Discuss declaring variables.

3. In C++ the main() returns a value of type _____ to the operating system.
(a) Istream (b) Ostream
(c) int (d) None
4. Default arguments are useful in situation when some argument always have _____ value.
(a) same (b) smaller
(c) higher (d) different
5. Class function describe how the class function are _____.
(a) activated (b) increase
(c) implemented (d) decrease
6. A private function object cannot invoke using a _____ Operator.
(a) System (b) Dot
(c) Point (d) Program
7. The unary minus when applied to an object should change the _____ of each of its data item.
(a) Sign (b) Length
(c) Bytes (d) Character

Page 2 Code No. : 40561 E

12. (a) Explain function with no argument and no return values.

Or
(b) Demonstrate how function overloading is used in a C++ program.
13. (a) Explain the term of static class member.

Or
(b) Explain the parameterized constructor.
14. (a) Demonstrate how unary and binary operator are overloaded.

Or
(b) Explain the term virtual base class.
15. (a) Explain the overview of stream in C++.

Or
(b) Illustrate the use of manipulators for managing output.

Page 3 Code No. : 40561 E

Page 4 Code No. : 40561 E
[P.T.O.]

PART C — (5 × 8 = 40 marks)

Answer ALL questions, choosing either (a) or (b).

Each answer should not exceed 600 words.

16. (a) Outline the various types of expression used in a C++ program.

Or

- (b) Explain the various variables used in C++ program.

17. (a) Briefly explain term calling by reference and return by reference.

Or

- (b) Describe the usage of Main() function in a C++ program.

18. (a) Describe the concept of nesting of member function.

Or

- (b) Interpret the usage of multiple constructors in a class.

19. (a) Illustrate the concept of multilevel inheritance.

Or

- (b) Briefly explain the term arithmetic operation on pointer.

20. (a) Describe unformatted and formatted I/O operations.

Or

- (b) Illustrate the use of file pointers and their manipulators.

(6 pages)

Reg. No. :

Code No. : 41123 E

Sub. Code : JMPH 41

B.Sc. (CBCS) DEGREE EXAMINATION,
NOVEMBER 2018.

Fourth Semester

Physics — Main

COMPUTER PROGRAMMING IN C++

(For those who joined in July 2016 and afterwards)

Time : Three hours

Maximum : 75 marks

PART A — (10 × 1 = 10 marks)

Answer ALL questions.

Choose the correct answer :

1. _____ are primary run time entities in an object oriented programming.
 - (a) Classes
 - (b) Objects
 - (c) Variables
 - (d) Members

2. The _____ statement cause skipping of the statements till the end of a loop.
- (a) Stop (b) Change
(c) Continue (d) Run
3. In C++ manipulators are used to format the _____ display
- (a) Screen (b) Data
(c) Character (d) String
4. A Non-member function that can access the private data of class is known as
- (a) Friend function
(b) Static function
(c) Member function
(d) Library function
5. Templates are skeleton to _____.
- (a) Objects
(b) Constructors
(c) Classes
(d) Member variables

PART B — (5 × 5 = 25 marks)

Answer ALL questions, choosing either (a) or (b).

Each answer should not exceed 250 words.

11. (a) What are the datatypes in C++?

Or

- (b) Define C++ operator.

12. (a) Write a program to implement function overloading.

Or

- (b) Discuss the inline function.

13. (a) Briefly explain the objects and classes in C++.

Or

- (b) Discuss about dynamic constructors with example.

14. (a) Define polymorphism and virtual function.

Or

- (b) Explain Multiple inheritance with suitable example.

15. (a) Write a program to create a file of characters.

Or

- (b) Write short notes on: formatted console I/O Operation.

PART C — ($5 \times 8 = 40$ marks)

Answer ALL questions, choosing either (a) or (b).

Each answer should not exceed 600 words.

16. (a) Write a C++ program to find all possible roots of a Quadratic equation $Ax^2 + Bx + C = 0$.

Or

- (b) Explain the basic concepts of object oriented programming.

17. (a) Write a C++ program to find the area of circle, triangle using function overloading.

Or

- (b) Explain about friend functions with examples.

18. (a) Illustrate the rule for Copy constructors.

Or

- (b) What are nesting member functions? Explain with suitable program segments.

19. (a) Explain hierarchical Inheritance with a program

Or

(b) Write a C++ program to implement 'this' pointers.

20. (a) Explain various formatted I/O.

Or

(b) What are data files? Explain the functions used for different operations on data files in C++.
