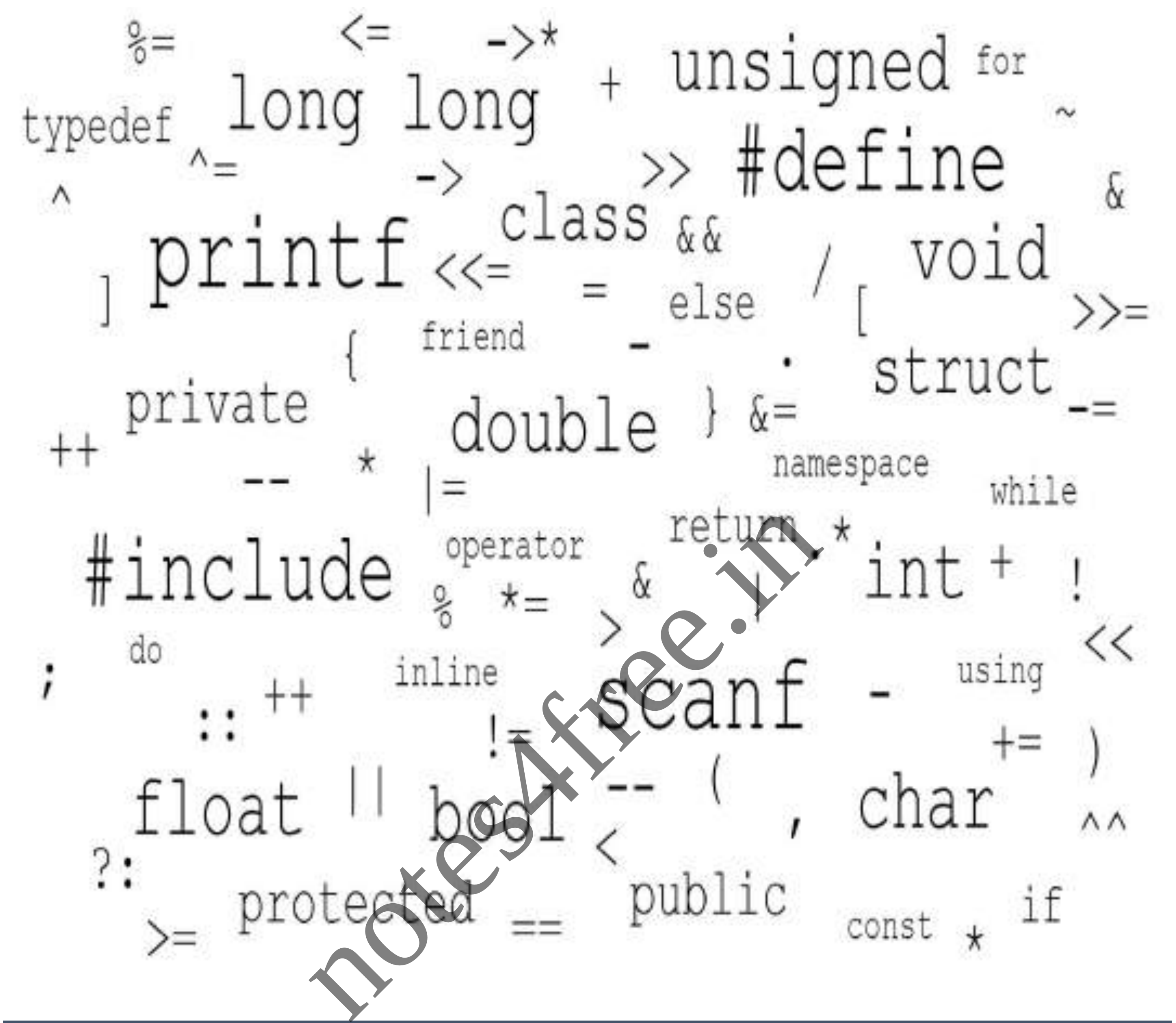


MODULE -1

BEGINNING WITH C++ AND ITS FEATURES

GANESH Y
Dept. of ECE RNSIT

MODULE -1
Beginning with C++ and its features

SYLLABUS

Beginning with C++ and its features: What is C++, Applications and structure of C++ program, Different Data types, Variables, Different Operators, expressions, operator overloading and control structures in C++ (Topics from Chapter-2,3 of Text1).

Differences between POP and OOP

	Procedure Oriented Programming	Object Oriented Programming
Divided Into	In POP, program is divided into small parts called **functions**.	In OOP, program is divided into parts called **objects**.
Importance	In POP, Importance is not given to **data** but to functions as well as **sequence** of actions to be done.	In OOP, Importance is given to the data rather than procedures or functions because it works as a **real world**.
Approach	POP follows **Top Down approach**.	OOP follows **Bottom Up approach**.
Access Specifiers	POP does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
Data Moving	In POP, Data can move freely from function to function in the system.	In OOP, objects can move and communicate with each other through member functions.
Expansion	To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Data Access	In POP, most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Data Hiding	POP does not have any proper way for hiding data so it is **less secure**.	OOP provides Data Hiding so provides **more security**.
Overloading	In POP, Overloading is not possible.	In OOP, overloading is possible in the form of Function Overloading and Operator Overloading.
Examples	Example of POP are: C, VB, FORTRAN, Pascal.	Example of OOP are: C++, JAVA, VB.NET, C#.NET.

Basic Concepts of Object-Oriented Programming

Objects:-

* *Objects* are the basic run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

* They may also represent user-defined data such as vectors, time and lists, Programming problem is analyzed in terms of objects and the nature of communication between them.

* Program objects should be chosen such that they match closely with the real-world objects. Objects take up space in the memory and have an associated address like a record in Pascal or a structure in C.

Classes:-

* We just mentioned th.at objects contain data, and code *to* manipulate that data. The entire set of data and code of an object can be made a user-defined data type with the help of a class.

* In fact, objects are variables of the type class. Once a class has been defined, we can create any number of objects belonging to that class. Each object is associated with the data of type class with which they are created.

* A class is thus a collection of objects of similar type. For example, mango, apple and orange are members of the class fruit. Classes are user-defined data types and behave like the built-in types of a programming language.

* The syntax used to create an object is no different than the syntax used to create an integer object in C. If fruit bas been defined as a class, then the statement

```
fruit mango;
```

will create an object **mango** belonging to the class **fruit.**

Data Encapsulation: -

* The wrapping up of data and functions into a single unit (called class) *is* known as ***encapsulation****.* Data encapsulation is the most striking feature of a class.

* The data is not accessible to the out-side world, and only those functions which are wrapped in the class can access it. These functions provide the interface between the object's data and the program. This insulation of the data from direct access by the program *is* called ***data hiding*** *or* ***information hiding.***

Data Abstraction:-

* *Abstraction* refers to the act of representing essential features without including the background details or explanations.

* Classes use the concept of abstraction and are defined as a list of abstract *attributes* such as size, weight and cost and *functions* to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created. The attributes are sometimes called *data numbers* because they hold information.

The functions that operate on these data are sometimes called ***methods or member functions.***

* Since the classes use the concept of data abstraction, they are known as *Abstract Data Types* (ADT).

Inheritance:-

* *Inheritance* is the process by which objects of one class acquire the properties of objects of another class**.** It supports the concept of ***hierarchical classification.***

* In OOP, the concept of inheritance provides the idea of *reusability.* This means that we can add additional features to an existing class without modifying it. This is possible by deriving a new class from the existing one. The new class will have the combined features of both the classes.

Polymorphism :-

* *Polymorphism* is another important OOP concept. Polymorphism a Greek term, means the ability to take more than one form.

* An operation may exhibit different behaviors in different instances. The behavior depends upon the types of data used in the operation. For example, consider the operation of addition. For two numbers, the operation will generate a sum. If the operands are strings. then the operation would produce a third string by concatenation. The process of making an operator to exhibit different behaviors in different instances is known as ***operator overloading***.

* Similarly Using a single function name to perform different types of tasks is known as ***function overloading***.

Benefits of OOP

• Through inheritance, we can eliminate redundant code and extend the use of existing classes.

• We can build programs from the standard working modules that communicate with one another, rather than having to start writing the code from scratch. This leads to saving *of* development time and higher productivity.

• The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program.

• It is possible to have multiple instances of an object to co-exist without any interference.

• It is possible to map objects in the problem domain to those in the program.

• It is easy to partition the work in a project based on objects.

• The data-centered design approach enables us to capture more details of a model in implementable form.

• Object-oriented systems can be easily upgraded from small to large systems.

• Message passing techniques for communication between objects makes the interface descriptions with external systems much simpler.

• Software complexity can be easily managed.

What is C++?

* **C++ is an object-oriented programming** language. It was developed by Bjarne Stroustrup at AT&T Bell Laboratories in USA.
* C++ is an extension of C with a major addition of the class construct feature of Simula67.
* Since the class was a major addition to the original C language, Stroustrup initially called the new language **'C with classes'.** However, later in 1983, the name was changed to C++. The idea of C++ comes from the C increment operator ++, thereby suggesting that C++ is an augmented (incremented) version of C.
* During the early 1990's the language underwent a number of improvements and changes. In November 1997, the ANSI/ISO standards committee standardized these changes and added several new features to the language specifications.
* C++ is a superset of C. Most of what we already know about C applies to C++ also. Therefore. almost all C programs are also C++ programs. However, there are a few minor differences that will prevent a C program to run under C++ compiler.
* The most important facilities that C++ adds on to C arc classes. inheritance, function overloading, and operator overloading. Those features enable creating of abstract data types, inherit properties &om existing data types and support polymorphism, thereby making C++ a truly object-oriented language.
* The addition of new features has transformed C from a language that currently facilitates top-down. structured design, to one that provides bottom-up, object oriented design.

Applications of C++

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

• Since C++ allows us to create hierarchy-related objects, we can build special object, oriented libraries which can be used later by many programmers.

• While C++ is able to map the real-world problem properly, the C part of C++ gives the language the ability to get close to the machine-level details.

• C++ programs are easily maintainable and expandable. When a new feature needs to be implemented, it is very easy to add to the existing structure of an object

A Simple C++ Program

Example of a C++ program that prints a string on the screen.

```
# include <iostream> // include header file
using namespace std;
int main ()
{
     cout<<"Hello world\n";   // C++ statement
     return 0;
}    // end of example
```

This simple program demonstrates several C++ features.

Program Features

Like C, the C++ program is a collection of functions. The above example contains only one function, `main()`. As usual, execution begins at `main()`.

Every C++ program must have a `main()`. C++ is a free-form language. With a few exceptions, the compiler ignores carriage returns and white spaces. Like C, the C++ statements terminate with semicolons.

Comments

```
// This is an example of
// C++ program to illustrate
// Some of its features
```

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

```
/* This is an example of
C++ program to illustrate
Some of its features */
```

the double slash comment cannot be used in the manner as shown below:

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

Output Operator

The only statement in above program is an output statement. The statement

```
cout <<"Hello world\n";
```

causes the string in quotation marks to be displayed on the screen. This statement introduces two new C++ features, cout and <<.

The identifier cout (pronounced as 'C out') is a predefined object that represents the standard output stream in C++. Here, the standard output stream represents the screen. It is also possible to redirect the output to other output devices.

The operator << is called the *insertion or put to* operator. It inserts (or sends) the contents of the variable on its right to the object on its left as shown in fig.1.

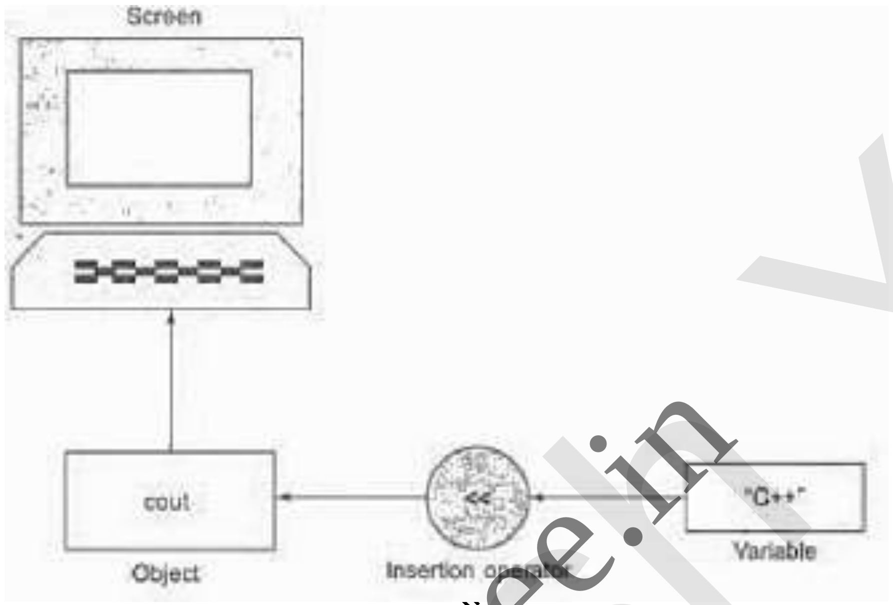


Fig.1 output using insertion operator

The iostream File

We have used the following #include directive in the program:

```
#include <iostream>
```

This directive causes the preprocessor to add the contents of the iostream file to the program. It contains declarations for the identifier cout and the operator <<.

Some old versions of C++ use a header file called iostream.h. This is one of the changes introduced by ANSI C++.

Namespace

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

```
using namespace std;
```

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

Return type of main( )

ln C++, main( ) returns an integer type value to the operating system. Therefore, every main( ) in *C++* should end with a return 0 statement; otherwise a warning or error might occur.

Since main() returns an integer type value, return type for main() is explicitly specified as int. Note that the default return type for all functions in C++ is **int**.

Variables

```
float number1, number2, sum, average;
```

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

Input Operator

The statement

```
cin >> number1;
```

is an input statement and causes the program to wait for the user to type in a number. The number keyed in is placed in the variable number1. The identifier **cin** (pronounced 'C in ') is a predefined object in C++ that corresponds to the standard input stream.

The operator >> is known as *extraction or get from* operator. It extracts (or takes) the value from the keyboard and assigns it to the variable on its right (Fig.2). This corresponds to the familiar scanf ( ) operation. Like <<, the operator >> can also be overloaded.

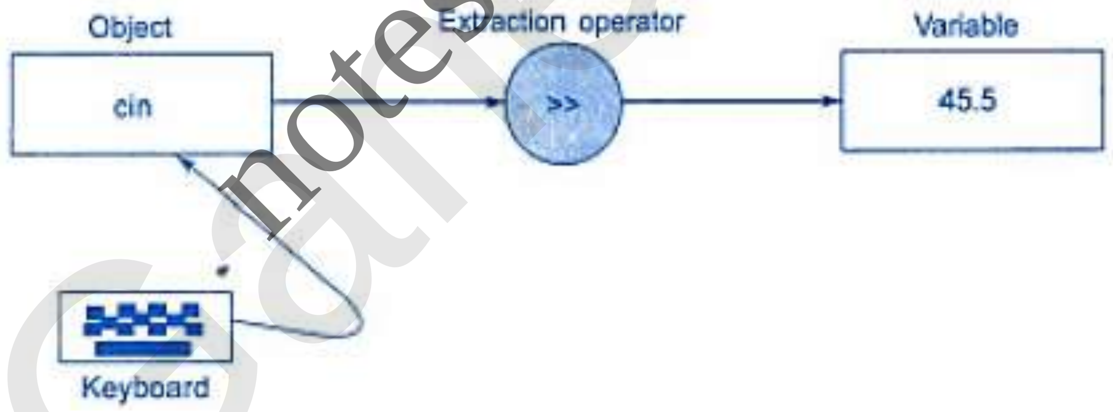


Fig.2 Input using extraction operator

Cascading of I/0 Operators

The statement

```
cout <<"Sum="<< sum<<"\n";
```

first sends the string "Sum=" to cout and then sends the value of sum. Finally, it sends the newline character so that the next output will be in the new line.

```
cout << "Sum=" << sum<<"\n"
     << "Average=" << average<<"\n";
```

This is one statement but provides two lines of output. If you want only one line of output, the statement will be:

```
                    cout << "Sum=" << sum<<","
                          << "Average=" << average<<"\n";
```

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

```
cin >>number1>> number2;
```

The values are assigned from left to right. That is, if we key in two values, say, 10 and 20, then 10 will be assigned to number1 and 20 to number2.

Structure of C++ Program

A typical C++ program would contain four sections as shown in Fig.3. These sections may be placed in separate code files and then compiled independently or jointly.

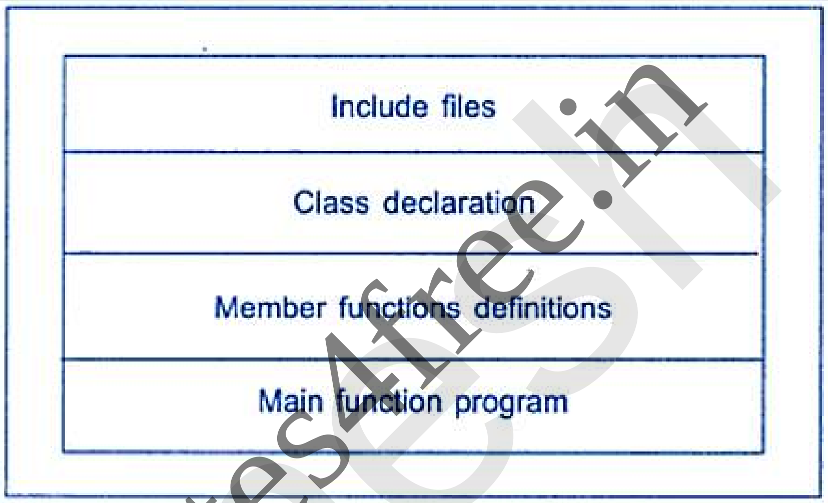


Fig 3 structure of a C++ program

It is a common practice to organize a program into three separate files:

* The class declarations are placed in a header file and the definitions of member functions go into another file.
* This approach enables the programmer to separate the abstract specification of the interface (class definition) from the implementation details (member functions definition).
* Finally, the main program that uses the class is placed in a third file which "includes"
* the previous two files as well as any other files required.

This approach is based on the concept of client-server model as shown in Fig. 4. The class definition including the member functions constitute the server that provides services to the main program known as client. The client uses the server through the public interface of the class.

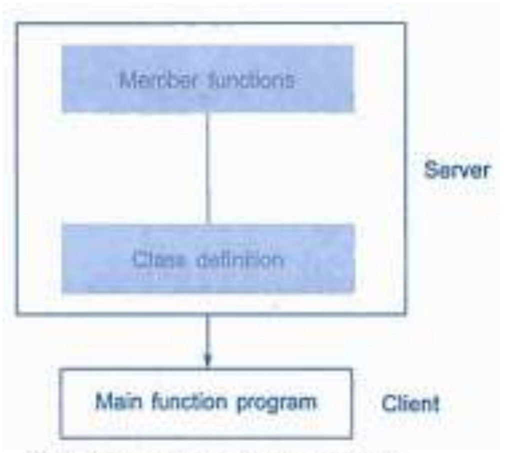


Fig. 4 The client-server model

Fallowing program shows the use of class in a C++ program.

```
#include <iostream>
using namespace std;
class person
{
	char name[30];
	int age;
	public:
	void getdata(void);
	void display(void);
} ;
void person :: getdata (void)
{
	cout << "Enter name: ";
	cin >> name;
	cout << "Enter age: ";
	cin >> age;
}
void person :: display(void)
{
	cout << "\n Name:  " << name;
	cout << "\n Age:   " << age;
}
int main()
{
	person p;
	p.getdata();
	p.display();
	return 0;
}
```

The program defines person as a new data of type class. The class person includes two basic data type items and two functions to operate on that data. These functions are called **member** functions. The main program uses person to declare variables of its type. As pointed out earlier, class variables are known as **objects**. Here, **p** is an object of type **person**.

Tokens

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

- Keywords
- Identifiers
- Constants
- Strings
- Operators

Keywords

The keywords implement specific C++ language features. They are explicitly reserved identifiers and cannot be used as names for the program variables or other user-defined program elements. Table. 1 gives the complete set of C++ keywords.

Table. 1 C++ keywords

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

Identifiers

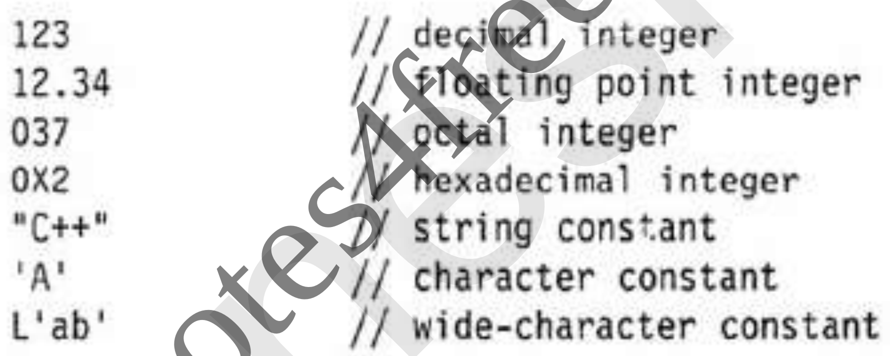
Identifiers refer to the names of variables, functions, arrays, classes, etc. created by the programmer. They are the fundamental requirement of any language. Each language has its own rules for naming these identifiers. The following rules are common to both C and C++:

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

A major difference between C and C++ is the limit on the length of a name. While ANSI C recognizes only the first 32 characters in a name, ANSI C++ places no limit on its length and, therefore, all the characters in a name are significant.

Constants

Constants refer to fixed values that do not change during the execution of a program.



```
123               // decimal integer
12.34             // floating point integer
037               // octal integer
0X2               // hexadecimal integer
"C++"             // string constant
'A'               // character constant
L'ab'             // wide-character constant
```

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

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

Strings

C++ supports two types of string representation - the C-style character string and the string class type introduced with Standard C++.

Basic Data types

Data types in C++ can be classified under various categories *as* shown in Fig. 5.

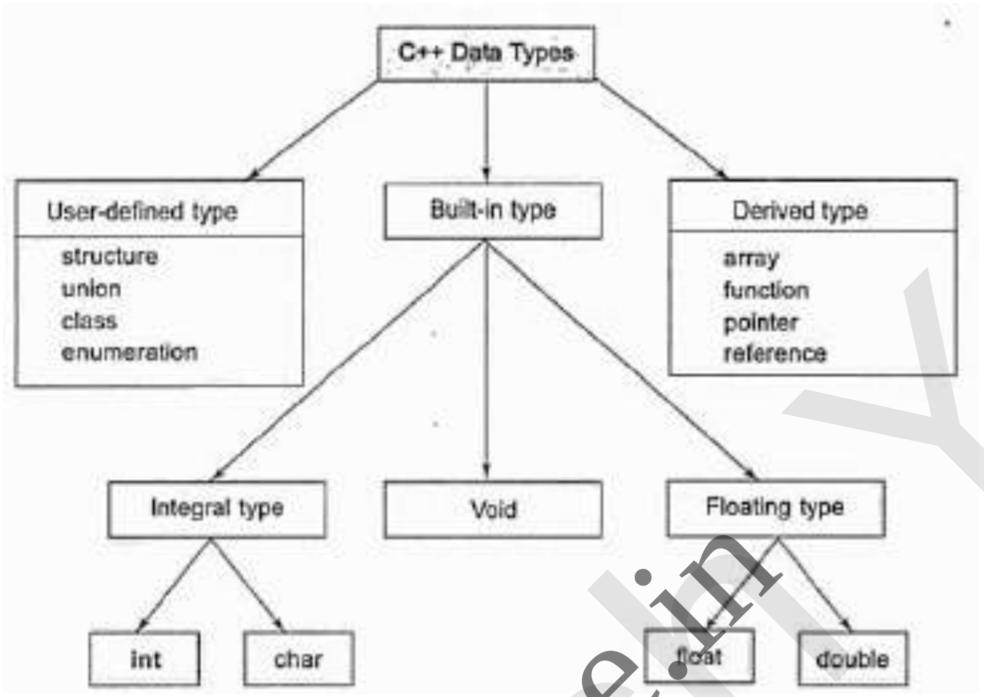


Fig. 5 C++ data types

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

Table 2: size and ranges of C++ basic data types (for 16-bit word machine)

Type	*Bytes*	*Range*
char	1	–128 to 127
unsigned char	1	0 to 255
signed char	1	– 128 to 127
int	2	– 32768 to 32767
unsigned int	2	0 to 65535
signed int	2	– 31768 to 32767
short int	2	– 31768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	–32768 to 32767
long int	4	–2147483648 to 2147483647
signed long int	4	–2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E–38 to 3.4E+38
double	8	1.7E–308 to 1.7E+308
long double	10	3.4E–4932 to 1.1E+4932

The fallowing explanation of void data type can be understood properly after discussion of pointers in module 4

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

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

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

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

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

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

are valid statements. But, the statement,

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

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

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

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

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

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

User-Defined Data Types

Structures

Standalone variables of primitive types are not sufficient enough to handle real world problems. It is often required to group logically related data items together. While arrays are used to group together similar type data elements, structures are used for grouping together elements with dissimilar types.

The **general format** of a structure definition is as follows:

```
struct   name
{
     Variable name1;
     Variable name2;
     . . . . . . . .
     . . . . . . . .
};
```

Consider an example of a book, which has several attributes such as title, number of pages, price etc. we can realize a book using structures as shown below:

```
struct   book
{
     char title[25];
     char author[25];
     int pages;
     float price;
};
struct book   book1, book2, book3;
```

here book1, book2 and book3 are declared as variables of the user-defined type book. We can access the member elements by dot(.) operator as

```
book1.pages=300;
book2.price=275.75;
```

Unions

Unions are conceptually similar to structures as they allow us to group together dissimilar type elements inside a single unit.

```
union   book
{
     char title[25];
     char author[25];
     int pages;
     float price;
};
```

But there are significant differences between structures and unions as far as their implementation is concerned.

The size of a structure type is equal to the sum of the sizes of individual member types. However, the size of a union is equal to the size of its largest member element.

Table 3: Differences between structures and unions

Structures	Unions
1.The keyword struct is used to define a structure	1. The keyword union is used to define a union.
2. When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is greater than or equal to the sum of sizes of its members. The smaller members may end with unused slack bytes.	2. When a variable is associated with a union, the compiler allocates the memory by considering the size of the largest memory. So, size of union is equal to the size of largest member.
3. Each member within a structure is assigned unique storage area of location.	3. Memory allocated is shared by individual members of union.
4. The address of each member will be in ascending order This indicates that memory for each member will start at different offset values.	4. The address is same for all the members of a union. This indicates that every member begins at the same offset value.
5 Altering the value of a member will not affect other members of the structure.	5. Altering the value of any of the member will alter other member values.
6. Individual member can be accessed at a time	6. Only one member can be accessed at a time.

Class

C++ also permits us to define another user-defined data type known as class which can be used, just like any other basic data type, to declare variables. The class variables are known as objects, which are the central focus or object-oriented programming.

Enumerated Data Type

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

Example:

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

In C++, the tag names **shape, colour, and position** become new type names. By using these tag names, we can declare new variables.

```
colour background = blue;    // allowed
colour background = 7;       // Error 1n C++
colour background = (colour) 7; //OK
```

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

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

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

```
enum colour {red, blue=4, green=6};
enum colour {red =5, blue, green};
```

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

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

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

```
int switch1 = off;
int switch2 = on;
```

Derived Data Types

Arrays

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

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

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

```
char string[4] = "xyz"; // OK for C++
```

Functions

Functions have undergone major changes in C++. While some of these changes are simple, others require a new way of thinking when organizing our programs. Many of these modifications and improvements were driven by the requirements of the object-oriented concept of C++.

Pointers

Pointers are declared and initialized as in C. Examples:

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

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

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

We cannot modify the address that **ptr1** is initialized to.

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

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

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

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

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

Symbolic Constants

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

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

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

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

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

```
const size =10;
//means
const int size =10;
```

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

The scoping of const values differs. A const in C++ defaults to the internal linkage and therefore it is local to the file where it is declared. In ANSI C, const values are global in nature.

They are visible outside the file in which they are declared. However, they can be made local by declaring them as **static** .

To give a const value an external linkage so that it can be referenced from another file. we must explicitly define it as an **extern** in C++. Example:

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

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

```
enum {X, Y, Z};
```

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

```
const X=0;
const Y=1;
const Z=2;
```

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

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

Declaration of Variables

We know that, in C, all variables must be declared before they are used in executable statements. This is true with C++ as well.

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

Dynamic Initialization of Variables

C++ permits initialization of the variables at run time. This is referred to as ***dynamic initialization***, In C++, a variable can be initialized at run time using expressions at the place of declaration.

For example

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

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

Reference Variables

C++ introduces a new kind of variable known as the *reference* variable. A reference variable provides an *alias* (alternative name) for a previously defined variable.

A reference variable is created as follows:

```
     data_type & reference_name = variable_name;
```

For example, if we make the variable **sum** a reference to the variable **total**, then sum and total can be used interchangeably to represent that variable.

```
float total=100;
float & sum= total;

cout « total;
and
cout << sum;
```

both print the value 100. The statement

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

will change the value of both total and sum to 110. Reference variables are used as function arguments, which will be discussed in call by reference method.

Where Variables Are Declared

Variables will be declared in three basic places: inside functions, in the definition of function parameters, and outside of all functions. These are local variables, formal parameters, and global variables.

Local Variables

- Variables that are declared inside a function are called *local variables*. In some C/C++ literature, these variables are referred to as ***automatic*** variables.
- Local variables exist only while the block of code in which they are declared is executing. That is, a local variable is created upon entry into its block and destroyed upon exit.

For example, consider the following two functions:

```
void func1(void)
{
     int x;
     x = 10;
}
void func2(void)
{
     int x;
     x = -199;
}
```

Formal Parameters

- If a function is to use arguments, it must declare variables that will accept the values of the arguments. These variables are called the ***formal parameters*** of the function.
- They behave like any other local variables inside the function. As shown in the following program fragment, *their declarations occur after the function name and inside parentheses:*

```
int fun1( char c)
{
     c='a';
     return 0;
}
```

Global Variables

- Unlike local variables, *global variables* are known throughout the program and may be used by any piece of code. Also, they will hold their value throughout the program's execution.

* We can create global variables by declaring them outside of any function. Any expression may access them, regardless of what block of code that expression is in.

```
#include <stdio.h>
int count;    /* count is global */
void func1(void);
int main(void)
{
     count = 100;
     func1();
     return 0;
}
void func1(void)
{
     int temp;
     temp = count;
     cout <<"count is"<<count; /* will print 100 */
}
```

Storage classes

There are four storage class specifiers supported by C++:

extern
auto
static
register
mutable

These specifiers tell the compiler how to store the subsequent variable. The general form of a declaration that uses one is shown here.

```
storage_specifier type var_name;
```

extern

Because C/C++ allows separate modules of a large program to be separately compiled and linked together, there must be some way of telling all the files about the global variables required by the program. Although C technically allows you to define a global variable more than once, it is not good practice (and may cause problems when linking).

More importantly, in C++, you may define a global variable *only once* and inform all files in program about these variables.

File One

```
int x, y;
char ch;
int main(void)
{
    /* ... */
}
void func1(void)
{
    x = 123;
}
```

File Two

```
extern int x, y;
extern char ch;
void func22(void)
{
    x=y/10;
}
void func23(void)
{
    y=10;
}
```

Automatic

Automatic storage class assigns a variable to its default storage type. *auto* keyword is used to declare automatic variables.

However, if a variable is declared without any keyword inside a function, it is automatic by default. This variable is **visible** only within the function it is declared and its **lifetime** is same as the lifetime of the function as well. Once the execution of function is finished, the variable is destroyed.

Syntax of Automatic Storage Class Declaration

```
datatype var_name1 [= value];
```

or

```
auto datatype var_name1 [= value];
```

Example of Automatic Storage Class

```
auto int x;
float y = 5.67;
```

Static

Static storage class ensures a variable has the **visibility** mode of a local variable but **lifetime** of an external variable. It can be used only within the function where it is declared but destroyed only after the program execution has finished.

When a function is called, the variable defined as static inside the function retains its previous value and operates on it. This is mostly used to save values in a recursive function.

For example,

```
static int x = 101;
static float sum;
```

Register

Register storage assigns a variable's storage in the CPU registers rather than primary memory. It has its lifetime and visibility same as automatic variable.

The purpose of creating register variable is to increase access speed and makes program run faster. If there is no space available in register, these variables are stored in main memory and act similar to variables of automatic storage class. So only those variables which requires fast access should be made register.

For example,

```
register int id;
register char a;
```

Example of Storage Class

```
//C++ program to create automatic, global, static and register
variables.
    #include<iostream>
    using namespace std;
    int g;    //global variable, initially holds 0

    void test_function()
    {
        static int s;    //static variable, initially holds 0
        register int r;    //register variable
        r=5;
        s=s+r*2;
        cout<<"Inside test_function"<<endl;
        cout<<"g = "<< g <<endl;
        cout<<"s = "<< s <<endl;
        cout<<"r = "<< r <<endl;
    }

    int main()
    {
        int a;    //automatic variable
        g=25;
        a=17;
        test_function();
        cout<<"Inside main"<<endl;
        cout<<"a = "<<a<<endl;
        cout<<"g = "<<g<<endl;
        test_function();
        return 0;
    }
```

In the above program, *g* is a global variable, *s* is static, *r* is register and *a* is automatic variable.

We have defined two function, first is ***main()*** and another is ***test_function().***

Since *g* is global variable, it can be used in both function. Variables *r* and *s* are declared inside ***test_function()*** so can only be used inside that function.

However, ***s*** being static isn't destroyed until the program ends. When *test_function()* is called for the first time, *r* is initialized to 5 and the value of *s* is 10 which is calculated from the statement,

```
s=s+r*2;
```

After the termination of *test_function()*, *r* is destroyed but *s* still holds 10. When it is called second time, *r* is created and initialized to 5 again.

Now, the value of *s* becomes 20 since *s* initially held 10. Variable *a* is declared inside *main()* and can only be used inside *main()*.

Output

```
Inside test_function
g = 25
s = 10
r = 5
Inside main
a = 17
g = 25
Inside test_function
g = 25
s = 20
r = 5
```

Mutable

In C++, a class object can be kept constant using keyword *const*. This doesn't allow the data members of the class object to be modified during program execution. But, there are cases when some data members of this constant object must be changed.

For example, during a bank transfer, a money transaction has to be locked such that no information could be changed but even then, its state has to be changed from - ***started*** to ***processing*** to ***completed.*** In those cases, we can make these variables modifiable using a **mutable** storage class.

Syntax for Mutable Storage Class Declaration

```
    mutable datatype var_name1;
```

For example,

```
mutable int x;
mutable char y;
```

Example of Mutable Storage Class

```
// C++ program to create mutable variable.
     #include<iostream>
     using namespace std;

     class test
     {
         mutable int a;
         int b;
         public:
             test(int x,int y)
             {
                 a=x;
                 b=y;
             }
             void square_a() const
             {
                 a=a*a;
             }
             void display() const
             {
                 cout<<"a = "<<a<<endl;
                 cout<<"b = "<<b<<endl;
             }
     };

     int main()
     {
         const test x(2,3);
         cout<<"Initial value"<<endl;
         x.display();
         x.square_a();
         cout<<"Final value"<<endl;
         x.display();
         return 0;
     }
```

A class *test* is defined in the program. It consists of a mutable data member *a*. A constant object *x* of class test is created and the value of data members are initialized using user-defined constructor.

Since, *b* is a normal data member, its value can't be changed after initialization. However *a* being mutable, its value can be changed which is done by invoking *square_a()* method. *display()* method is used to display the value the data members.

Output

```
Initial value          Final value
a = 2                  a = 4
b = 3                  b = 3
```

Storage Class	Keyword	Lifetime	Visibility	Initial Value	Storage	Purpose
Automatic	auto	Function Block	Local	Garbage	Stack segment	Local variables used by a single function
External	extern	Whole Program	Global	Zero	Data segment	Global variables used throughout the program
Static	static	Whole Program	Local	Zero	Data segment	Local variables retaining their values throughout the program
Register	register	Function Block	Local	Garbage	CPU registers	Variables using CPU for storage purpose
Mutable	mutable	Class	Local	Garbage	Depends on the scope of class	

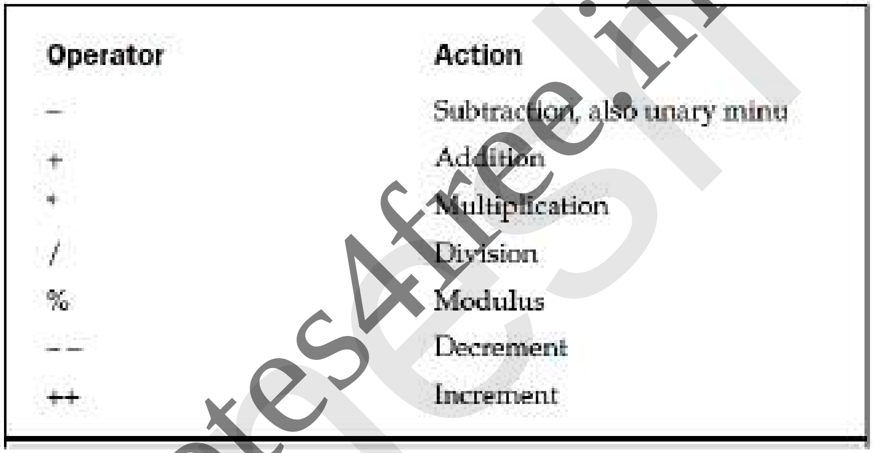
Extra info (not in syllabus)

Operators

Operators are the symbols which tell the computer to execute certain mathematical or logical operations. A mathematical or logical expression is generally formed with the help of an operator. C ++ programming offers a number of operators which are classified into different categories viz.

1. Arithmetic operators
2. Relational operators
3. Logical operators
4. Assignment operators
5. Bitwise operators
6. Special operators

1. Arithmetic Operators



Operator	Action
–	Subtraction, also unary minu
+	Addition
*	Multiplication
/	Division
%	Modulus
– –	Decrement
++	Increment

Note: '%' cannot be used on floating data type.

C programming allows the use of **++** and **–** operators which are increment and decrement operators respectively. Both the increment and decrement operators are unary operators. The increment operator ++ adds 1 to the operand and the decrement operator – subtracts 1 from the operand. The general syntax of these operators are:

Increment Operator: *m++ or ++m*;

Decrement Operator: *m--or --m*;

In the example above, *m++* simply means *m=m+1;* and *m--* simply means *m=m-1;* Increment and decrement operators are mostly used in for and while loops.

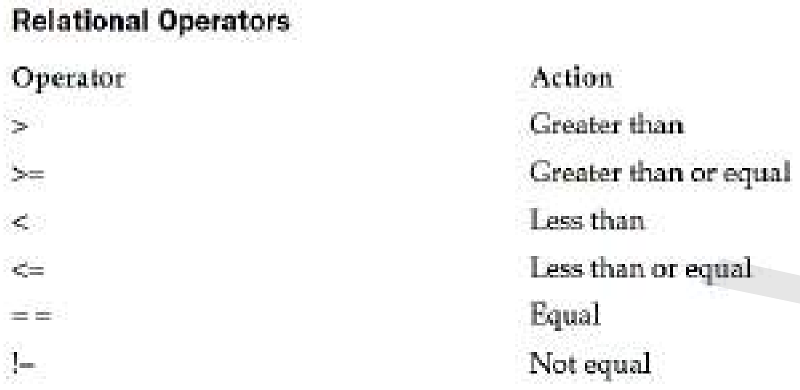
++m and *m++* performs the same operation when they form statements independently but they function differently when they are used in right hand side of an expression.

++m is known as prefix operator and *m++* is known as postfix operator. A prefix operator firstly adds 1 to the operand and then the result is assigned to the variable on

the left whereas a postfix operator firstly assigns value to the variable on the left and then increases the operand by 1. Same is in the case of decrement operator.

2.Relational Operators

Relational operators are used when we have to make comparisons. C programming offers 6 relational operators.

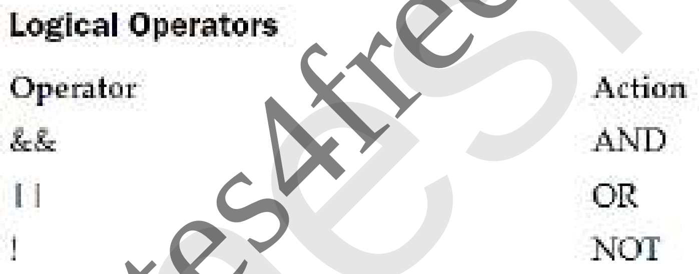


Relational Operators

Operator	Action
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
= =	Equal
!-	Not equal

3. Logical Operators

Logical operators are used when more than one conditions are to be tested and based on that result, decisions have to be made. C programming offers three logical operators. They are:



Logical Operators

Operator	Action
&&	AND
||	OR
!	NOT

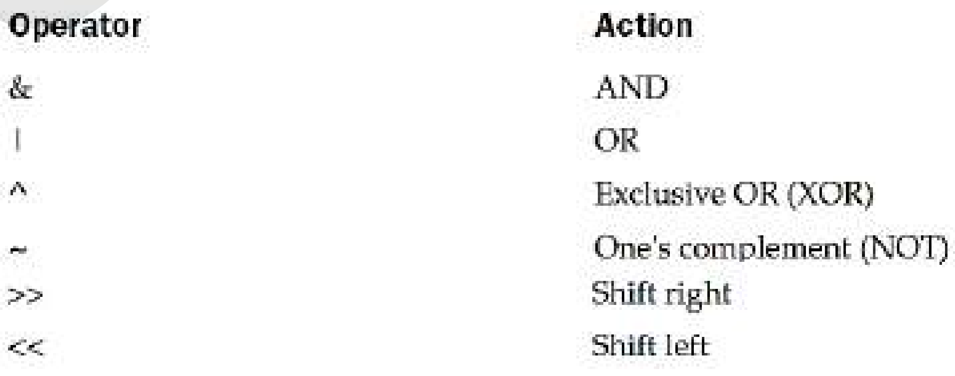
4. Assignment Operators

Assignment operators are used to assign result of an expression to a variable. '=' is the assignment operator in C. Furthermore, C also allows the use of shorthand assignment operators. Shorthand operators take the form:

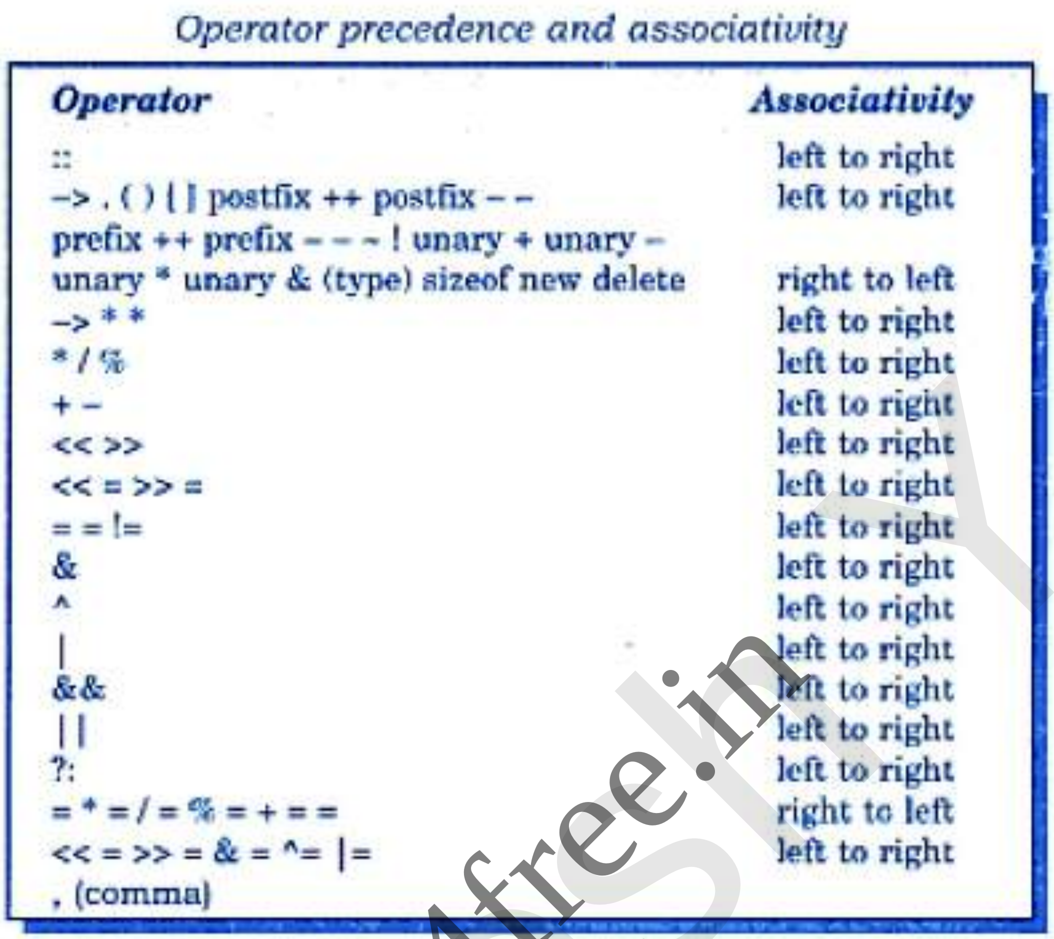
```
var op = exp;
```

5. Bitwise Operator

In C programming, bitwise operators are used for testing the bits or shifting them left or right. The bitwise operators available in C are:



Operator	Action
&	AND
|	OR
^	Exclusive OR (XOR)
~	One's complement (NOT)
>>	Shift right
<<	Shift left



Operator precedence and associativity

Operator	Associativity
::	left to right
–> . ( ) [ ] postfix ++ postfix – –	left to right
prefix ++ prefix – – ~ ! unary + unary – unary * unary & (type) sizeof new delete	right to left
–> * *	left to right
* / %	left to right
+ –	left to right
<< >>	left to right
<< = >> =	left to right
= = !=	left to right
&	left to right
^	left to right
|	left to right
&&	left to right
||	left to right
?:	left to right
= * = / = % = + = =	right to left
<< = >> = & = ^= |=	left to right
, (comma)	

7. Special operators:-

The ? Operator

C/C++ contains a very powerful and convenient operator that replaces certain statements of the if-then-else form. The ternary operator **?** takes the general form

```
Exp1 ? Exp2 : Exp3;
```

where *Exp1*, *Exp2*, and *Exp3* are expressions.

The **?** operator works like this: *Exp1* is evaluated. If it is true, *Exp2* is evaluated and becomes the value of the expression. If *Exp1* is false, *Exp3* is evaluated and its value becomes the value of the expression. For example, in

```
x = 10;
y = x>9 ? 100 : 200;
```

y is assigned the value 100. If **x** had been less than 9, **y** would have received the value 200. The same code written using the **if-else** statement is

```
x = 10;
if(x>9) y = 100;
else y = 200;
```

<u>The & and * Pointer Operators</u> are discussed in module 4

<u>The Compile-Time Operator *sizeof*</u>

sizeof is a unary compile-time operator that returns the length, in bytes, of the variable or parenthesized type-specifier that it precedes. For example, assuming that integers are 4 bytes and doubles are 8 bytes,

```
double f;
printf("%d", sizeof (f));
printf("%d", sizeof(int));
```

<u>The Comma Operator</u>

The comma operator strings together several expressions. The left side of the comma operator is always evaluated as **void**. This means that the expression on the right side becomes the value of the total comma-separated expression. For example,

```
x = (y=3, y+1);
```

first assigns **y** the value 3 and then assigns **x** the value 4. The parentheses are necessary because the comma operator has a lower precedence than the assignment operator.

<u>The Dot (.) and Arrow (–>) Operators</u>

In C, the **.** (dot) and the **–>**(arrow) operators access individual elements of structures and unions. In C++, the dot and arrow operators are also used to access the members of a class.

For example,

```
struct employee
{
     char name[80];
     int age;
     float wage;
} emp;
struct employee *p = &emp; /* address of emp into p */
```

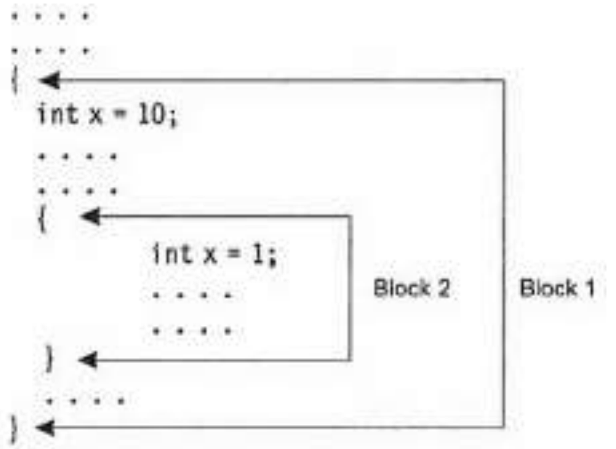
you would write the following code to assign the value 123.23 to the **wage** member of structure variable **emp**:

```
emp.wage = 123.23;
```

However, the same assignment using a pointer to **emp** would be

```
p->wage = 123.23;
```

Scope Resolution Operator



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

```
:: variable-name
```

```
#include <iostream>
using namespace std;
int m = 10; // global m
int main()
{
     int m = 20; // m redeclared, local to main
     {
          int k = m;
          int m = 30; // m declared again
             // local to inner block
          cout << "we are in inner block \n";
          cout << "k =" << k << "\n";
          cout << "m =" << m << "\n";
          cout << "::m =" << ::m << "\n";
     }
     cout << "\n We are in outer block \n";
     cout << "m = 11 "<< m << "\n";
     cout << "::m =" << ::m << "\n";
     return 0;
}
```

Output

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

Memory Management Operators

* C uses **`malloc()`** and **`calloc()`** functions to allocate memory dynamically at run time. Similarly, it uses the function **`free()`** to free dynamically allocated memory.
* Although C++ supports these functions, it also defines two unary operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way. Since these operators manipulate memory on the free store, they are also known as ***free store*** operators.

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

1. It automatically computes the size of the data object. We need not use the operator **`sizeof`.**

2. It automatically returns the correct pointer type, so that there is no need to use a type cast.

3. It is possible to initialize the object while creating the memory space.

4. Like any other operator, **new** and **delete** can be overloaded.

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

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

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

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

```
p = new int;        int *p = new int;          Subsequently, the statements
q = new float;      float *q = new float ;     *p = 25;
                                               *q = 7.5;
```

assign 25 to the newly created **int** object and 7.5 to the **float** object.
We can also initialize the allocated memory using the **new** operator. This is done as follows:

```
pointer-variable = new data-type(value);
```

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

similarly, memory for array data type can be allocated as

```
pointer-variable = new data- type[size];
```

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

creates a memory space for an array of 10 integers. **p[0]** will refer to the first element, **p[1]** to the second element, and so on.

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

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

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

```
delete pointer-variable;
```

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

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

```
delete [size] pointer-variable;
```

Recent versions of C++ do not require the size to be specified. For example,

```
delete [ ] p;
```

will delete the entire array pointed to by **p.**

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

```
..........
..........
p = new int;
if(!p)
{
     cout << "allocation failed \n";
}
..........
..........
```

Member Dereferencing Operators

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

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

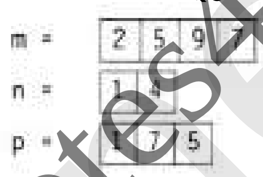
Manipulators

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

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

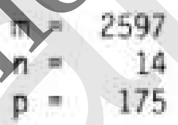
```
cout <<"m = "<< m << endl
     << "n = " << n << endl
     << "p = " << p << endl;
```

If we assume the values of the variables as 2597, 14, and 175 respectively, the output will appear as follows:



```
m = 2597
n = 14
p = 175
```

It should rather appear as under:



```
m = 2597
n =   14
p =  175
```

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

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

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

		3	4	5

```
//Use of manipulators
#include <iostream>
#include <iomanip> // for setw
using namespace std;
int main()
{
     long pop1=2425785, pop2=47, pop3=9761;
     cout << setw(8) <<"LOCATION"<< setw(12)<<"POPULATION"<< endl
           << setw(8) <<"Portcity"<< setw(12) << pop1 << endl
           << setw(8) <<"Hightown"<< setw(12) << pop2 << endl
           << setw(8) <<"Lowville"<< setw(12) << pop3 << endl;
     return 0;
}
```

Type Cast Operator

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

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

```
(type-name) expression // C notation
type-name (expression) // C++ notation
Examples:
average = sum/(float)i; // C notation
average = sum/float(i); // C++ notation
```

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

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

In such cases we must use C type notation.

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

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

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

Expressions and their types

- Constant Expressions
- Integral Expressions
- Float Expressions
- Pointer Expressions
- Relational expressions
- Logical Expressions
- Bitwise Expressions

Constant Expressions

Constant Expressions consists of only constant values

```
15
12+1/2.0
'x'
```

Integral Expressions

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

```
m
m = n - 5
m = 'x'
5 + int(2.0)
```

where m and n are integer variables.

Float Expressions

Float Expressions are those which, after all conversions, produce floating-point results.

Examples:

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

where x and y are floating point variables.

Pointer Expressions

Pointer Expressions produce address values. Examples:

```
&m
ptr
ptr + 1
"xyt"
```

where m is a variable and ptr is a pointer.

Relational Expressions

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

```
x <= y
a+b ==c+d
m+n > 100
```

Relational expressions are also known as Boolean expressions.

Logical Expressions

Logical Expressions combine two or more relational expressions and produces bool type results. Examples:

```
a>b  &&  x== 10
x==10 || y==5
```

Bitwise Expressions

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

```
x << 3 // Shift three bit position to left
y >> 1 // Shift one bit position to right
```

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

Special Assignment Expressions

Chained Assignment

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

First 10 is assigned to **y** and then to **x**.

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

```
float a=b = 12.34; // is illegal.
This may be written as
float a=12.34, b=12.34; // correct
```

Embedded Assignment

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

Here, the value 50 is assigned to y and then the result 50+ 10 = 60 is assigned to x. This statement is identical to

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

Compound Assignment

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

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

may be written as

```
x += 10;
```

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

```
variable1 op= variable2;
```

where *op* is a binary arithmetic operator. This means that

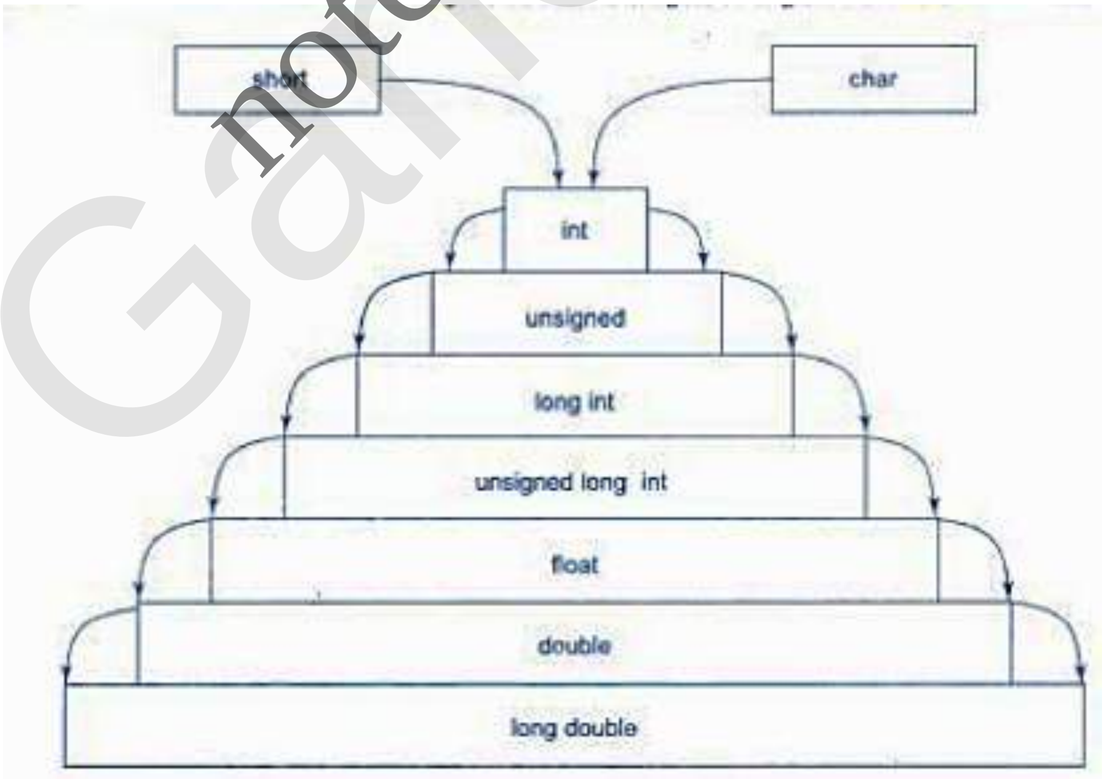
```
variable1 = variable1 op variable2;
```

Implicit Conversions

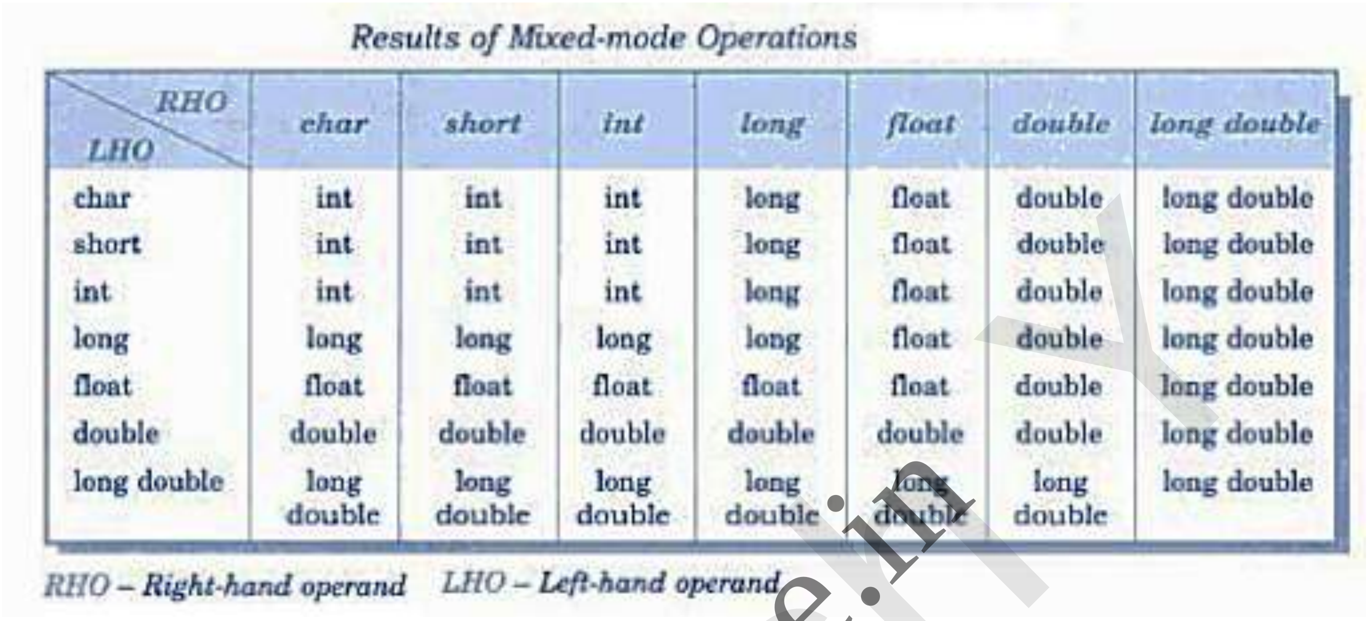
We can mix data types in expressions. For example,

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

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



When the compiler encounters an expression, it divides the expressions into sub expressions consisting of one operator and one or two operands. For a binary operator, if the operands type differ the compiler converts one of them to match with the other, using the rule that the ***"smaller" type is converted to the "wider" type***.



Results of Mixed-mode Operations

LHO \ RHO	char	short	int	long	float	double	long double
char	int	int	int	long	float	double	long double
short	int	int	int	long	float	double	long double
int	int	int	int	long	float	double	long double
long	long	long	long	long	float	double	long double
float	float	float	float	float	float	double	long double
double	double	double	double	double	double	double	long double
long double	long double	long double	long double	long double	long double	long double	long double

RHO – Right-hand operand LHO – Left-hand operand

For example, if one of the operand is an **int** and the other is a **float**, the **int** is converted into a **float** because a **float** is wider than an **int.** The "waterfall" model shown in above figure illustrates this rule.

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

Operator Overloading

Overloading means assigning different meanings to an operation, depending on the context.

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

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

Thus, the statement

```
cout<<75.86;
```

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

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

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

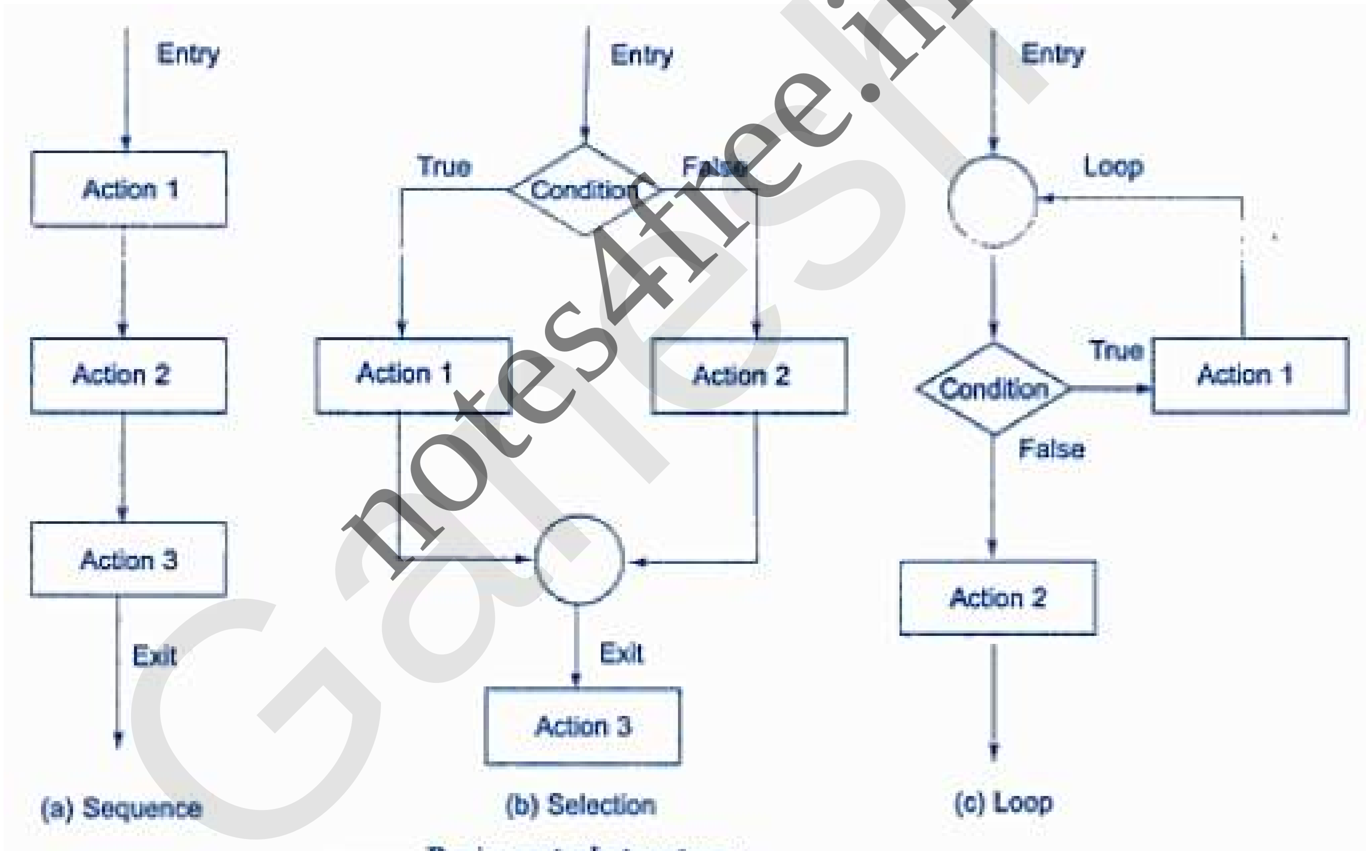
Almost all C++ operators can be overloaded with a few exceptions such as the member-access operators (`.and .*`), conditional operator (`?:`), scope resolution operator (`::`) and the size operator (`sizeof`).

Control Structures

One method of achieving the objective of an accurate, error resistant and maintainable code is to use one or any combination of the following three control structures:

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

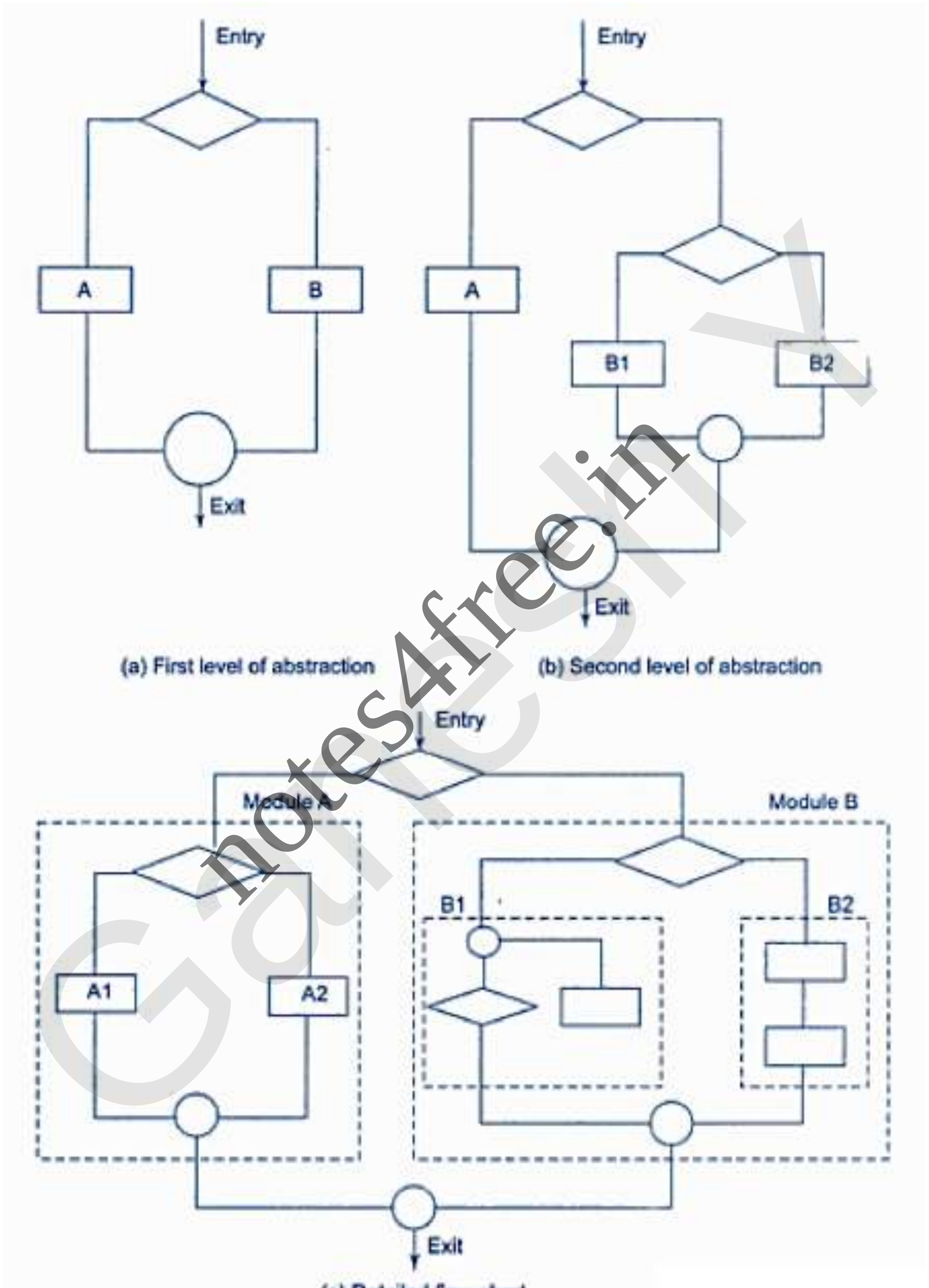
Figure below shows how these structures are implemented using *one-entry, one-exit* concept, a popular approach used in modular programming.



Basic control structures

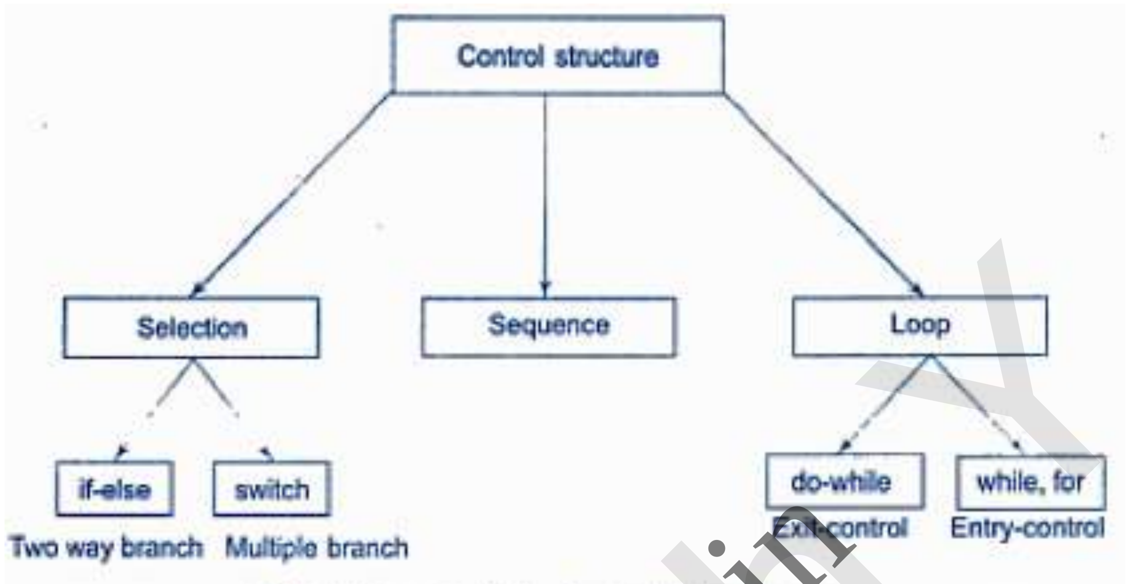
It i s important to understand that all program processing can be coded by using only these three logic structures. The approach of using one or more of these basic control constructs in programming is known as ***structured programming,*** an important technique in software engineering.

Using these three basic constructs, we may represent a function structure either in detail or in summary form as shown in following Figs (a), (b) and (c).



Different levels of abstraction

Like C, C++ also supports all the three basic control structures, and implements them using various control statements as shown in following Fig. This shows that *C++ combines the power of structured programming with the object-oriented paradigm.*



C++ statements to implement in two forms

The if statement

The if statement is implemented in two forms:

- Simple **if** statement
- **If.....else** statement

Examples:

```
Form 1
if(expression is true)
{
     action1;
}
action2;
action3;
```

```
Form 2
if (expression is
true)
{
     action1;
}
else
{
     action2;
}
action3;
```

The switch statement

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

```
switch(expression)
{
     case 1:
     {
          action1;
     }
     case 2:
     {
          action2;
     }
     case 3:
     {
          action3;
     }
     default:
     {
          action4;
     }
}
action5;
```

The do-while statement

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

```
do
{
     action1;
}
while(condition is true);
act1on2;
```

The while statement

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

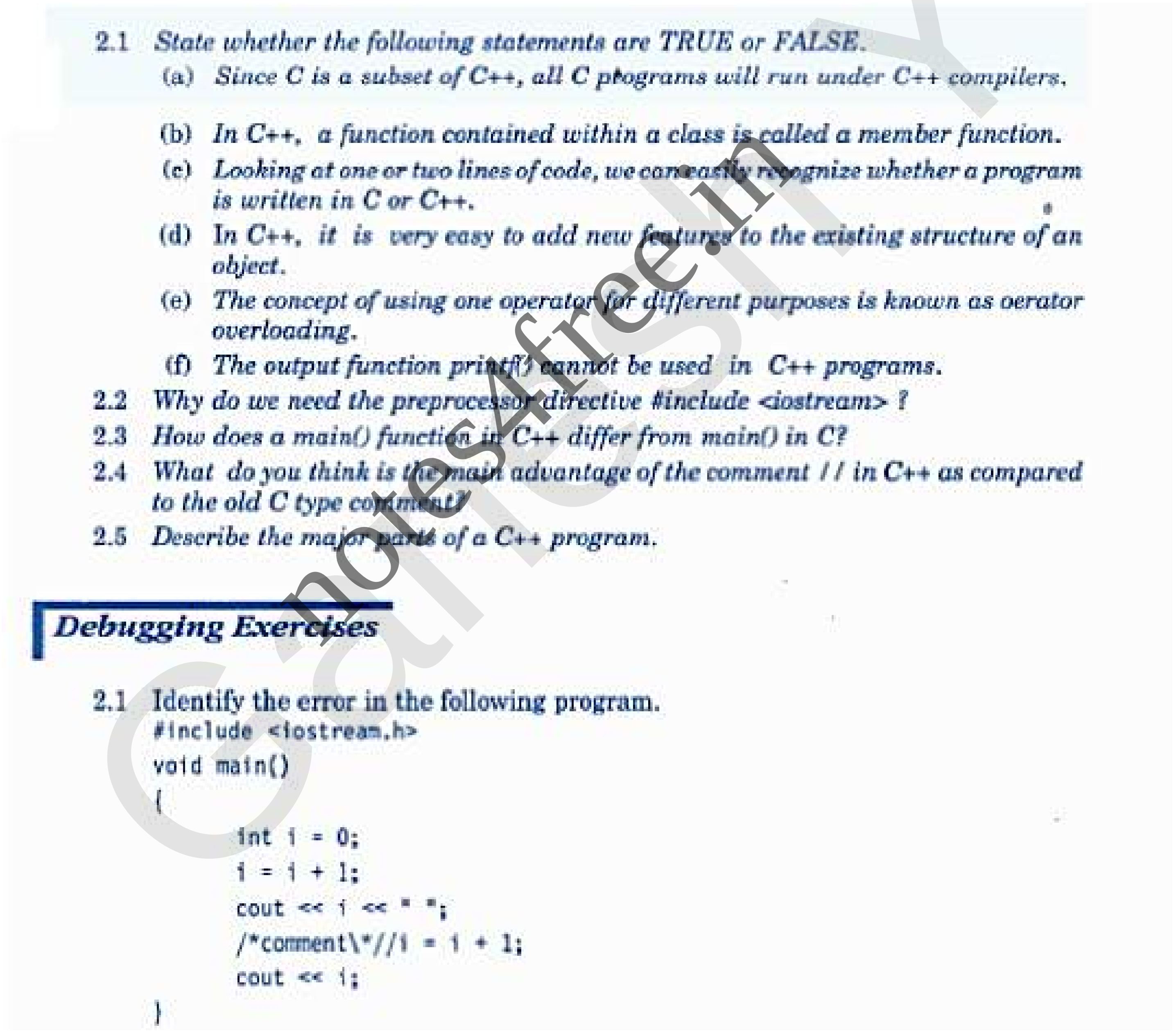
```
while(condition is true)
{
     action1;
}
action2;
```

The for statement

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

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

Exercise Questions and Solutions



2.1 *State whether the following statements are TRUE or FALSE.*

(a) *Since C is a subset of C++, all C programs will run under C++ compilers.*

(b) *In C++, a function contained within a class is called a member function.*

(c) *Looking at one or two lines of code, we can easily recognize whether a program is written in C or C++.*

(d) *In C++, it is very easy to add new features to the existing structure of an object.*

(e) *The concept of using one operator for different purposes is known as oerator overloading.*

(f) *The output function printf() cannot be used in C++ programs.*

2.2 *Why do we need the preprocessor directive #include <iostream> ?*

2.3 *How does a main() function in C++ differ from main() in C?*

2.4 *What do you think is the main advantage of the comment // in C++ as compared to the old C type comment?*

2.5 *Describe the major parts of a C++ program.*

Debugging Exercises

2.1 Identify the error in the following program.

```
#include <iostream.h>
void main()
{
        int i = 0;
        i = i + 1;
        cout << i << " ";
        /*comment\*//i = i + 1;
        cout << i;
}
```

2.2 **Identify the error in the following program.**

```
#include <iostream.h>
void main()
{
        short i=2500, j=3000;
        cout >> "i + j = " >> -(i+j);
}
```

2.3 **What will happen when you run the following program?**

```
#include <iostream.h>
void main()
{
        int i=10, j=5;
        int modResult=0;
        int divResult=0;

        modResult = i%j;
        cout << modResult << " ";

        divResult = i/modResult;
        cout << divResult;
}
```

2.4 Find errors, if any, in the following C++ statements.

(a) cout << "x=" x;
(b) m = 5; // n = 10; // s = m + n;
(c) cin >>x; >>y;
(d) cout << \n "Name:" << name;
(e) cout <<"Enter value:"; cin >> x;
(f) /*Addition*/ z = x + y;

Programming Exercises

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

```
Maths     = 90
Physics   = 77
Chemistry = 69
```

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

2.3 *Write a program to input an integer value from keyboard and display on screen "WELL DONE" that many times.*

2.4 *Write a program to read the values of a, b and c and display the value of x, where*

$$x = a / b - c$$

Test your program for the following values:

(a) $a = 250, b = 85, c = 25$
(b) $a = 300, b = 70, c = 70$

2.5 *Write a C++ program that will ask for a temperature in Fahrenheit and display it in Celsius.*

2.6 *Redo Exercise 2.5 using a class called* ***temp*** *and member functions.*

2.1: State whether the following statements are TRUE or FALSE.
(a) Since C is a subset of C++, all C peograms will run under C++ compilers.
(b) In C++, a function contained within a class is called a member function.
(c) Looking at one or two lines of code, we can easily recognize whether a program is written in C or C++.
(d) In C++, it is very easy to add new features to the existing structure of an object.
(e) The concept of using one operator for different purposes is known as aerator overloading. 10 The output function printf) cannot be used in C++ programs.

Ans:
a> FALSE
b> TRUE
c> FALSE
*** most lines of codes are the same in C & C++
d> TRUE
e> TRUE
f> FALSE

2.2: Why do we need the preprocessor directive #include<iostream>?

Ans: '#include<iostream>' directive causes the preprocessor to add-the contents of iostream file to the program.
2.3: How does a main() function in C++ differ from main{} in C?

Ans: In C main () by default returns the void type but in C++ it returns integer by default.
2.4: What do you think is the main advantage of the comment / / in C++ as compared to the old C type comment?

Ans: '//' is more easy and time-saving than '/* */'

2.5:Describe the major parts of a C++ program.

Ans: Major parts of a C++ program :
1. Include files
2. Class declaration
3. Member function definitions
4. Main function program

Debugging Exercises

2.1: **Identify the error in the following program.**

```
#include<iostream.h>
void main()
{
   int i = 0;
   i = i + 1;
   cout « i « " ";
   /*comment \*//i = i + 1;
   cout << i;
}
```

Ans: Syntax error→/* comment*//i=i+1;

2.2: **Identify the error in the following program.**

```
#include<iostream.h>
void main()
{
    short i=2500, j=3000;
    cour>> "i+j=">> -(i+j);
}
```

Ans: cout >> "i+j=">> Illegal structure operation.→-(i + j);

2.3: **What will happen when you run the following program?**

```
#include<iostream.h>
void main()
{
    int i=10, j=5;
    int modResult=0;
    int divResult=0;
    modResult = i%j;
    cout<<modResult<<" ";
    divResult = i/modResult;
    cout<<divResult;
}
```

Ans: floating point Error or divide by zero→divResult = i/modResult;

Note: If this kind of Error exist in a program, the program will successfully compile but it will show Run Error.

2.4: Find errors, if any, in the following C++ statements.
(a) cout.<<"x=" x; (b) m = 5; // n = 10; // = m + n: (c) cin >>x; >>y;
(d) cout <<\h 'Name:" <<name;
(e) cout <<"Enter value:"; cin >> x:
(f) /*Addition*/ z = x + y;

Ans:

	Error	Correction
A	Statement missing	cout<<"x="<<x;
B	No error	
C	Expression-syntax-error	cin>>x>>y;
D	Illigal character '\'. Statement missing	cout<<"\n Name"<<name;
E	No error	
F	No error	

Programming Exercises

2.1: Write a program to display the following output using a single cout statement

Maths = 90
Physics = 77
Chemistry = 69

Solution:

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

        char *sub[]={"Maths","Physics","Chemestry"};
        int mark[]={90,77,69};
        for(int i=0;i<3;i++)
         {
cout<<setw(10)<<sub[i]<<setw(3)<<"="<<setw(4)<<mark[i]<<endl;
          }
   return 0;
}
```

output

```
Maths = 90
Physics = 77
Chemistry = 69
```

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

Solution:

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

int main()
{
    float a,b;
        cout<<" Enter two values  :"<<endl;
    cin>>a>>b;
    if(a>b)
        cout<<" larger value = "<<a<<endl;
    else
        cout<<" larger value = "<<b<<endl;
    return 0;
}
```

output

```
Enter two values : 10 20
larger value = 20
```

2.3: Write a program to input an integer from the keyboard and display on the screen "WELL DONE" that many times.

Solution:

```
Solution:
#include<iostream.h>
#include<iomanip.h>
int main()
{
    int n;
    char *str;
    str="WELL DONE";
    cout<<" Enter an integer value ";
    cin>>n;
    for(int i=0;i<n;i++)
    {
        cout<<str<<endl;
    }
    return 0;
}
```

output

```
Enter an integer value 5
WELL DONE
WELL DONE
WELL DONE
WELL DONE
WELL DONE
```

2.4: **Write a program to read the values a, b and c and display x, where**
x = a / b –c.
Test the program for the following values:
(a) a = 250, b = 85, c = 25
(b) a = 300, b = 70, c = 70

Solution:

```
#include<iostream.h>
#include<iomanip.h>
int main()
{
    float a,b,c,x;
    cout<<" Enter the value of a,b, &c :"<<endl;
    cin>>a>>b>>c;
    if((b-c)!=0)
    {
         x=a/(b-c);
         cout<<" x=a/(b-c) = "<<x<<endl;
    }
    else
    {
       cout<<"  x= infinity "<<endl;
    }
    return 0;
}
```

During First Run:

output

```
Enter the value of a,b, &c : 250 85 25
x=a/(b-c) = 4.166667
```

During Second Run:

output

```
Enter the value of a,b, &c : 300 70 70
x= infinity
```

2.5: **Write a C++ program that will ask for a temperature in Fahrenheit and display it in Celsius**
Solution:

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

int main()
{
   float f,theta;
   cout<<" Enter  the temperature in Feranhite    scale  : ";
   cin>>f;
   theta=((f-32)/9)*5;
   cout<<" Temperature in Celsius =  "<<theta<<endl;
   return 0;
}
```

output

```
Enter the temperature in Feranhite scale : 105
Temperature in Celsius = 40.555557
```

2.6: **Redo Exercise 2.5 using a class called temp and member functions.**

Solution:

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

class temp
{
    float f,theta;
public:
    float conversion(float f);
};

float temp::conversion(float f)
{
    theta=((f-32)/9)*5;
    return theta;
}
int main()
{
    temp t;
    float f;
    cout<<" Enter temperature in Farenheite scale :"<<endl;
    cin>>f;
    cout<<" Temperature in Celsius scale = "<<t.conversion(f)<<endl;
    return 0;
}
```

output

```
Enter the temperature in Feranhite scale : 112
Temperature in Celsius = 44.444443
```

3.2 *An **unsigned int** can be twice as large as the **signed int**. Explain how?*

3.3 *Why does C++ have type modifiers?*

3.4 *What are the applications of **void** data type in C++?*

3.5 *Can we assign a **void** pointer to an **int** type pointer? If not, why? How can we achieve this?*

3.6 *Describe, with examples, the uses of enumeration data types.*

3.7 *Describe the differences in the implementation of **enum** data type in ANSI C and C++.*

3.8 *Why is an array called a derived data type?*

3.9 *The size of a **char** array that is declared to store a string should be one larger than the number of characters in the string. Why?*

3.10 *The **const** was taken from C++ and incorporated in ANSI C, although quite differently. Explain.*

3.11 *How does a constant defined by **const** differ from the constant defined by the preprocessor statement **#define**?*

3.12 *In C++, a variable can be declared anywhere in the scope. What is the significance of this feature?*

3.13 *What do you mean by dynamic initialization of a variable? Give an example.*

3.14 *What is a reference variable? What is its major use?*

3.15 *List at least four new operators added by C++ which aid OOP.*

3.16 *What is the application of the scope resolution operator :: in C++?*

3.17 *What are the advantages of using **new** operator as compared to the function **malloc()**?*

3.18 *Illustrate with an example, how the **setw** manipulator works.*

3.19 *How do the following statements differ?*

(a) *char * const p;*

(b) *char const *p;*

Debugging Exercises

3.1 What will happen when you execute the following code?

```
#include <iostream.h>
void main()
{
        int i=0;
        i=400*400/400;
        cout << i;
}
```

3.2 Identify the error in the following program.

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

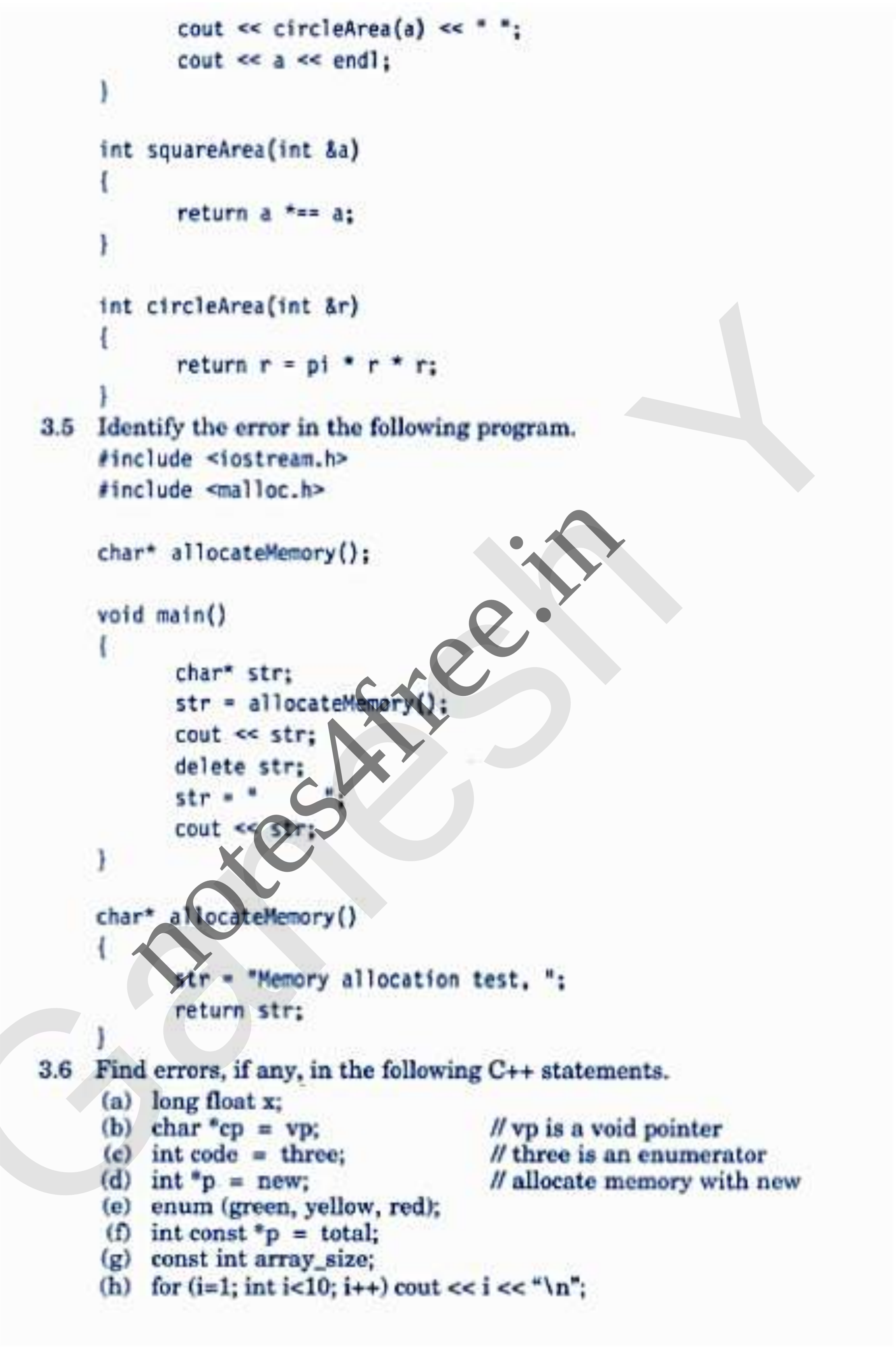
```
{
        int num[]={1,2,3,4,5,6};
        num[1]==[1]num  ?    cout<<"Success"    :    cout<<"Error";
}
```

3.3 **Identify the errors in the following program.**

```
#include <iostream.h>
void main()
{
  int i=5;
  while(i)
  {
          switch(i)
          {
          default:
          case 4:
          case 5:

          break;

          case 1:
          continue;

          case 2:
          case 3:
          break;

          }
          i--;
  }
}
```

3.4 **Identify the error in the following program.**

```
#include <iostream.h>
#define pi 3.14
int squareArea(int &);
int circleArea(int &);

void main()
{
       int a=10;
       cout << squareArea(a) << " ";
```



```
        cout << circleArea(a) << " ";
        cout << a << endl;
}

int squareArea(int &a)
{
        return a *== a;
}

int circleArea(int &r)
{
        return r = pi * r * r;
}
```

3.5 **Identify the error in the following program.**

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

char* allocateMemory();

void main()
{
        char* str;
        str = allocateMemory();
        cout << str;
        delete str;
        str = "       ";
        cout << str;
}

char* allocateMemory()
{
        str = "Memory allocation test, ";
        return str;
}
```

3.6 **Find errors, if any, in the following C++ statements.**

(a) long float x;
(b) char *cp = vp;   // vp is a void pointer
(c) int code = three;   // three is an enumerator
(d) int *p = new;   // allocate memory with new
(e) enum (green, yellow, red);
(f) int const *p = total;
(g) const int array_size;
(h) for (i=1; int i<10; i++) cout << i << "\n";

(i) int & number = 100;
(j) float *p = new int [10];
(k) int public = 1000;
(l) char name[3] = "USA";

Programming Exercises

3.1 *Write a function using reference variables as arguments to swap the values of a pair of integers.*

3.2 *Write a function that creates a vector of user-given size **M** using **new** operator.*

3.3 *Write a program to print the following output using **for** loops.*

```
1
22
333
4444
55555
..........
```

3.4 *Write a program to evaluate the following investment equation*

$V = P(1 + r)^n$

and print the tables which would give the value of V for various combination of the following values of P, r and n:

P: 1000, 2000, 3000, ..., 10,000

r: 0.10, 0.11, 0.12, ..., 0.20

n: 1, 2, 3, ..., 10

(Hint: P is the principal amount and V is the value of money at the end of n years. This equation can be recursively written as

$V = P(1 + r)$

$P = V$

In other words, the value of money at the end of the first year becomes the principal amount for the next year and so on.

3.5 *An election is contested by five candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and **count** the votes cast for each candidate using an array variable count. In case, a number read is outside the range 1 to 5, the ballot should be considered as a 'spoilt ballot', and the program should also count the number of spoilt ballots.*

3.6 *A cricket team has the following table of batting figures for a series of test matches:*

Player's name	*Runs*	*Innings*	*Times not out*
Sachin	8430	230	18
Saurav	4200	130	9
Rahul	3350	105	11
.	.	.	.
.	.	.	.

Write a program to read the figures set out in the above form, to calculate the batting averages and to print out the complete table including the averages.

3.7 *Write programs to evaluate the following functions to 0.0001% accuracy.*

(a) $\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots\cdots$

(b) $SUM = 1 + (1/2)^2 + (1/3)^3 + (1/4)^4 + \cdots\cdots$

(c) $\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots\cdots$

3.8 *Write a program to print a table of values of the function*
$y = e^{-x}$
for x varying from 0 to 10 in steps of 0.1. The table should appear as follows.

TABLE FOR Y = EXP [-X]

X	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.0									
1.0									
.									
.									
.									
9.0									

3.9 *Write a program to calculate the variance and standard deviation of N numbers.*

$$\text{Variance} = \frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2$$

$$\text{Standard Deviation} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

where $\bar{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$

3.10 *An electricity board charges the following rates to domestic users to discourage large consumption of energy:*

For the first 100 units	-	*60P per unit*
For next 200 units	-	*80P per unit*
Beyond 300 units	-	*90P per unit*

All users are charged a minimum of Rs. 50.00. If the total amount is more than Rs. 300.00 then an additional surcharge of 15% is added.

Write a program to read the names of users and number of units consumed and print out the charges with names.

Ans:
In case of unsigned int the range of the input value is : 0 to $2^m - 1$. [where m is no. of bit]
In case of signed int the range of the input value is : -2^{m-1} to $+ (2^{m-1} - 1)$

$$\frac{\text{maximum value for unsigned int}}{\text{value for signed int}} = \frac{2^m - 1}{\frac{(2^{m-1}-1) + 2^{m-1}}{2}}$$

$$= \frac{2^m - 1}{\frac{2.2^{m-1}-1}{2}}$$

$$= \frac{2^m - 1}{\frac{2^m-1}{2}}$$

$$= 2$$

So, maximum value for unsigned int can be twice as large as the signed int.
* Here the absolute value of lower value -2^{m-1} for signed int must be considered for finding average value of signed int.

3.3: **Why does C++ have type modifiers?**

Ans: To serve the needs of various situation.

3.4: **What are the applications of void data type in C++?**

Ans: Two normal uses of void are
(1) to specify the return type of a function when it is not returning any value.
(2) To indicate any empty argument list to a function.

Example : void function (void)

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

Example :
void *gp; //gp is generic pointer.
A pointer value of any basic data type can be assigned to a generic pointer
int * ip;
gp = ip; // valid.

3.5: **Can we assign a void pointer to an int type pointer? If not, why? Now can we achieve this?**

Ans: We cannot assign a void pointer to an int type pointer directly. Because to assign a pointer to another pointer data type must be matched. We can achieve this using casting.

Example :
void * gp;
int *ip;
ip = (int *) gp
/br<>
3.6: **Describe, with examples, the uses of enumeration data types.**

Ans: An enumerated data type is a user-defined type. It provides a way for attaching names to numbers in ANSIC

Example :
enum kuet (EEE, CSE, ECE, CE, ME, IEM);

The enum keyword automatically enumerates
EEE to 0
CSE to 1
ECE to 2
CE to 3
ME to 4
IEM to 5

In C++ each enumerated data type retains its own separate type.

3.7: **Describe the differences in the implementation of enum data type in ANSI C and C++.**

Ans: Consider the following example :
enum kuet (EEE, CSE, ECE, CE, ME, IEM);

Here, kuet is tag name.

In C++ tag name become new type name. We can declare a new variables Example:

kuet student;
ANSI C defines the types of enum to be int.

In C int value can be automatically converted to on enum value. But in C++ this is not permitted.

Example:
student cgp = 3.01 // Error in C++
// OK in C.

student cgp = (student) 3.01 //OK in C++

3.8: **Why is an army called a derived data type?**

Ans: Derived data types are the data types which are derived from the fundamental data types. Arrays refer to a list of finite number of same data types. The data can be accessed by an index number from o to n. Hence an array is derived from the basic date type, so array is called derived data type.

3.9: **The size of a char array that is declared to store a string should be one larger than the number of characters in the string. Why?**

Ans: An additional null character must assign at the end of the string that's why the size of char array that is declared to store a string should be one larger than the number of characters in the string.

3.10: **The const was taken from C++ and incorporated in ANSI C, although quite differently. Explain.**

Ans: In both C and C++, any value declared as const cannot be modified by the program in any way. However there are some differences in implementation. In C++ we can use const in a constant expression, such as const int size = 10; char name [size];
This would be illegal in C. If we use const modifier alone, it defaults to int. For example, const size = 10; means const int size = 10;
C++ requires const to be initialized. ANSI C does not require an initialization if none is given, it initializes the const to o. In C++ a const is local, it can be made as global defining it as external. In C const is global in nature , it can be made as local declaring it as static.

3.11: **How does a constant defined by cowl differ from the constant defined by the preprocessor statement %define?**

Ans: Consider a example : # define PI 3.14159

The preprocessor directive # define appearing at the beginning of your program specifies that the identifier PI will be replace by the text 3.14159 throughout the program.

The keyword const (for constant) precedes the data type of a variable specifies that the value of a variable will not be changed throughout the program.

In short, const allows us to create typed constants instead of having to use # define to create constants that have no type information.

3.12: In C++. a variable can be declared anywhere in the scope. What is the significance of this feature?

Ans: It is very easy to understand the reason of which the variable is declared.

3.13: What do you mean by dynamic initialization of a variable? Give an example.

Ans: When initialization is done at the time of declaration then it is know as dynamic initialization of variable

Example :
float area = 3.14159*rad * rad;

3.14: What is a reference variable? What is its major use?

Ans: A reference variable provides an alias (alternative name) for a previously defined variable. A major application of reference variables is in passing arguments to functions.

3.15: List at least four new operators added by C++ which aid OOP.

Ans:
New opperators added by C++ are :
1. Scope resolution operator ::
2. Memory release operator delete   delete
3. Memory allocation operator   new
4. Field width operator   setw
5. Line feed operator   endl

3.16: What is the application of the scope resolution operator :: in C++?

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

3.17: What are the advantages of using new operator as compared to the junction ntallocOr

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

3.18: **Illustrate with an example, how the seize manipulator works.**

Ans:
setw manipulator specifies the number of columns to print. The number of columns is equal the value of argument of setw () function.

For example :

setw (10) specifies 10 columns and print the massage at right justified.

cout << set (10) << "1234"; will print

```
                                    1    2    3    4
```

If argument is negative massage will be printed at left justified.

cout <<setw(-10)<< "1234"; will print

```
1    2    3    4
```

3.19: **How do the following statements differ?**
(a) char *const p;
(b) char canal *p;

Ans:
(a) Char * const P; means constant pointer.
(b) Char const * P; means pointer to a constant.

In case of (a) we con not modify the address of p.
In case of (b) we can not modify the contents of what it points to.

Debugging Exercises

3.1: **What will happen when you execute the following code?**

```
1#include <iostream.h>
2void main()
3{
4    int i=0;
5    i=400*400/400;
6    cout<<i;
7}
```

Ans: i = 400*400/400; Here, 400*400 = 160000 which exceeds the maximum value of int variable. So wrong output will be shown when this program will be run.

Correction :

```
Int I = 0;
```

should be changed as

```
long int i = 0;
```

to see the correct output.

3.2: **Identify the error in the following program.**

```
include<iostream.h>
void main()
{
    int num[]={1,2,3,4,5,6};
    num[1]==[1]num ? cout<<"Success" : cout<<"Error";
}
```

Ans: num [1] = [1] num?. You should write index number after *array name* but here index number is mention before array name in **[1] num**

So expression syntax error will be shown.

Correction : num[1] = num[1]? is the correct format

3.3: **Identify the errors in the following program.**

```
#include <iostream.h>
void main()
{
    int i=5;
    while(i)
  {
   switch(i)
   {
   default:
   case 4:
```

```
case 5:
break;
case 1:
continue;
case 2:
case 3:
break;
}
i-;
}
}
```

Ans:

```
case 1 :
continue;
```

The above code will cause the following situation:
Program will be continuing while value of i is 1 and value of i is updating. So infinite loop will be created.
Correction: At last line i- should be changed as i–;

3.4: **Identify the errors in the following program.**

```
#include <iostream.h>
#define pi 3.14
int squareArea(int &);
int circleArea(int &);
void main()
{
    int a-10;
    cout << squareArea(a) << " ";
    cout « circleArea(a) « ";
    cout « a « endl;
{
    int squareArea(int &a)
{
    return a *== a;
}
int circleArea(int &r)
{
return r = pi * r * r;
}
```

Ans: Assignment operator should be used in the following line:

1return a *==a;

That means the above line should be changed as follows:

1return a *=a;

3.5: **Missing**

3.6: **Find errors, if any, in the following C++ statements.**
(a) long float x;
(b) char *cp = vp; // vp is a void pointer
(c) int code = three; // three is an enumerator
(d) int sp = new; // allocate memory with new
(e) enum (green, yellow, red);
(f) int const sp = total;
(g) const int array_size;
(h) for (i=1; int i<10; i++) cout << i << "/n"; (i) int & number = 100; (j) float *p = new int 1101;
(k) int public = 1000; (l) char name[33] = "USA";

Ans:

No.	Error	Correction
(a)	too many types	float x; or double x;
(b)	type must be matched	char *cp = (char*) vp;
(c)	No error	
(d)	syntax error	int*p = new int [10];
(e)	tag name missing	enum colour (green, yellow, red)
(f)	address have to assign instead of content	int const * p = &total;
(g)	C++ requires a const to be initialized	const int array-size = 5;
(h)	Undefined symbol i	for (int I = 1; i <10; i++) cout << i << "/n";
(i)	invalid variable name	int number = 100;
(j)	wrong data type	float *p = new float [10];
(k)	keyword can not be used as a variable name	int public1 = 1000;
(l)	array size of char must be larger than the number of characters in the string	char name [4] = "USA";

Programming Exercises

3.1: Write a function using reference variables as arguments to swap the values of a pair of integers.

Solution:

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

void swap_func(int &a,int &b)
{

    cout<<" Before swapping "<<endl
        <<" a = "<<a<<endl<<" b = "<<b<<endl<<endl;
    int temp;
    temp=a;
    a=b;
    b=temp;
    cout<<" After swapping "<<endl
        <<" a = "<<a<<endl<<" b = "<<b<<endl<<endl;

}

int main()
{
    int x,y;
    cout<<"  Enter two integer value : "<<endl;
    cin>>x>>y;
    swap_func (x,y);
    return 0;
}
```

output

```
Enter two integer value : 56 61
Before swapping
a = 56
b = 61
After swapping
a = 56
b = 61
```

3.2: **Write a function that creates a vector of user given size M using new operator.**

Solution:

```
#include<iostream.h>
#include<iomanip.h>
int main()
{
   int m;
   int *v;
   cout<<"  Enter vector size : "<<endl;
   cin>>m;
   v=new int [m];
   cout<<" to check your performance  insert "<<m<<" integer value"<<endl;
   for(int i=0;i<m;i++)
   {
      cin>>v[i];
   }
   cout<<" Given integer value are :"<<endl;
   for(i=0;i<m;i++)
   {

      if(i==m-1)
      cout<<v[i];
      else
      cout<<v[i]<<",";

   }
   cout<<endl;
   return 0;
}
```

output

```
Enter vector size : 5
to check your performance insert 5 integer value
7 5 9 6 1
Given integer value are :
7, 5, 9, 6, 1
```

3.3: **Write a program to print the following outputs using for loops**

```
1
22
333
4444
```

55555
………………

Solution:

```
#include<iostream.h>
#include<iomanip.h>
int main()
{
   int n;
   cout<<" Enter your desired number :"<<endl;
   cin>>n;
   cout<<endl<<endl;
   for(int i=1;i<=n;i++)
   {
      for(int j=1;j<=i;j++)
      {
         cout<<i;
      }
      cout<<endl;
   }
   return 0;
}
```

output

```
Enter your desired number : 6
1
22
333
4444
55555
666666
```

3.4: **Write a program to evaluate the following investment equation**
$V = P(1+r)^n$
and print the tables which would give the value of V for various
of the following values of P, r and n:
P: 1000, 2000, 3000,……………,10,000
r: 0.10, 0.11, 0.12,…………………,0.20
n: 1, 2, 3,…………………………………..,10
(Hint: P is the principal amount and V is the value of money at the end of n years. This equation can be recursively written as
$V = P(1 + r)$
$P = V$
In other words, the value of money at the end of the first year becomes the principal amount for the next year and so on)

Solution:

```
#include<iostream.h>
#include<iomanip.h>
#include<math.h>
#define  size 8

int main()
{
   float v,pf;
   int n=size;
   float p[size]={1000,2000,3000,4000,5000,6000,7000,8000};//9000,1000};
   float r[size]={0.11,0.12,0.13,0.14,0.15,0.16,0.17,0.18};//,0.19,0.20};

   cout<<setw(5)<<"n=1";
      for(int i =2;i<=size;i++)
   cout<<setw(9)<<"n="<<i;
   cout<<"\n";

   for(i=0;i<size;i++)
   {
      cout<<setw(-6)<<"p=";
      for(int j=0;j<size;j++)
      {
        if(j==0)
         pf=p[i];

         v=pf*(1+r[i]);

         cout.precision(2);
                cout.setf(ios::fixed, ios::floatfield);
         cout<<v<<setw(10);
          pf=v;
      }
       cout<<"\n";

   }
   return 0;
}
```

output

```
n=1        n=2         n=3         n=4         n=5         n=6         n=7
p=1110    1232.1     1367.63    1518.07    1685.06    1870.41    2076.16
p=2240   2508.8    2809.86    3147.04    3524.68    3947.65   4421.36
p=3390   3830.7    4328.69    4891.42    5527.31    6245.86   7057.82
p=4560   5198.4    5926.18    67 55.84    7701.66    8779.89   10009.08
p=5750    6612.5    7604.37    8745.03    10056.79  11565.3    13300.1
p=6960   8073.6    9365.38   10863.84  12602.05  14618.38  16957.32
```

```
p=8190  9582.3    11211.29    13117.21    15347.14   17956.15  21008.7
p=9440  11139.2   13144.26   15510.22    18302.06   21596.43  25483.79
```

3.5: An election is contested by five candidates. The candidates are numbered 1 to 5 and the voting is done by marking the candidate number on the ballot paper. Write a program to read the ballots and count the vote cast for each candidate using an array variable count. In case, a number read is outside the range 1 to 5, the ballot should be considered as a "spoilt ballot" and the program should also count the numbers of "spoilt ballots".

Solution:

```
#include<iostream.h>
#include<iomanip.h>
int main()
{
    int count[5];
    int test;
    for(int i=0;i<5;i++)
    {
        count[i]=0;
    }
    int spoilt_ballot=0;
    cout<<" You can vot candidate 1 to 5 "<<endl
     <<" press 1 or 2 or 3 or 4 or 5 to vote "<<endl
     <<" candidate 1 or 2 or 3 or 4 or 5 respectively "<<endl
     <<" press any integer value outside the range 1 to 5 for NO VOTE    "<<endl<<" press any
negative value to terminate  and see result :"<<endl;

    while(1)
      {
         cin>>test;

         for(int i=1;i<=5;i++)
         {
            if(test==i)
            {
               count[i-1]++;
            }
         }
        if(test<0)
            break;
       else if(test>5)
            spoilt_ballot++;
     }
     for(int k=1;k<=5;k++)
     cout<<" candidate "<<k<<setw(12);
     cout<<endl;
     cout<<setw(7);
    for(k=0;k<5;k++)
    cout<<count[k]<<setw(13);
    cout<<endl;
    cout<<" spoilt_ballot "<<spoilt_ballot<<endl;
    return 0;
}
```

output

```
You can vot candidate 1 to 5
press 1 or 2 or 3 or 4 or 5 to vote
candidate 1 or 2 or 3 or 4 or 5 respectively
press any integer value outside the range 1 to 5 for NO VOTE
press any negative value to terminate and see result :
1
1
1
5
4
3
5
5
2
1
3
6
-1
candidate 1 candidate 2 candidate 3 candidate 4 candidate 5
4  1  2  1  3
spoilt_ballot 1
```

3.6: **A cricket has the following table of batting figure for a series of test matches:**

Player's name   Run   Innings   Time not
outSachin   8430   230   18Saurav   4200   130

.   .   .   .

Write a program to read the figures set out in the above forms, to calculate the batting arranges and to print out the complete table including the averages.

Solution:

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

char *serial[3]={" FIRST "," SECOND " ," THIRD "};//global declaration
```

```
int main()
{
   int n;
    char name[100][40];
   int *run;
   int *innings;
   int *time_not_out;
   cout<<" How many players' record would you insert ? :";
   cin>>n;
   //name=new char[n];
   run=new int[n];
   innings=new int[n];
   time_not_out=new int[n];

   for(int i=0;i<n;i++)
   {
      if(i>2)
      {
         cout<<"\n Input details of "<<i+1<<"th"<<" player's"<<endl;
      }
      else
      {
         cout<<" Input details of "<<serial[i]<<"player's : "<<endl;
      }

         cout<<" Enter name : ";

         cin>>name[i];
         cout<<" Enter run : ";
         cin>>run[i];
         cout<<" Enter innings : ";
         cin>>innings[i];
         cout<<" Enter times not out : ";
         cin>>time_not_out[i];
   }

      float *average;
      average=new float[n];
      for(i=0;i<n;i++)
      {
         float avrg;
         average[i]=float(run[i])/innings[i];

      }
      cout<<endl<<endl;
      cout<<setw(12)<<"player's name "<<setw(11)<<"run"<<setw(12)<<"innings"<<setw(16)<<"Average"<<setw
out"<<endl;
      for(i=0;i<n;i++)
      {
         cout<<setw(14)<<name[i]<<setw(11)<<run[i]<<setw(9)<<innings[i]<<setw(18)<<average[i]<<setw(15)<<
    }
    cout<<endl;

    return 0;
}
```

output

```
How many players record would you insert ? :2
Input details of FIRST player's :
Enter name : Sakib-Al-Hassan
Enter run : 1570
Enter innings : 83
Enter times not out : 10
Input details of SECOND player's :
Enter name : Tamim
Enter run : 2000
Enter innings : 84
Enter times not out : 5
player's name run innings Average times not out
Sakib-Al-Hassan 1570 83 18.915663 10
Tamim 2000 84 23.809525 5
```

3.7: **Write a program to evaluate the following function to 0.0001% accuracy**

(a) $\sin x = x - x^3/3! + x^5/5! - x^7/7! + \ldots\ldots\ldots\ldots$

(b) $SUM = 1+(1/2)^2 + (1/3)^3 + (1/4)^4 + \ldots\ldots\ldots$

(c) $\cos x = 1 - x^2/2! + x^4/4! - x^6/6! + \ldots\ldots\ldots$

Solution (a):

```
#include<iostream.h>
#include<math.h>
#include<iomanip.h>
#define accuracy 0.0001
#define pi 3.1416

long int fac(int a)
{
   if(a<=1)
      return 1;
   else
      return a*fac(a-1);
}
int main()
```

```
{
  float y,y1,x,fx;
  int n=1;
  int m;
  //const float pi=3.1416;
  cout<<" Enter the value of angle in terms of degree:  ";
     cin>>x;
     float d;
     d=x;
  int sign;
     sign=1;
if(x<0)
{
     x=x*(-1);
     sign=-1;
}
again:
  if(x>90 && x<=180)
  {

     x=180-x;

  }
  else if(x>180 && x<=270)
  {
     x=x-180;
     sign=-1;
  }
  else if(x>270 && x<=360)
  {
     x=360-x;
     sign=-1;
  }

  else if(x>360)
  {
     int m=int(x);
     float fractional=x-m;
     x=m%360+fractional;
     if(x>90)
        goto again;
     else
     sign=1;

  }
  x=(pi/180)*x;
  m=n+1;
  fx=0;
  for(;;)
  {
      long int h=fac(n);
```

```
        y=pow(x,n);
        int factor=pow(-1,m);
        y1=y*factor;
        fx+=y1/h;
        n=n+2;
        m++;
        if(y/h<=accuracy)
           break;
   }

   cout<<"sin("<<d<<")= "<<fx*sign<<endl;
   return 0;
}
```

output

```
Enter the value of angle in terms of degree: 120
sin(120)= 0.866027
```

Solution (b):

```
#include<iostream.h>
#include<math.h>
#define accuracy 0.0001
int main()
{
   int n;
   float sum,n1,m;
   n=1;sum=0;
   for(int i=1;;i++)
   {
        n1=float(1)/n;
        m=pow(n1,i);
        sum+=m;
        if(m<=accuracy)
          break;

        n++;
        }
   cout<<sum<<"\n";
   return 0;
}
      Sample Output(b)

        Solution: (c)
       #include<iostream.h>
#include<math.h>
#define accuracy 0.0001
```

```

        long int fac(int n)
{
    if(n<=1)
       return 1;
    else
       return n*fac(n-1);
}

        int main()
{

      float y,y1,x,fx;
      int n=1;
      int m;
      const float pi=3.1416;
      cout<<" Enter the value of angle in terms of degree:  ";
      cin>>x;
      if(x<0)
        x=x*(-1);
      x=(pi/180)*x;

        fx=1;

         m=2;
        float y2;
        long int h;
        for(;;)
        {
            h=fac(m);
            int factor=pow(-1,n);
            y1=pow(x,m);
            y2=(y1/h)*factor;
            fx+=y2;
            if(y1/h<=accuracy)
               break;
            m=m+2;
            n++;
        }
        cout<<fx<<"\n";
}
```

output

```
Enter the value of angle in terms of degree: 60
0.866025
```

3.8: Write a program to print a table of values of the function

$Y = e^{-x}$

For x varying from 0 to 10 in steps of 0.1. The table should appear as follows

TABLE   FOR   Y =EXP[-X];

```
  X    0.1     0.2      0.3     0.4     0.5     0.6    0.7     0.8      0.900

1.0

.

.

9.0
```

Solution:

```
#include<iostream.h>
#include<iomanip.h>
#include<math.h>
        int main()
{
      float x,y;
      cout<<"                TABLE FOR  Y=EXP(-X)            :\n\n";
      cout<<"x";
      for(float k=0;k<.7;k=k+0.1)
      cout<<setw(10)<<k;
      cout<<"\n";
      for(k=0;k<10*.7;k=k+0.1)
      cout<<"-";
      cout<<"\n";
            for(float j=0;j<10;j++)
      {
         cout<<j<<setw(4);
         for(float i=0;i<.7;i=i+0.1)
         {
               x=i+j;
               y=exp(-x);
               cout.precision(6);
                              cout.setf(ios::fixed,ios::floatfield);
               cout<<setw(10)<<y;
         }
        cout<<"\n";
    }
    return 0;
}
```

Note: Here we work with 0.4 for a good looking output.

output

```
TABLE FOR Y=EXP(-X)

x       0          0.1          0.2          0.3          0.4

0       1          0.904837     0.818731     0.740818     0.67032

1   0.367879     0.332871     0.301194     0.272532     0.246597

2   0.135335     0.122456     0.110803     0.100259     0.090718

3   0.049787    0.045049    0.040762    0.036883    0.033373

4   0.018316     0.016573     0.014996     0.013569     0.012277

5   0.006738    0.006097    0.005517     0.004992    0.004517

6   0.002479    0.002243    0.002029     0.001836    0.001662

7   0.000912    0.000825    0.000747     0.000676    0.000611

8   0.000335   0.000304   0.000275     0.000249    0.000225

9   0.000123    0.000112     0.000101     0.000091     0.000083
```

3.9: **Write a program to calculate the variance and standard deviation of N numbers**

Variance $=1/N \sum(x_i - x)^2$

Standard deviation$=\sqrt{1/N \sum(x_i - x)^2}$

Where $x = 1/N \sum x_i$

Solution:

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

```
#include<math.h>
        int main()
{
    float *x;
            cout<<" How many number ? :";
    int n;
    cin>>n;
    x=new float[n];
    float sum;
    sum=0;
    for(int i=0;i<n;i++)
    {
          cin>>x[i];
          sum+=x[i];
    }
    float mean;
    mean=sum/n;
    float v,v1;
    v1=0;
    for(i=0;i<n;i++)
    {
          v=x[i]-mean;
          v1+=pow(v,2);
    }
    float variance,std_deviation;
    variance=v1/n;
    std_deviation=sqrt(variance);
    cout<<"\n\n variance = "<<variance<<"\n standard deviation = "<<std_deviation<<"\n";

   return 0;
}
```

output

```
How many number ? :5
10
2
4
15
2
variance = 26.24
standard deviation = 5.122499
```

3.10: **An electricity board charges the following rates to domestic users to discourage large consumption of energy:**

For the first 100 units – 60P per unit

For the first 200 units – 80P per unit

For the first 300 units – 90P per unit

All users are charged a minimum of Rs. 50.00. If the total amount is more than Rs. 300.00 then an additional surcharge of 15% is added.

Write a program to read the names of users and number of units consumed and print out the charges with names.

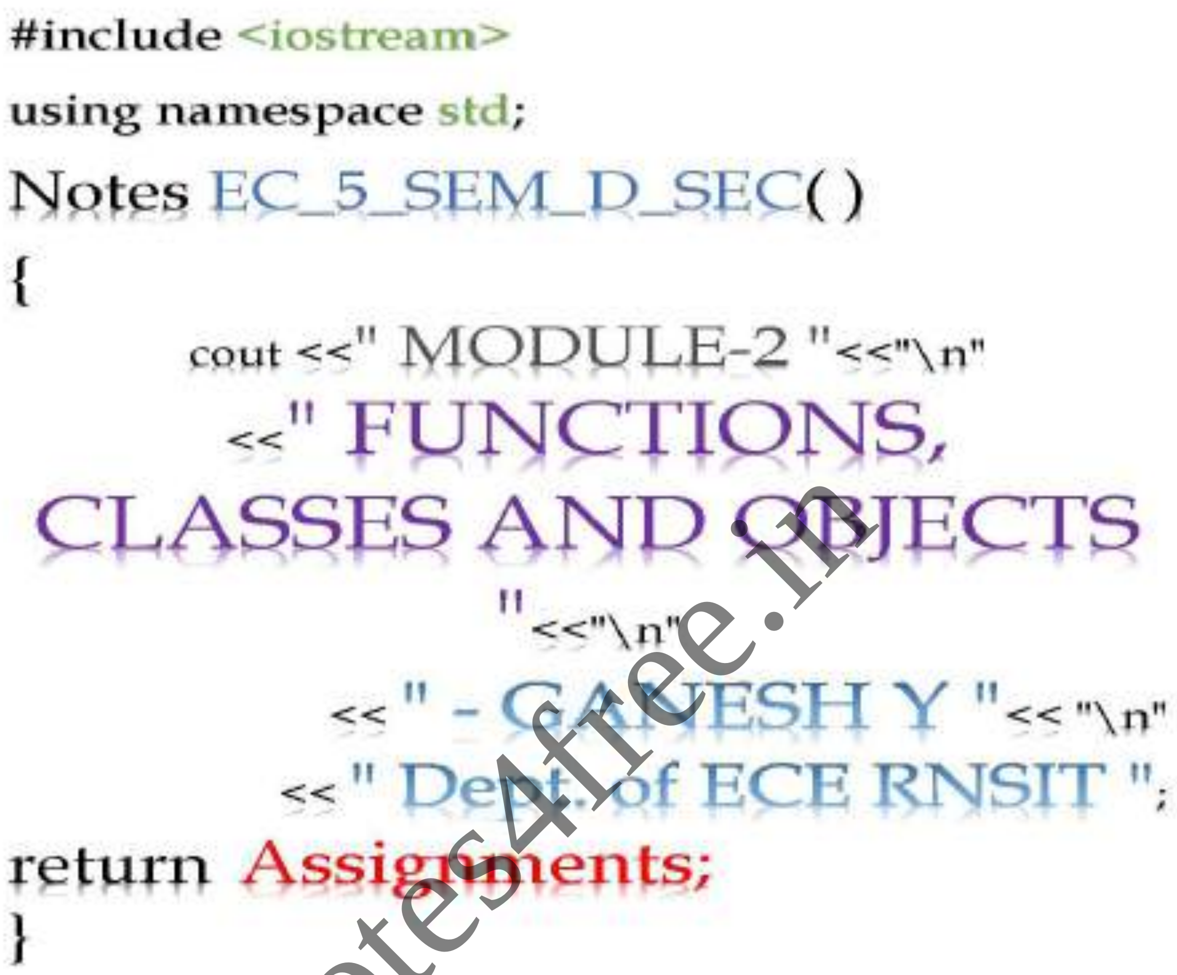
Solution:

```
#include<iostream.h>
#include<iomanip.h>
int main()
{
   int unit;
   float charge,additional;
   char name[40];
   while(1)

   {
      input:
           cout<<" Enter consumer name & unit consumed :";
             cin>>name>>unit;
             if(unit<=100)
             {
                charge=50+(60*unit)/100;
             }
             else if(unit<=300 && unit>100)
             {
                charge=50+(80*unit)/100;
             }
             else if(unit>300)
             {
                charge=50+(90*unit)/float(100);
                additional=(charge*15)/100;
                charge=charge+additional;
             }
             cout<<setw(15)<<"Name"<<setw(20)<<"Charge"<<endl;
             cout<<setw(15)<<name<<setw(20)<<charge<<endl;
             cout<<" Press o for exit / press 1 to input again :";
             int test;
             cin>>test;
             if(test==1)
             goto input;
             else if(test==0)
             break;
   }
```

```
    return 0;
}
```

output

```
Enter consumer name & unit consumed :sattar 200
Name                Charge

sattar              210

Press o for exit / press 1 to input again :1

Enter consumer name & unit consumed :santo 300

Nmae            Charge

santo           290

Press o for exit / press 1 to input again : 0
```



```
#include <iostream>
using namespace std;
Notes EC_5_SEM_D_SEC( )
{
        cout <<" MODULE-2 "<<"\n"
          <<" FUNCTIONS,
   CLASSES AND OBJECTS
                    "<<"\n"
              << " - GANESH Y "<< "\n"
            << " Dept. of ECE RNSIT ";
return Assignments;
}
```

MODULE -2

FUNCTIONS, CLASSES AND OBJECTS

GANESH Y
Dept. of ECE RNSIT

MODULE -2
Functions, Classes and Objects

SYLLABUS

Functions, classes and Objects: Functions, Inline function, function overloading, friend and virtual functions, Specifying a class, C++ program with a class, arrays within a class, memory allocation to objects, array of objects, members, pointers to members and member functions (Selected Topics from Chap-4,5 of Text1).

Introduction

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

Recall that we have used a syntax similar to the following in developing C programs.

```
void show();  /* Function declaration */
main ()
{
    ......
    show();     /* Function call */
    ......
}
void show()       /* Function definition */
{
    .......
    .......     //Function body
}
```

When the function is called control is transferred to the first statement in function body. The other statements in the function body are then executed and control returns to the main program when the closing brace is encountered.

C++ is no exception. Functions continue to be the building blocks of C++ programs. In fact, C++ bas added many new features to functions to make them more reliable and flexible. Like C++ operators, a C++ function can be overloaded to make it perform different tasks depending on the arguments passed to it. Most of these modifications are aimed at meeting the requirements of object oriented facilities.

The Main Function

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

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

This is perfectly valid because the main() in C does not return any value. In C++, the main() returns a value of type int to the operating system. C++, therefore, explicitly defines main() as matching one of the following prototypes:

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

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

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

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

Many operating systems test the return value (called *exit value)* to determine if there is any problem. The normal convention *is* that an exit value of zero means the program ran successfully. while a nonzero value means there was a problem. The explicit use of a **return(0)** statement will indicate that the program was successfully executed.

Function Prototyping

Function ***prototyping*** is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values. With function prototyping, a *template* is always used when declaring and defining a function.

When a function is called, the compiler uses the template to ensure that proper arguments are passed, and the return value is treated correctly. Any violation in matching the arguments or the return types will be caught by the compiler at the time of compilation itself. These checks and controls did not exist in the conventional C functions.

Remember, C also uses prototyping. But it was introduced first in C++ and the success of this feature inspired the ANSI C committee to adopt it.

However, there is a major difference in prototyping between C and C++. While C++ makes the prototyping essential, ANSI C makes it optional, perhaps, to preserve the compatibility with classic C.

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

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

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

Example:

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

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

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

is illegal.

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

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

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

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

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

```
            float volume(float a, float b, float c)
            {
                float v=a*b*c;
                .............
                .............
            }
```

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

```
float cube1= volume(b1,w1,h1); // Function call
```

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

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

```
void display( );
```

Which is similar to

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

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

```
void do_something(...);
```

The general form of a function is

```
ret-type function-name (parameter list)
{
     body of the function
}
```

The parameter declaration list for a function takes this general form:

```
f(type varname1, type varname2, . . . , type varnameN)

     f(int i, int k, int j) /* correct */
     f(int i, k, float j) /* incorrect */
```

Call by Value

In traditional C, a function call passes arguments by value. The ***called function*** creates a new set of variables and copies the values of arguments into them. The function does not have access to the actual variables in the calling program and can only work on the copies of values.

```
#include <stdio.h>
int sqr(int x )/* formal parameters */
{
     x = x*x;
     return(x);
}
int main(void)
{
```

```
                int t=10,a;
                a=sqr(t);  /* actual parameters */
                return 0;
            }
```

Call by Reference

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

```
        swap (int &x, int &y)
        {
            int temp;
            temp = x;
            x = y;
            y = temp;
        }
        int main()
        {
            int i, j;
            i = 10;
            j = 20;
            swap(i, j);
            return 0;
        }
    // C style Call by reference using pointers
    swap (int *x, int *y)
    {
        int temp;
        temp = *x; /* save the value at address x */
        *x = *y;   /* put y into x */
        *y = temp; /* put x into y */
    }
    int main()
    {
        int i, j;
        i = 10;
        j = 20;
        swap(&i, &j); /* pass the addresses of i and j */
        return 0;
    }
```

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

Return by reference

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

```
int & max(int &x, int &y)
{
     if (x > y)
          return x;
     else
          return y;
}
int main()
{
 int m=10,n=8,p;
 p=max(m,n); // p=m=10
 max(m,n)=-1; // returned variable=m=-1
 return 0;
}
```

Since the return type of `max()` is `int &`, the function returns reference to x or y (and not the values). Then a function call such as `max(m,n)` will yield a reference to either m or n depending on their values.

This means that this function call can appear on the left-hand side of an assignment statement as `max(m,n)=-1;`

Inline Functions

One of the objectives of using functions in a program is to save some memory space. Which becomes appreciable when a function is likely to be called many times.

However, every time a function it; called, it takes a lot of extra time in executing a series of instructions for tasks such as *jumping to the function, saving registers, pushing arguments into the stack, and returning to the calling function.*

When a function is small, a substantial percentage of execution time may be spent in such overheads.

One solution to this problem is to use ***macro definitions***, popularly known as ***macros.*** Preprocessor macros are popular in C.

The major drawback with macros is that they are not really functions and therefore, the usual error checking does not occur during compilation.

C++ has a different solution to this problem. To eliminate the cost of calls to small functions, C++ proposes a new feature called ***inline function***.

An inline function is a function that is expanded in line when it is invoked. That is, the compiler replaces the function call with the corresponding function code (something similar to macros expansion).

The inline functions are defined as follows:

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

For example

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

The above inline function can be invoked by statements like

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

If the arguments are expressions such as 2.5 + 1.5, the function passes the value of the expression, 4 in this case. <u>This makes the inline feature far superior to macros.</u>

We should exercise care before making a function inline. The speed benefits of inline functions diminish as the function grows in size. At some point the overhead of the function call becomes small compared to the execution of the function, and the benefits of inline functions may be lost. **In such cases, the use of normal functions will be more meaningful.**

Usually the functions are made inline when they are small enough to be defined in one or two lines. Example:

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

All inline functions must be defined before they are called.

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

<u>*Some of the situations where inline expansion may not work are:*</u>

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

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

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

output

```
121.228
l.25713
```

Default Arguments

C++ allows us to call a function without specifying all its arguments. In such cases, the function assigns a ***default value*** to the parameter which does not have a matching argument in the function call.

Default values are specified when the function *is* declared. The compiler looks at the prototype(declaration) to see how many arguments a function uses and alerts the program for possible default values.

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

The default value is specified in a manner syntactically similar to a variable initialization.

The above prototype declares a default value of 0.15 to the argument rate. A subsequent function call like

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

the function use default value of 0.15 for rate. The call

```
value=amount(5000,5,0.12); // no missing argument
```
passes an explicit value of 0.12 to rate

One important point to note is that only the trailing arguments can have default values and therefore we must add defaults from ***right to left***. We cannot provide a default value to a particular argument in the middle of an argument list. Some examples of function declaration with default values are:

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

```
#include <iostream>
using namespace std;
void repchar(char='*', int=45); //declaration with
int main()
{
    repchar(); //prints 45 asterisks
    repchar('='); //prints 45 equal signs
    repchar('+', 30); //prints 30 plus signs
    return 0;
}
// displays line of characters
void repchar(char ch, int n) //defaults supplied
{
    for(int j=0; j<n; j++) //loops n times
    cout << ch; //prints ch
    cout << endl;
}
```
<u>Advantages of providing the default arguments are:</u>

1. We can use default arguments to add new parameters to the existing functions.

2. Default arguments can be used to combine similar functions into one.

Example 2:

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

const Arguments

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

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

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

Recursion

Recursion is a situation where a function calls itself meaning, one of the statements in the function definition makes a call to the same function in which it is present.

It may sound like an infinite looping condition but just as a loop has a conditional check to take the program control out of the loop, recursive function also possesses a base case which returns the program from the current instance of the function to call back to the calling function.

Example 1:

```
//Calculating Factorial of a Number
#include <iostream>
#include <conio.h>
```

```
using namespace std;
long fact (int n)
{
    if(n==0)   //base case
        return 1;
    return (n* fact(n-1)); // recursive function call
}
 int main()
 {
    int num;
    cout<<"Enter a positive integer: ";
    cin>>num;
    cout<<"Factorial of "<<num<< "is"<<fact(num);
    getch();
    return 0;
 }
```

Example 2:

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk.

Approach :

<u>Take an example for 2 disks :</u>

Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.

Step 1 : Shift first disk from 'A' to 'B'.

Step 2 : Shift second disk from 'A' to 'C'.

Step 3 : Shift first disk from 'B' to 'C'.

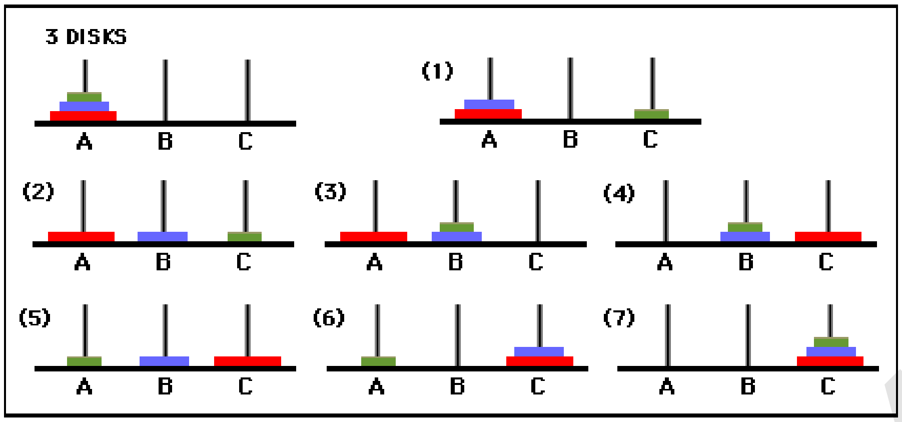
<u>The pattern here is :</u>

Shift 'n-1' disks from 'A' to 'B'.

Shift last disk from 'A' to 'C'.

Shift 'n-1' disks from 'B' to 'C'.

Image illustration for 3 disks:



```
#include <iostream>
#include <conio.h>
using namespace std;
void TOH(int d, char tower1, char tower2, char tower3)
{
     if(d==1)    //base case
     {
        cout<<"\n Shift top disk from tower "<<tower1<<" to tower
"<<tower2;
     return;
     }
     TOH(d-1,tower1,tower3,tower2); // recursive function call
     cout<<"\n Shift top disk from tower "<<tower1<<" to tower "<<
tower2;
     TOH(d-1,tower1,tower3,tower2); // recursive function call
}
 int main()
 {
     int disk;
     cout<<"Enter the no of disks: ";
     cin>>disk;
     if (disk<1)
          cout<<"\nThere are no disks to shift";
     else
     cout<<"\nThere are "<<disk<<"disks in tower1\n";
     TOH(disk,'1','2','3');
     cout <<"\n\n"<<disk<<" disks in tower 1 are shifted to tower
2";
     getch();
     return 0;}
```

Function Overloading

As stated earlier, *overloading* refers to the use of the same thing for different purposes.

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

Using the concept of function overloading; we can design a family of functions with one function name but with different argument lists. The function would perform different operations depending on the argument list in the function call. The correct function to be invoked is determined by checking the number and type of the arguments but not on the function type.

For example, an overloaded add() function handles different types of data as shown below:

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

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

Example:

```
#include<stdlib.h>
#include<iostream>
using namespace std;
int square(int x)
{
     return x*x;
}
float square(float x)
{
     return x*x;
}
int main()
{
```

```
                cout<<square(5)<<endl;
                cout<<square(2.5f);
                return 0;
            }
```

A function call first matches the prototype having the same number and type of arguments and then calls the appropriate function for execution. A best match must be unique. The function selection involves the following steps:

1. The compiler first tries to find an exact match in which the types of actual arguments are the same, and use that function.

2. If an exact match is not found, the compiler uses the integral promotions to the actual arguments, such as,

```
char to int
float to double
```

to find a match

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

```
long square(long n)
double square(double)
```

A function call such as

```
square(10)
```

will cause an error because int argument can be converted to either long or double, thereby creating an ambiguous situation as to which version of square() should be used.

4. If all of the steps fail, then the compiler will try the user defined conversions in combination with integral promotions and built-in conversions to find a unique match. User defined conversions are often used in handling class objects.

Example 2:

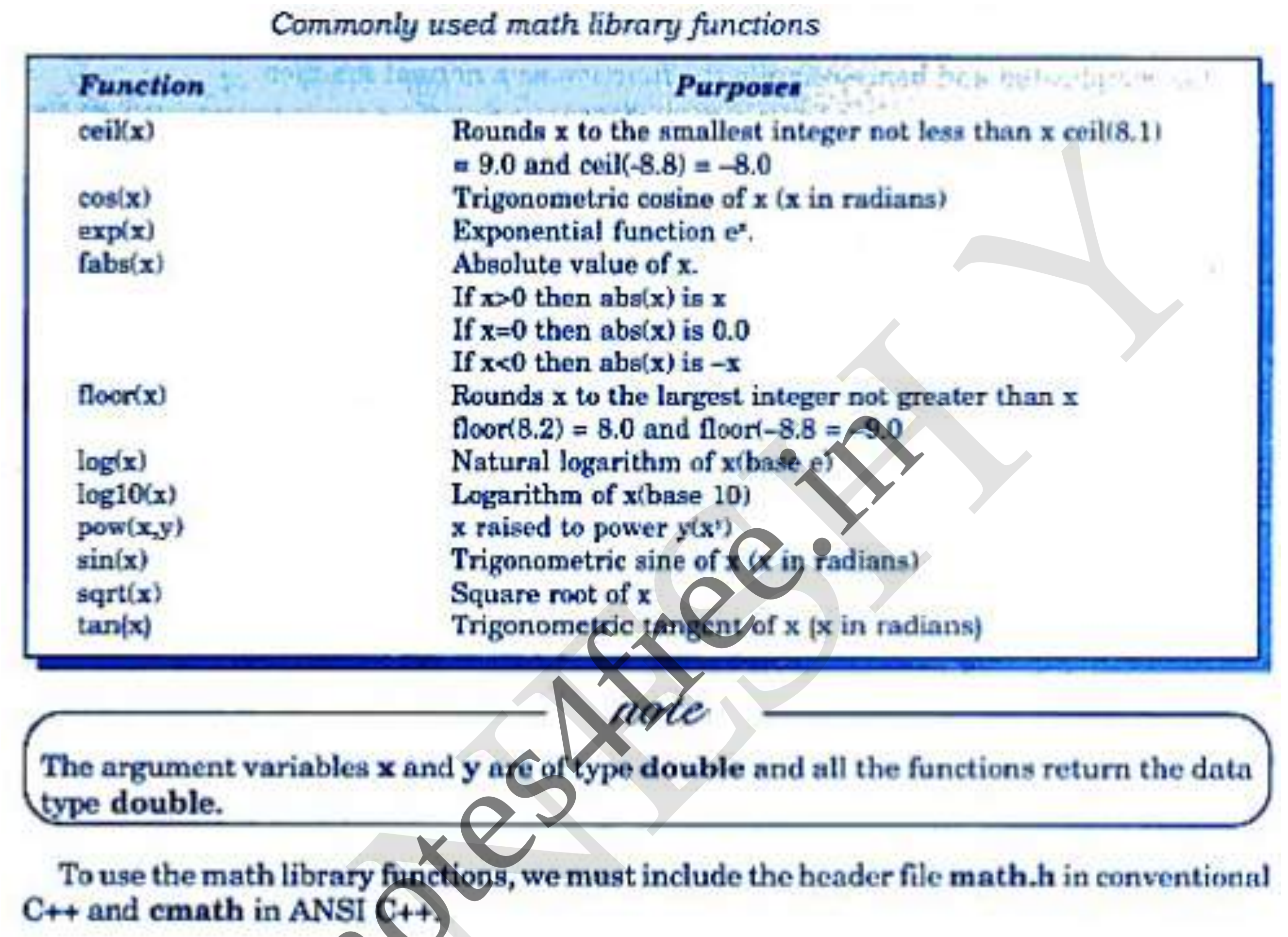
```
//function volume() is overloaded three times
#include <iostream>
using namespace std;
// declarations (prototypes)
int volume (int);
double volume (double, int);
long volume (long, int, int);
int main()
{
     cout<< volume(10) <<"\n";
     cout<< volume(2.5,8) <<"\n";
     cout<< volume(100l,75,15) <<"\n";
     return 0;
}

//function definitions
int volume(int s) //cube
{
     return (s*s*s);
}

double volume(double r, int h) // cylinder
{
     return (3.14519*r*r*h);
}
long volume (long l, int b, int h) // rectangular box
{
     return (l*b*h);
}
```

Overloading of the functions should be done with caution. We should not overload unrelated functions and should reserve function overloading for functions that perform closely related tasks.

Sometimes, the default arguments may be used instead of overloading. This may reduce the number of functions to be defined.

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 following table.



Commonly used math library functions

Function	Purposes
ceil(x)	Rounds x to the smallest integer not less than x ceil(8.1) = 9.0 and 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 abs(x) is x If x=0 then abs(x) is 0.0 If x<0 then abs(x) is –x
floor(x)	Rounds x to the largest integer not greater than x floor(8.2) = 8.0 and 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++.

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 canplex cl, 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 membe.rs.

<u>Extensions to Structures</u>

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++ decleration
```

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.

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.*

Specifying a Class

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 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 declarations;
};
```

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.

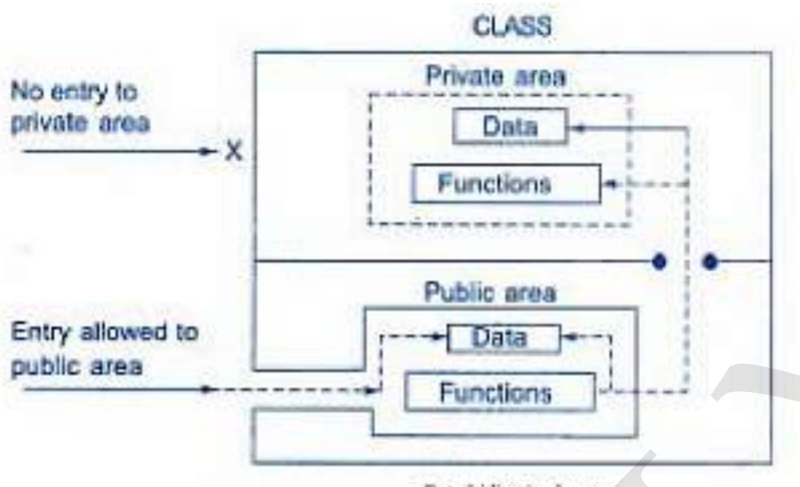
The class members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also.

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.

The variables declared inside the class are known as ***data members*** and the functions are known as ***member functions.***

Only the member functions can have access to the private data members and private functions. However, the public members (both functions and data) can be accessed from outside the class.

The binding of data and functions together into a single class-type variable is referred to as ***encapsulation.***



Data hiding in classes

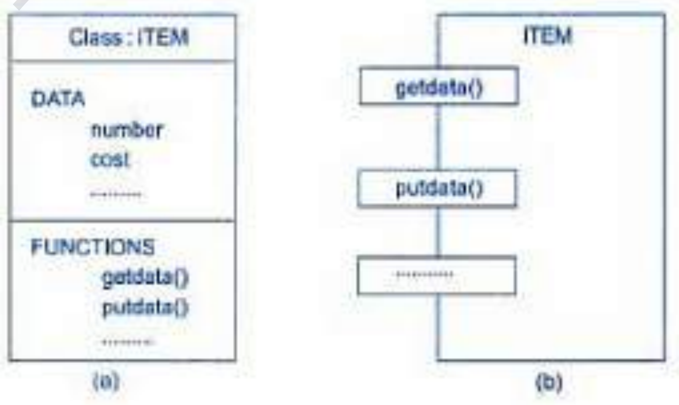
A Simple Class Example

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);
    void putdata(void);
}; // ends with semicolon
```

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.

Figure shows two different notations used by the OOP analysts to represent a class.



Representation of class

Creating Objects

```
item x; // memory for x is created
item p,q,r;
```

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.

Note that class specification, like a structure, provides only a template and does not create any memory space for the objects.

```
class item
{
     ......
     ......
     ......

} p,q,r;
```

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.

Similarly, the statement

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

would display the values of data members.

```
getdata(100,75.5); // error
x.number = 100; // error
     class xyz
     {
          int x;
          int y;
     public:
          int z;
```

```
    }
    xyz p;
    p.x =0;//error
    p.z =10;// ok z is public
```

Defining Member Functions

Member functions can be defined in two places:

- **Outside the class definition.**
- **Inside the class definition.**

Outside the Class Definition

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.*

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

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 non-member 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.

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

};
```

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

A C++ Program with Class

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

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

};
```

```
// ............ Member Function Definition ................
void item :: getdata ( int a, float b)
{
     number = a;
     cost = b;
}
// ............ Main program ...............................
int main()
{

     item x;// creats object x
     cout << "\nobject x " << "\n";
     x.getdata(100, 299.95);
     x.putdata();
     item y;// creats another object y
     cout << "\nobject y " << "\n";
     y.getdata(200, 120.25);
     y.putdata();
     return 0;
}
```

Result:

```
object x
Number=100
Cost=299.95
object y
Number=200
Cost=120.25
```

Making an Outside Function Inline

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:

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

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.***

```
#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;
     }
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()
{
     ones(); //calling member function
```

```
    cout<<"\nThe ones complement of the above binary number is: 
"<<s;
}
};
int main ( )
{
    binary b;
    b.read() ;
    b.displayones();
    getch();
    return 0;
}
```

Private Member Functions

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 n 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
```

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
}
```

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' ts tnt 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. Following program shows how these operations are implemented using a class with arrays as data members.

```
#include <iostream>
using namespace std;
const int 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";
}
void items::remove(void) // delete a specified item
{
     int a;
     cout<<"\n 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<<"\n code      price\n";
     for(int i=0; i<count; i++)
     {
          cout<<"\n"<<itemcode[i]
              <<"          "<<itemprice[i];
     }
     cout<<"\n ";
}

//==========================================
int main()
{
items order;
order.cnt();
int x;
do
     {
          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 <<"\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<<"\n error in input; try again\n";
          }

     } while (x!=5);
     return 0;
}
```

The sample output of above code is

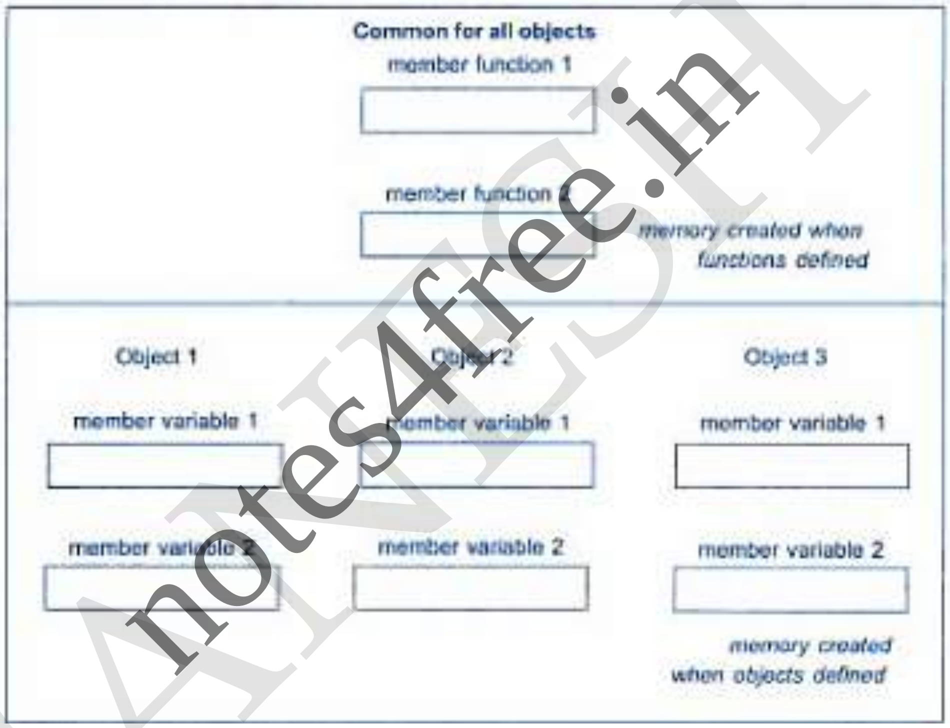
```
you can do the following; enter appropriate number
1 : add an item
2 : display total value
3 : delete an item
4 : display all items
5 : quit
what is your option? 1
enter item code : 111
enter item cost:100

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 code222

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
```

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.

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 above Fig.

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.

```
class items
{
    static int num;
    ..............
    ..............
public:
    ..............
    ..............
};
int items:: num; // definition of static data members
```

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.***

Example:

```
#include <iostream>
using namespace std;
class item
{
    static int count;
    int number;
    public:
    void getdata(int a)
```

```
     {
     number=a;
     count++;
     }
     void getcount(void)
     {
          cout <<"Count: ";
          cout << count << "\n";
     }
};
int item::count;
int main()
{
     item a, b, c; //count is initialized to zero
     a.getcount ();
     b.getcount();
     c.getcount ();
     a.getdata(100); // get data into object a
     b.getdata (200);// get data into object b
     c.getdata(300);// get //display countdata into object c
     cout << "After reading data"<< "\n";
     a.getcount ();//display count
     b.getcount();
     c.getcount ();
     return 0;
}
```

Output:

```
Count: 0
Count: 0
Count: 0
After reading data
Count: 3
Count: 3
Count: 3
```

Static Member Functions

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

• A static function can have access to only other static members (functions or variables) declared in the same class.

• A static member function can be called using the class name (instead of its objects) as follows:

```
class_name :: function_name();
```

Example:

```
class items
{
    static int num;
    ..............
    ..............
public:
    ..............
    ..............
    static void showcount()
    {
        cout<< num;
    }
};

int items:: num; // definition of static data members

int main()
{
    ..............
    ..............
    items :: showcount();
    ..............
    ..............
}
```

Example :

```
#include <iostream>
using namespace std;
class static_type
{
    static int i; //static data member
public:
    static void init(int x) { i=x; } //static member function
    void show() { cout<<i; }
};
int static_type::i; // define static i
int main()
{
    static_type::init(100);//initialise static data before object
                                        creation
    static_type x;
    x.show( ); //displays 100
    return 0;
}
```

Example 2:

```
#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();
    test t3;
    t3.setcode();
    test::showcount();
    t1.showcode();
    t2.showcode();
    t3.showcode();
    return 0;
}
```

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[10];
    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:

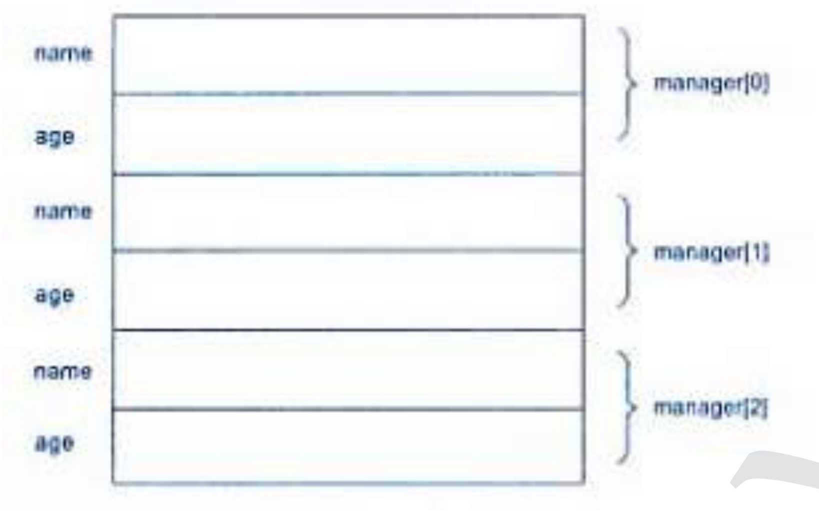
```
employee manager[3] ; //array of manager
employee foreman[15] ; //array of foremen
employee worker[75] ; //array of worker
```

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[i].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 following 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.



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

class employee
{
     char name[10];
     float age;
     public:
     void getdata(void);
     void putdata(void);
};
void employee::getdata(void)
{
     cout <<"\nEnter name: ";
     cin >> name;
     cout <<"\nEnter age: ";
     cin >> age;
}
void employee::putdata(void)
{
     cout <<"\nName: "<<name;
     cout <<"\nAge: "<<age;
}

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(int i=0; i<size; i++)
        {
            cout <<"\nManager"<<i+1<<"\n";
            manager[i].putdata();
        }
        return 0;
    }
```

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. (pass by vale)

• Only the address of the object is transferred to the function. (pass by reference)

```
#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 object as arguments
};
void time::sum(time t1, time t2)
{
    minutes = t1.minutes +t2.minutes;
```

```
		hours= minutes/60;
		minutes = minutes%60;
		hours= hours + t1.hours+ t2.hours;
	}
	int main()
	{
		time t1,t2,t3;
		t1.gettime(2,45); // get t1
		t2.gettime(3,30); // get t2
		t3.sum(t1,t2); //t3=t1+t2
		cout<<"t1 = "; t1.puttime(); // display t1
		cout<<"t2 = "; t2.puttime(); // display t2
		cout<<"t3 = "; t3.puttime(); // display t3
		return 0;
	}
```

Friendly Functions

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.

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

class base
{
	int val1,val2;
public:
	void get()
	{
		cin>>val1>>val2;
	}
	friend float mean(base ob);
};

float mean(base ob)
{
	return float(ob.val1 + ob.val2)/2;
}
```

```
int main()
{
     base obj;
     obj.get();
     cout<<"\n Mean value is : "<<mean(obj);
     return 0;
}
```

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.
- Member functions of one class can be friend function of another class.

```
class X
{
     int fun();
};
class Y
{
     friend int X::fun();
};
```

- 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 follows:

```
class Z
{
     friend class X;
};
```

- Consider following example

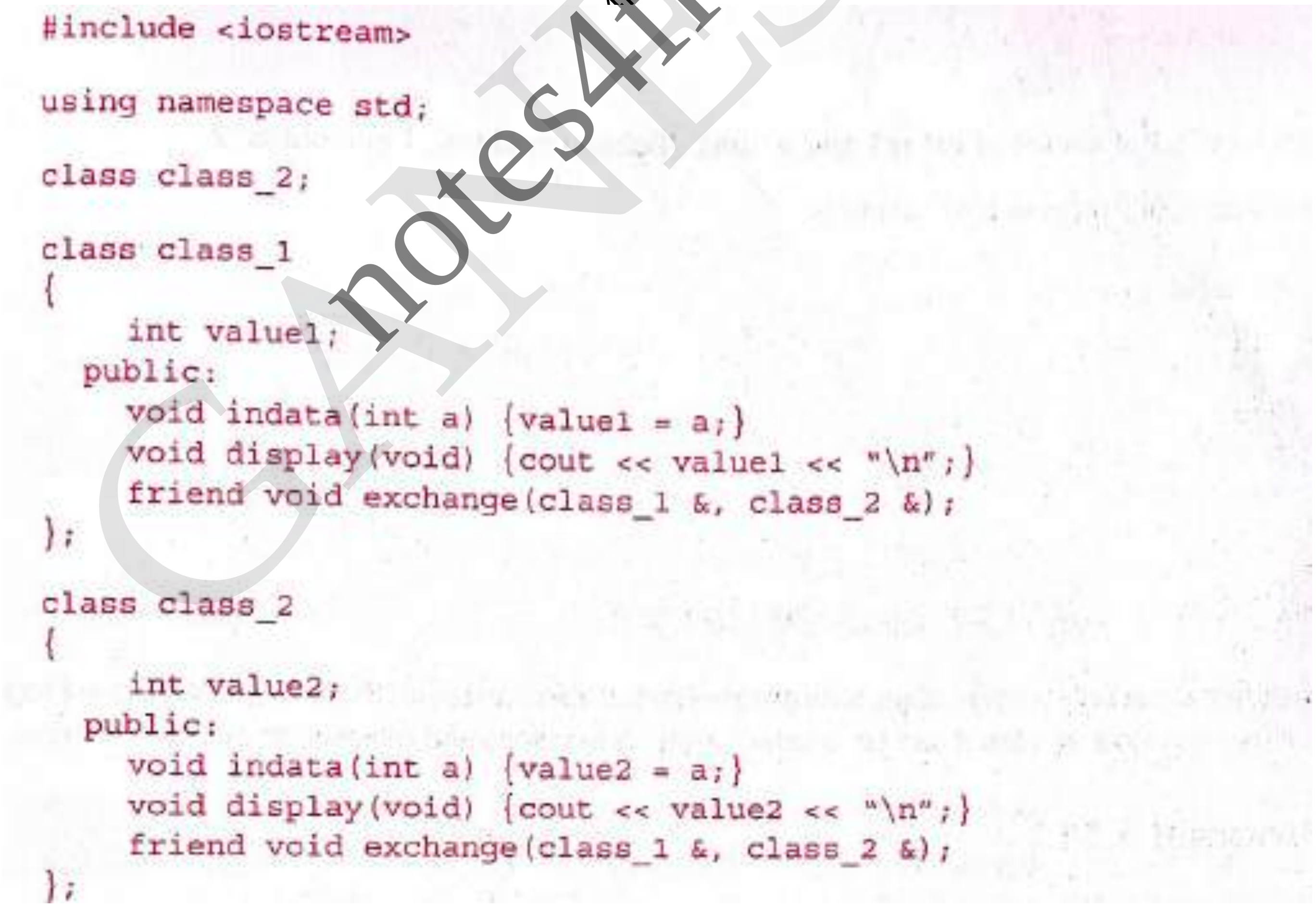
```
class Y;// forward declaration
class X
{
```

```
                friend int fun(X,Y);
        };
        class Y
        {
                friend int fun(X,Y);
        };

        int fun(X x, Y y)
        {
                ........;
        }
```

• 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.

// write a C++ prog using friend function to exchange the private values of the two 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)
{
     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;
}
```

Returning Objects

A function cannot only receive objects as arguments but also can return them.

```
X fun(X x, X y)
{
    X z;
    z= x+y;
    return z;
}
```

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.

Pointers to Objects

```
#include <iostream>
using namespace std;
class cl {
     int i;
public:
     int get_i() { return i; }
};
int main()
{
     cl ob, *p;
     p = &ob; // get address of ob
     cout << p->get_i(); // use -> to call get_i()
     return 0;
}
```

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 ::* p = &A :: m;
```

The p 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".

```
int *p = &m; // wont 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 **p** 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 **p** as follows:

```
cout << a.*p; // display
cout << a.m; //same as obave
```

Now, look at the following code:

```
ap = &a; // ap is pointer to object a
cout << ap -> *p; // display m
cout << ap -> m; // some 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 ***p** 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. Example:

```
#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)(10,40);
                cout << "SUM = "<< sum(n) << "\n";
        return 0;
        }
```

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
        {
                .......
                .......
        };
        .......
        student s1(a); // create object
        .......
}
```

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

```
38    return 0;
39}
```

output

```
Enter consumer name & unit consumed :sattar 200
Name                Charge

sattar              210

Press o for exit / press 1 to input again :1

Enter consumer name & unit consumed :santo 300

Nmae           Charge

santo          290

Press o for exit / press 1 to input again : 0
```

Chapter 4

Review Questions

4.1: **State whether the following statements are TRUE or FALSE.**
(a) A function argument is a value returned by the function to the calling program.
(b) When arguments are passed by value, the function works with the original arguments in the calling program.
(c) When a function returns a value, the entire function call can be assigned to a variable.
(d) A function can return a value by reference.
(e) When an argument is passed by reference, a temporary variable is created in the calling program to hold the argument value.
(f) It is not necessary to specify the variable name in the function prototype.

Ans:

(a) FALSE	(d) TRUE
(b) FALSE	(e) FALSE
(c) TRUE	(f) TRUE

4.2: **What are the advantages of function prototypes in C++?**

Ans: Function prototyping is one of the major improvements added to C++ functions. The prototype describes the function interface to the compiler by giving details such as the number and type of arguments and the type of return values.

4.3: **Describe the different styles of writing prototypes.**

Ans:
General form of function prototyping :
return_type function_name (argument_list)

Example :
int do_something (void);
float area (float a, float b);
float area (float, float);

4.4: **Find errors, if any, in the following function prototypes.**
(a) float average(x,y);
(b) int mul(int a,b);
(c) int display(….);
(d) void Vect(int? &V, int & size);
(e) void print(float data[], size = 201);

Ans:

No.	Error	Correction
(a)	Undefined symbol x, y	float average (float x, floaty)
(b)	Undefined symbol b	int mul (int a, int b);
(c)	No error	
(d)	invalid character in variable name	void vect (int &v, int &size);
(e)	Undefined symbol 's'	void print (float data [ ], int size = 20);

4.5: **What is the main advantage of passing arguments by reference?**

Ans: When we pass arguments by reference, the formal arguments in the called function become aliases to the 'actual' arguments in the calling function.

4.6: **When will you make a function inline? Why?**

Ans: When a function contains a small number of statements, then it is declared as inline function.

By declaring a function inline the execution time can be minimized.

4.7: **How does an inline function differ from a preprocessor macro?**

Ans: The macos are not really functions and therefore, the usual error checking does not occur during compilation. But using inline-function this problem can be solved.

4.8: **When do we need to use default arguments in a function?**

Ans: When some constant values are used in a user defined function, then it is needed to assign a default value to the parameter.
Example :

```
Float area (float r, float PI = 3.1416)
{
        return PI*r*r;
}
```

4.9: **What is the significance of an empty parenthesis in a function declaration?**

Ans: An empty parentheses implies arguments is void type.

4.10: **What do you meant by overloading of a function? When do we use this concept?**

Ans: Overloading of a function means the use of the same thing for different purposes.
When we need to design a family of functions-with one function name but with different argument lists, then we use this concept.

4.11: **Comment on the following function definitions:**
(a)

```
int *f( )
{
int m = 1;
.....
.....
return(&m);
}
```

(b)

```
double f( )
{
.....
.....
return(1);
}
```

(c)

```
int & f()
{
int n - 10;
.....
.....
return(n);
}
```

Ans:

No.	Comment
(a)	This function returns address of m after execution this function.
(b)	This function returns 1 after execution.
(c)	returns address of n

Debugging Exercises

4.1: **Identify the error in the following program.**

```
#include <iostream.h>
int fun()
{
    return 1;
}
float fun()
{
    return 10.23;
}
void main()
{
    cout <<(int)fun() << ' ';
    cout << (float)fun() << ' ';
}
```

Solution: Here two function are same except return type. Function overloading can be used using different argument type but not return type.

Correction : This error can be solved as follows :

```
#include<iostream.h>

int fun()
{
   return 1;
}
float fun1()
{
   return 10.23;
}

void main()
{
    cout<<fun()<<"  ";
    cout<<fun1()<<"  ";
}
```

4.2: **Identify the error in the following program.**

```
#include <iostream.h>
void display(const Int constl=5)
{
  const int const2=5;
  int arrayl[constl];
  int array2[const2];
  for(int 1=0; i<5; 1++)
  {
    arrayl[i] = i;
    array2[i] = i*10;
    cout <<arrayl[i]<< ' ' << array2[i] << ' ' ;
  }
}
void main()
{
  display(5);
}
```

Solution:

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

```
void display()
{
    const int const1=5;
    const int const2=5;
    int array1[const1];
    int array2[const2];

      for(int i=0;i<5;i++)
      {
         array1[i]=i;
         array2[i]=i*10;
         cout<<array1[i]<<"  "<<array2[i]<<"  ";
      }
}

void main()
{
   display();
}
```

4.3: **Identify the error in the following program.**

```
#include <iostream.h>
int gValue=10;
void extra()
{
    cout << gValue << ' ';
}
void main()
{
   extra();
   {
   int gValue = 20;
   cout << gValue << ' ';
   cout << : gValue << ' ';
   }
}
```

Solution:
Here cout << : gvalue << " "; replace with cout <<::gvalue<< " ";

```
#include <iostream.h>
int gValue=10;
void extra()
{
     cout << gValue << ‘ ‘;
}
void main()
{
    extra();
    {
    int gValue = 20;
    cout << gValue << ‘ ‘;
    cout <<::gvalue<< " ";
    }
}
```

4.4: **Find errors, if any, in the following function definition for displaying a matrix: void display(int A[][], int m, int n)**

```
{
  for(1=0; i<m; i++)
  for(j=o; j<n; j++)
   cout<<" "<<A[i][j];
   cout<<"\n";
}
```

Solution:
First dimension of an array may be variable but others must be constant.

Here int A [ ] [ ] replace by the following code:
int A [ ] [10];
int A[10] [10];
int A[ ] [size];
int A [size] [size];

Where const int size = 100;
any other numerical value can be assigned to size.

Programming Exercises

4.1: **Write a function to read a matrix of size m*n from the keyboard.**

Solution:

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

void matrix(int m,int n)
```

```
{
    float **p;
    p=new float*[m];
    for(int i=0;i<m;i++)
    {
        p[i]=new float[n];
    }
    cout<<" Enter "<<m<<"by"<<n<<" matrix elements one by one "<<endl;
    for(i=0;i<m;i++)
    {
        for(int j=0;j<n;j++)
        {
            float value;
            cin>>value;
            p[i][j]=value;
        }
    }
    cout<<" The given matrix is :"<<endl;
    for(i=0;i<m;i++)
    {
        for(int j=0;j<n;j++)
        {
            cout<<p[i][j]<<"   ";
        }
        cout<<"\n";
    }
}

int main()
{
    int r,c;
    cout<<" Enter size of matrix : ";
    cin>>r>>c;
    matrix(r,c);
    return 0;
}
```

output

```
Enter size of matrix : 3   4

Enter 3 by 4 matrix elements one by one

1    2    3    4

2    3    4    5

3    4    5    6

The given matrix is :
```

```
1   2   3   4

2   3   4   5

3   4   5   6
```

4.2: **Write a program to read a matrix of size m*n from the keyboard and display the same on the screen using function.**

Solution:

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

void matrix(int m,int n)
{
   float **p;s
   p=new float*[m];
   for(int i=0;i<m;i++)
   {
      p[i]=new float[n];
   }
   cout<<" Enter "<<m<<" by "<<n<<" matrix elements one by one "<<endl;
   for(i=0;i<m;i++)
   {
      for(int j=0;j<n;j++)
      {
         float value;
         cin>>value;
         p[i][j]=value;
      }
   }
   cout<<" The given matrix is :"<<endl;
   for(i=0;i<m;i++)
   {
      for(int j=0;j<n;j++)
      {
         cout<<p[i][j]<<"   ";
      }
      cout<<"\n";
   }
}

int main()
{
   int r,c;
   cout<<" Enter size of matrix : ";
```

```
    cin>>r>>c;
    matrix(r,c);
    return 0;
}
```

output

```
Enter size of matrix : 4   4

Enter 4 by 4 matrix elements one by one

1    2    3    4    7

2    3    4    5    8

3    4    5    6    9

The given matrix is :

1    2    3    4    7

2    3    4    5    8

3    4    5    6    9
```

4.3: **Rewrite the program of Exercise 4.2 to make the row parameter of the matrix as a default argument.**

Solution:

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

void matrix(int n,int m=3)
{
    float **p;
    p=new float*[m];
    for(int i=0;i<m;i++)
    {
        p[i]=new float[n];
    }
    cout<<" Enter "<<m<<" by "<<n<<" matrix elements one by one "<<endl;
    for(i=0;i<m;i++)
    {
        for(int j=0;j<n;j++)
        {
            float value;
            cin>>value;
            p[i][j]=value;
        }
    }
    cout<<" The given matrix is :"<<endl;
    for(i=0;i<m;i++)
    {
        for(int j=0;j<n;j++)
        {
            cout<<p[i][j]<<"    ";
        }
        cout<<"\n";
    }
}

int main()
{
    int c;
    cout<<" Enter column of matrix : ";
    cin>>c;
    matrix(c);
    return 0;
}
```

output

```
Enter column of matrix  :  3

Enter 3 by 3 matrix elements one by one

1    2    3

2    3    4

3    4    5

The given matrix is :

1    2    3

2    3    4

3    4    5
```

4.4: **The effect of a default argument can be alternatively achieved by overloading. Discuss with examples.**

Solution:

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

void matrix(int m,int n)
{
   float **p;
   p=new float*[m];
   for(int i=0;i<m;i++)
   {
      p[i]=new float[n];
   }
   cout<<" Enter "<<m<<"by"<<n<<" matrix elements one by one "<<endl;
   for(i=0;i<m;i++)
   {
      for(int j=0;j<n;j++)
      {
         float value;
         cin>>value;
         p[i][j]=value;
      }
   }
   cout<<" The given matrix is :"<<endl;
   for(i=0;i<m;i++)
   {
      for(int j=0;j<n;j++)
      {
         cout<<p[i][j]<<"   ";
      }
      cout<<"\n";
   }
}
void matrix(int m,long int n=3)
{
   float **p;
   p=new float*[m];

        for(int i=0;i<m;i++)
   {
      p[i]=new float[n];
   }
   cout<<" Enter "<<m<<" by "<<n<<" matrix elements one by one "<<endl;
   for(i=0;i<m;i++)
   {
      for(int j=0;j<n;j++)
      {
         float value;
         cin>>value;
         p[i][j]=value;
```

```
       }
   }
   cout<<" The given matrix is :"<<endl;
   for(i=0;i<m;i++)
   {
      for(int j=0;j<n;j++)
      {
         cout<<p[i][j]<<"    ";
      }
      cout<<"\n";
   }
}

int main()
{
   int r;
   cout<<" Enter row of matrix : ";
   cin>>r;
   matrix(r);
   return 0;
}
```

output

```
Enter column of matrix  :  2

Enter 2 by 3 matrix elements one by one

1    0    1

0    2    1

The given matrix is :

1    0    1

0    2    1
```

4.5: **Write a macro that obtains the largest of the three numbers.**

Solution:

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

float  large(float a,float b,float c)
{
    float largest;
    if(a>b)
    {
        if(a>c)
            largest=a;
        else
            largest=c;
    }
    else
    {
        if(b>c)
            largest=b;
        else
            largest=c;
    }
    return largest;
}

int main()
{
    float x,y,z;
    cout<<" Enter three values : ";
    cin>>x>>y>>z;
    float largest=large(x,y,z);
    cout<<" large = "<<largest<<endl;
    return 0;
}
```

output

```
Enter three values : 4  5  8

large = 8
```

4.6: **Redo Exercise 4.16 using inline function. Test the function using a main function.**

Solution:
Blank

4.7: **Write a function power() to raise a number m to power n. The function takes a double value for m and int value for n and returns the result correctly. Use a default value of 2 for n to make the function to calculate the squares when this argument is omitted. Write a main that gets the values of m and n from the user to test the function.**

Solution:

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

long double power(double m,int n)
{
   long double mn=pow(m,n);
   return mn;
}
long double power(double m,long int n=2)
{
   long double mn=pow(m,n);
   return mn;
}
int main()
{
   long double mn;
   double m;
   int n;

   cout<<" Enter the value of m & n"<<endl;
   cin>>m>>n;
   mn=power(m,n);
   cout<<" m to power n : "<<mn<<endl;
   mn=power(m);
   cout<<" m to power n : "<<mn<<endl;
   return 0;
}
```

output

```
Enter the value of m & n

12    6

m to power n : 2985984

m to power n: 144
```

4.6: **Redo Exercise 4.16 using inline function. Test the function using a main function.**

Solution:

Blank

4.7: **Write a function power() to raise a number m to power n. The function takes a double value for m and int value for n and returns the result correctly. Use a default value of 2 for**

n to make the function to calculate the squares when this argument is omitted. Write a main that gets the values of m and n from the user to test the function.

Solution:

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

long double power(double m,int n)
{
   long double mn=pow(m,n);
   return mn;
}
long double power(double m,long int n=2)
{
   long double mn=pow(m,n);
   return mn;
}
int main()
{
   long double mn;
   double m;
   int n;

   cout<<" Enter the value of m & n"<<endl;
   cin>>m>>n;
   mn=power(m,n);
   cout<<" m to power n : "<<mn<<endl;
   mn=power(m);
   cout<<" m to power n : "<<mn<<endl;
   return 0;
}
```

output

```
Enter the value of m & n

12    6

m to power n : 2985984

m to power n: 144
```

4.8: **Write a function that performs the same operation as that of Exercise 4.18 but takes an int value for m. Both the functions should have the same name. Write a main that calls both the functions. Use the concept of function overloading.**

Solution:

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

long double power(int m,int n)
{
   long double mn= (long double)pow(m,n);
   return mn;
}
long double power(int m,long int n=2)
{
   long double mn=(long double)pow(m,n);
   return mn;
}
int main()
{
   long double mn;
   int m;
   int n;

   cout<<" Enter the value of m & n"<<endl;
   cin>>m>>n;
   mn=power(m,n);
   cout<<" m to power n : "<<mn<<endl;
   mn=power(m);
   cout<<" m to power n : "<<mn<<endl;
   return 0;
}
```

output

```
Enter the value of m & n

15    16

m to power n : 6.568408e+18

m to power n: 225
```

Chapter 5

Review Questions

5.1: **How do structures in C and C++ differ?**

Ans:
C structure member functions are not permitted but in C++ member functions are permitted.

5.2: **What is a class? How does it accomplish data hiding?**

Ans:
A class is a way to bind the data and its associated functions together. In class we can declare a data as private for which the functions accomplish data–outside the class can not access the data and thus if hiding.

5.3: **How does a C++ structure differ from a C++ class?**

Ans:
Initially (in C) a structure was used to bundle different of data types together to perform a particular functionality C++ extended the structure to contain functions also. The difference is that all declarations inside a structure are default public.

5.4: **What are objects? How are they created?**

Ans:
Object is a member of class. Let us consider a simple example. int a; here a is a variable of int type. Again consider class fruit.
{
}
here fruit is the class-name. We can create an object as follows:
fruit mango;
here mango is a object.

5.5: **How is a member function of a class defined?**

Ans:
member function of a class can be defined in two places:
* Outside the class definition.
* Inside the class definition.

Inside the class definition : same as other normal function.

Outside the class definition : general form:
return-type class-name : function-name (argument list)

```
{
function body
}
```

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.**

Ans:
Yes, We can distinguish them during calling to main ( ) function. The following example illustrates this:

```
#include<iostream.h>
void f()
{
cout<<"Outside Of class \n";
}

class santo
{
public:
void f()
{
            cout<<"Inside of class \n";
}
};

void main()
{
f();  // outside f() is calling.
santo robin;
robin.f();  // Inside f() is calling.
}
```

5.7: **Describe the mechanism of accessing data members and member functions in the following cases:**
(a) Inside the main program.
(b) Inside a member function of the same class.
(c) Inside a member function of another class.

Ans:
(a) Using object and dot membership operator.
(b) Just like accessing a local variable of a function.
(c) Using object and dot membership operator.

The following example explains how to access data members and member functions inside a member function of another class.

```
#include<iostream.h>

class a
{
     public:
            int x;
      void display()
       {
           cout<<"This is class a \n";
            x=111;
       }
};

class b
{
     public:
      void display()
      {
          a  s;
          cout<<" Now member function 'display()' of class a is calling from class b \n";
          s.display();
          cout<<" x = "<<s.x<<"\n";
      }
};

void main()
{
         b  billal; // billal is a object of class b.
          billal.display();
}
```

5.8: **When do we declare a member of a class static?**

Ans:
When we need a new context of a variable the n we declare this variable as static.

5.9: **What is a friend function? What are the merits and demerits of using friend functions?**

Ans:
A function that acts as a bridge among different classes, then it is called friend function.

Merits :
We can access the other class members in our class if we use friend keyword. We can access the members without inheriting the class.

demerits :
Maximum size of the memory will occupied by objects according to the size of friend members.

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

Ans:
(a) FALSE
(b) TRUE
(c) FALSE

*A function designed as public can be accessed like any other ordinary functions from the member function of same class.
(d) TRUE
(e) TRUE
(f) FALSE
(g) FALSE
(h) TRUE
(i) FALSE
(j) FALSE

Debugging Exercises

5.1: **Identify the error in the following program**

```
#include <iosream.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);
}
```

Solution:
Void setvalue (in w, int l) function must be public.

5.2: **Identify the error in the following program**

```
#include <iosream.h>
class Room
{
   int width, int length;
   void setValue(int w, int h)
   {
        width = w;
        length = h;
    }
};
void main()
{
Room objRoom;
objRoom.width=12;
}
```

Solution:
Void setvalue (int w, int l) function must be public.

5.3: **Identify the error in the following program**

```
#include <iosream.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 objlteml;
  Item objltem2;

  cout << objlteml.getCount() << ' ';
  cout << objltem2.getCount() << ' ';

  cout << objlteml.getCountAddress() << ' ';
  cout << objltem2.getCountAddress() << ' ';
}
```

Solution:

```

int* getCountAddress ( )
 {
   return &count;
 }
```

Note: All other code remain unchanged.

5.4: **Identify the error in the following program**

```
#include <iosream.h>
class staticfunction
{
  static int count;
public:
  static void setCounto()
   {
    count++;
   }
  void displayCount()
 {
   cout << count;
 }
};
int staticFunction::count = 10;
void main()
{
    staticFunction obj1;
    obj1setcount(5);
    staticFunction::setCount();
    obj1.displayCount();
}
```

Solution:
setCount ( ) is a void argument type, so here obj1.setCount (5); replace with obj1.setcount( );

5.5: **Identify the error in the following program**

```
#include <iosream.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 1)
   {

      1.inches this->inches;
      1.feet += this->feet;
      if(1.inches>12)
   {

      1.inches-=12;
      1.feet++;
   }
   return 1;
}
int getFeet()
{
    return feet;
 }
float getInches()
{
  return inches;
}
};
void main()
{

 Length objLength1;
 Length objLenngth1(5, 6.5);
 objLength1 = objLength1.addLength(objLength2);
 cout << objLenth1.getFeet() << ' ';
 cout << objLength1.getInches() << ' ';
}
```

Solution:
Just write the main function like this:

```
#include<iostream.h>

void main()
{
   Length objLength1;
   Length objLength2(5,6.5);
   objLength1=objLength1.addLenghth(objLenghth2);

   cout<<objLength1.getFeet()<<"  ";
   cout<<objLength1.getInches()<<"  ";
}
```

5.6: **Identify the error in the following program**

```
#include <iosream.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;
}
```

Solution:
Undefined structure Room in main ( ) function.
Correction : Change the main ( ) Function as follow:

```
void main()
{
 Area();
}
```

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 the name and balance.

Write a main program to test the program.

Solution:

```
#include<iostream.h>
#include<iomanip.h>
class bank
{
    char name[40];
    int ac_no;
    char ac_type[20];
    double balance;
public:
    int assign(void);
    void deposite(float b);
    void withdraw(float c);
    void display(void);
};

int bank::assign(void)
{
    float initial;
    cout<<" You have to pay   500 TK to open your account \n"
    <<" You have to store at least 500 TK to keep your account active\n"
    <<"Would you want to open a account????\n"
    <<" If Yes press 1 \n"
    <<" If No press 0 : ";
    int test;
    cin>>test;
    if(test==1)
    {
        initial=500;
        balance=initial;
      cout<<" Enter name ,account number & account type to creat account : \n";
         cin>>name>>ac_no>>ac_type;
    }
    else
    ;// do nothing

    return  test;

}
void bank::deposite(float b)
{
    balance+=b;
}
void bank::withdraw(float c)
{
    balance-=c;
    if(balance<500)
    {
        cout<<" Sorry your balance is not sufficient to withdraw "<<c<<"TK\n"
        <<" You have to store at least 500 TK to keep your account active\n";
          balance+=c;
  }
}
void bank::display(void)
{
  cout<<setw(12)<<"Name"<<setw(20)<<"Account type"<<setw(12)<<"Balance"<<endl;
  cout<<setw(12)<<name<<setw(17)<<ac_type<<setw(14)<<balance<<endl;
}

int main()
{
  bank account;

  int  t;
  t=account.assign();
  if(t==1)
  {
    cout<<" Would you want to deposite: ?"<<endl
    <<"If NO press 0(zero)"<<endl
    <<"If YES enter deposite ammount :"<<endl;
    float dp;
    cin>>dp;
    account.deposite(dp);
    cout<<" Would you want to withdraw : ?"<<endl
    <<"If NO press 0(zero)"<<endl
    <<"If YES enter withdrawal ammount :"<<endl;
    float wd;
    cin>>wd;
    account.withdraw(wd);
    cout<<" see details :"<<endl<<endl;
    account.display();
  }
      else if(t==0)
  cout<<" Thank you ,see again\n";
  return 0;
}
```

output

```
You have to pay 500 TK to open your account
You have to store at least 500 TK to keep your account active
Would you want to open a account????
If Yes press 1
If No press 0 : 0
Thank you ,see again
```

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.

Solution:

```
#include<iostream.h>
#include<iomanip.h>
class vector
{
   float *p;
   int size;
public:
   void creat_vector(int a);
   void set_element(int i,float value);
   void modify(void);
   void multiply(float b);
   void display(void);
};

void vector::creat_vector(int a)
{
   size=a;
   p=new float[size];
}
void vector::set_element(int i,float value)
{
   p[i]=value;
}
void vector :: multiply(float b)
{
   for(int i=0;i<size;i++)
      p[i]=b*p[i];
}
void vector:: display(void)
{
    cout<<"p["<<size<<"] = ( ";
    for(int i=0;i<size;i++)
    {
       if(i==size-1)
        cout<<p[i];
       else
       cout<<p[i]<<" , ";

    }
    cout<<")"<<endl;
}

void vector::modify(void)
{
   int i;
   cout<<" to edit a given element enter position of the element : ";
   cin>>i;
   i--;
   cout<<" Now enter new value of "<<i+1<<"th  element : ";
   float v;
   cin>>v;
   p[i]=v;
   cout<<" Now new contents : "<<endl;
   display();

   cout<<" to delete an element enter position of the element :";
   cin>>i;
   i--;

   for(int j=i;j<size;j++)
   {
      p[j]=p[j+1];
   }
   size--;
   cout<<" New contents : "<<endl;
   display();
}

int main()
{
   vector santo;
   int s;
   cout<<" enter size of vector : ";
   cin>>s;
   santo.creat_vector(s);
   cout<<" enter "<<s<<" elements  one by one :"<<endl;
   for(int i=0;i<s;i++)
   {
      float v;
      cin>>v;
      santo.set_element(i,v);
   }
   cout<<" Now contents  :"<<endl;
   santo.display();
   cout<<" to multiply this vector by a scalar quantity enter this scalar quantity : ";
```

```
float m;
cin>>m;
santo.multiply(m);
cout<<" Now contents : "<<endl;
santo.display();
santo.modify();
return 0;
}
```

output

```
enter size of vector : 5

enter 5 elements one by one :

11    22    33    44    55

Now contents p[5] = ( 11  ,  22  ,  33  ,  44  ,  55)

to multiply this vector by a scalar quantity enter this scalar quantity : 2

Now contents :

p[5] = ( 22  ,  44  ,  66  ,  88  ,  110)

to edit a given element enter position of the element : 3

Now enter new value of 3th element : 100

Now new contents :

p[5] = ( 22  ,  44  ,  100  ,  88  ,  110)

to delete an element enter position of the element :2

New contents :

p[4] = ( 22  ,  100  ,  88  ,  110)
```

5.3: **Modify the class and the program of Exercise 5.1 for handling 10 customers.**

Solution:

```
#include<iostream.h>
#include<iomanip.h>
#define size 10
char *serial[size]={" FIRST "," SECOND "," THIRD "," 4th "," 5th "," 6th "," 7th "," 8th ","
```

```
9th ","10th"};

class bank
{
   char name[40];
   int ac_no;
   char ac_type[20];
   double balance;
public:
   int assign(void);
   void deposit(float b);
   void withdraw(float c);
   void displayon(void);
         void displayoff(void);
};

int bank::assign(void)
{
   float initial;
   cout<<" You have to pay   500 TK to open your account \n"
   <<" You have to store at least 500 TK to keep your account active\n"
   <<"Would you want to open a account????\n"
   <<" If Yes press 1 \n"
   <<" If No press 0 : ";
   int test;
   cin>>test;
   if(test==1)
   {
      initial=500;
      balance=initial;
cout<<" Enter name ,account number & account type to create account : \n";
        cin>>name>>ac_no>>ac_type;
   }
   else
   ;// do nothing

   return  test;

}
void bank::deposit(float b)
{
   balance+=b;
}
void bank::withdraw(float c)
{
   balance-=c;
   if(balance<500)
   {
   cout<<" Sorry your balance is not sufficient to withdraw "<<c<<"TK\n"
   <<" You have to store at least 500 TK to keep your account active\n";
            balance+=c;
   }
}
void bank::displayon(void)
{
   cout<<setw(12)<<name<<setw(17)<<ac_type<<setw(14)<<balance<<endl;
}
void bank::displayoff(void)
{       cout<<" Account has not created"<<endl;  }
int main()
{
   bank account[size];
        int  t[10];
   for(int i=0;i<size;i++)
   {
               cout<<" Enter information for "<<serial[i]<<"customer : "<<endl;
      t[i]=account[i].assign();
               if(t[i]==1)
               {
         cout<<" Would you want to deposit: ?"<<endl
         <<"If NO press 0(zero)"<<endl
         <<"If YES enter deposit amount :"<<endl;
         float dp;
         cin>>dp;
         account[i].deposit(dp);
         cout<<" Would you want to with draw : ?"<<endl
         <<"If NO press 0(zero)"<<endl
         <<"If YES enter withdrawal amount :"<<endl;
         float wd;
         cin>>wd;
         account[i].withdraw(wd);
         cout<<endl<<endl;
               }
               else if(t[i]==0)
               cout<<"Thank  you , see again  \n";

   }

    cout<<" see details :"<<endl<<endl;
    cout<<setw(12)<<"Name"<<setw(20)<<"Account type"
              <<setw(12)<<"Balance"<<endl;

   for(i=0;i<size;i++)
   {
                if(t[i]==1)
        account[i].displayon();
               else  if(t[i]==0)
                account[i].displayoff();
   }
   return 0;
}
```

Note: Here we will show output only for **Three** customers. But when you run this program you can see output for 10 customer.

output

```
Enter information for FIRST customer :
You have to pay 500 TR to open your account
You have to store at least 500 TR to keep your account active Would you want to open a
account????
If Yes press 1
If No press 0 : 0
Thank you , see again
Enter information for SECOND customer :
You have to pay 500 TR to open your account
You have to store at least 500 TR to keep your account active Would you want to open a
account????
If Yes press 1
If No press 0 : 1
Enter name ,account number & account type to create account :
Robin 11123 saving
Would you want to deposit: ?
If HO press 0(zero)
If YES enter deposit amount :
0
Would you want to with draw : ?
If HO press 0(zero)
If YES enter withdrawal amount :
0
Enter information for 3rd customer :
You have to pay 500 TK to open your account
You have to store at least 500 TK to keep your account active Would you want to open a
account????
If Yes press 1
If No press 0 : 1
Enter name ,account number & account type to create account :
Billal 11123 fixed
Would you want to deposit: ?
If HO press 0(zero)
If YES enter deposit amount :
1000000
Would you want to with draw : ?
If HO press 0(zero)
If YES enter withdrawal amount :
100000

see details :

Name        Account type        Balance

Account has not created
```

Robin	saving	500
Billal	fixed	900500

5.4: **Modify the class and the 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)**

Solution:

```
#include<iostream.h>
#include<iomanip.h>
#define size 8
class vector
{
    float *p;

public:
    void creat_vector(void);
    void set_element(int i,float value);
    friend void add(vector v1,vector v2);

};
void vector::creat_vector(void)
{
    p=new float[size];
}
void vector::set_element(int i,float value)
{
    p[i]=value;
}
void add(vector v1,vector v2)
{

    float *sum;
    cout<<"sum["<<size<<"] = (";
    sum= new float[size];

    for(int i=0;i<size;i++)
    {
        sum[i]=v1.p[i]+v2.p[i];
        if(i==size-1)
             cout<<sum[i];
        else
            cout<<sum[i]<<" , ";
    }
    cout<<")"<<endl;

}

int main()
{
    vector x1,x2,x3;
    x1.creat_vector();
    x2.creat_vector();
    x3.creat_vector();
    cout<<" Enter "<<size<<" elements of FIRST vector : ";
    for(int i=0;i<size;i++)
    {
        float v;
        cin>>v;
        x1.set_element(i,v);
    }

    cout<<" Enter "<<size<<" elements of SECOND vector : ";
    for(i=0;i<size;i++)
    {
            float v;
        cin>>v;
        x2.set_element(i,v);
    }
    add(x1,x2);

    return 0;
}
```

output

```
Enter 8 elements of FIRST vector :  4  7  8  2  4  3  2  9

Enter 8 elements of SECOND vector :  1  2  3  4  5  6  7  8

sum[8] = (5 , 9 , 11 , 6 , 9 , 9 , 9 , 17)
```

5.5: **Create two classes DM and DB which store the value of distances. DM stores distances in meters and centimeters and DB in feet and inches. Write a program that can read values for the class objects and add one object of DM with another object of DB.**
Use a friend function to carry out the addition operation. The object that stores the results may be a DM object or DB object, depending on the units in which the results are required. The display should be in the format of feet and inches or meters and centimeters depending on the object on display.

Solution:

```
#include<iostream.h>
#define factor 0.3048
class DB;
class DM
{
            float d;
      public:
      void store(float x){d=x;}
      friend void sum(DM,DB);
      void show();
};
class DB
{
     float d1;
     public:
     void store(float y){d1=y;}
     friend void sum(DM,DB);
     void show();
};

void DM::show()
{

    cout<<"\n Distance = "<<d<<" meter or "<<d*100<<" centimeter\n";
}

void DB::show()
{

    cout<<"\n Distance = "<<d1<<" feet or "<<d1*12<<" inches \n";
}
void sum(DM m,DB b)
{

      float sum;

      sum=m.d+b.d1*factor;
      float f;
      f=sum/factor;
      DM m1;
      DB b1;

      m1.store(sum);
      b1.store(f);

     cout<<" press 1 to display result in meter\n"
       <<" press 2 to display result in feet \n"
       <<" What is your option ? : ";
       int test;
       cin>>test;

```

```
          if(test==1)
          m1.show();
          else if(test==2)
          b1.show();


}


int main()
{
     DM dm;
     DB db;
     dm.store(10.5);
     db.store(12.3);
    sum(dm,db);
     return 0;
}
```

output

```
Press 1 to display result in meter
Press 2 to display result in feet
What is your option ? 1
Distance = 14.24904 meter or 1424.903931 centimeter
```

Chapter 6

Review Questions

6.1: **What is a constructor? Is it mandatory to use constructors in a class?**

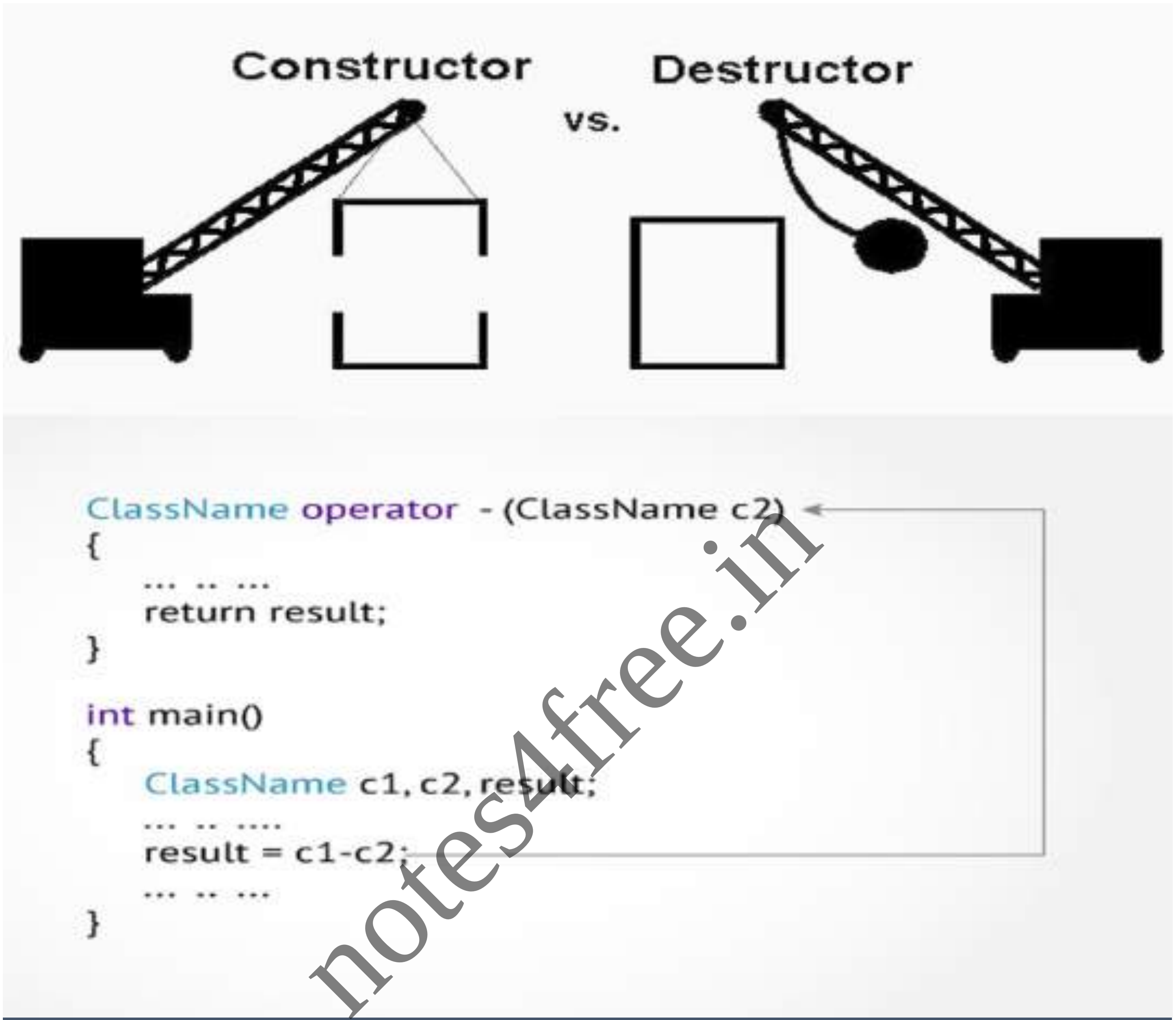
Ans:A constructor is a ‘special’ member function whose task is to initialize the object of its class. It is not mandatory to use constructor in a class.

6.2: **How do we invoke a constructor function?**

Ans:Constructor function are invoked automatically when the objects are created.

6.3: **List some of the special properties of the constructor functions.**

Ans:Special properties of the constructor functions:



```
ClassName operator  - (ClassName c2)
{
    ... .. ...
    return result;
}

int main()
{
    ClassName c1, c2, result;
    ... .. ....
    result = c1-c2;
    ... .. ...
}
```

MODULE -3

Constructors, Destructors and Operator Overloading

GANESH Y
Dept. of ECE RNSIT

MODULE -3
Constructors, Destructors and Operator overloading

SYLLABUS

Constructors, Multiple constructors in a class, Copy constructor, Dynamic constructor, Destructors, Defining operator overloading, Overloading Unary and binary operators, Manipulation of strings using operators (Selected topics from Chap-6, 7 of Text)

Introduction

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

```
A.input();
```

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

```
x.getdata(100,299.95);
```

passes the initial values as argument to the function getdata(), where these values are assigned to the private variables of object x.

All these 'function call' statements are used with the appropriate objects that have already been created. These functions cannot be used to initialize the member variables at the time *of* creation of their objects.

Providing the initial values as described above does not conform with the philosophy of C++ language. We stated earlier that one of the aims of C++ is to create user-defined data types such as **class**, that behave very similar to the built-in types.,

This means that we should be able to initialize a class type variable (object) when it is declared, much the same way as initialization of an ordinary variable. For example,

```
int m = 20;
float x = 5.75;
```

are valid initialization statements for basic data types.

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

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

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)

• 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.

PARAMETERIZED CONSTRUCTORS

The constructors that can take arguments are called ***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 constructor **integer()** may be modified to take arguments as shown below:

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

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

```
integer int1;
```

may not work. We must pass the initial values as arguments to the constructor function when an object is declared. This can be done in two ways:

• By calling the constructor explicitly.

• By calling the constructor implicitly.

The following declaration illustrates the first method:

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

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

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

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

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

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

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

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

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

is illegal.

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

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

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

```
//This program defines a class called Point that stores the x and y
//coordinates of a point.The class uses parameterized constructor for
//initializing the class objects
          #include <iostream>
          class Point
          {
               int x , y;
          public :
               Point (int a, int b) //inline parameterized constructor
          definition
               {
                    x=a;
                    y=b;
               }
               void display ()
               {
               Cout<<"("<<x<<","<<y<<")\n";
               }
          int main ()
          {
               point p1(1, 1); //invokes parameterized constructor
               point p2(5, 10);
               cout<<"Point pl =";
               p1.display();
               cout<<"Point p2 =";
               p2.display();
               return 0;
          }
```

The output of above Program would be:

Point p1= (1,1)
Point p2 = (5,10)

```
          #include <iostream>
          using namespace std;
          class str
          {
               int a_count,e_count,i_count,o_count,u_count;
               char user_str[100];
          public:
               str (char gg[]);
               void count_vowels();
```

```
};
str::str (char gg[])
{
        a_count=0;e_count=0;i_count=0;o_count=0;u_count=0;
        for (int i=0;i<100;i++)
        {
                user_str[i]=gg[i];
                if (user_str[i]=='\0')
                        break;
        }
}
void str::count_vowels()
{
        int i=0;
        do {
                switch (user_str[i])
                {
                        case 'A':
                        case 'a': a_count++;
                                        break;
                        case 'E':
                        case 'e': e_count++;
                                        break;
                        case 'I':
                        case 'i': i_count++;
                                        break;
                        case 'O':
                        case 'o': o_count++;
                                        break;
                        case 'U':
                        case 'u': u_count++;
                                        break;
                }
                i++;
        } while(user_str[i]!='\0');
        cout<<"\n a or A Count "<<a_count;
        cout<<"\n e or E Count "<<e_count;
        cout<<"\n i or I Count "<<i_count;
        cout<<"\n o or O Count "<<o_count;
        cout<<"\n u or U Count "<<u_count;
}
int main ()
{
        char s[]={'G','a','n','e','s','h'};
        str obj1(s);
        obj1.count_vowels();
        str obj2("Ganesh Yernally");
        obj2.count_vowels();
        return 0;
}
```

Multiple Constructors in a Class / Overloaded Constructors

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,

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

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

```
integer I1;
```

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

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

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

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

would invoke the third constructor which copies the values of **I2** into **I3.** In other words, it sets the value of every data element of **I3** to the value of the corresponding data element of **I2.**

```
#include <iostream>
using namespace std;
class complex
{
float x , y ;
public:
      complex() { } // constructor no arg
      complex (float a) {x=y=a;} // constructor-one arg
      complex (float real, float imag) {x=real; y=imag;}// constructor-two arg
      friend complex sum(complex, complex);
      friend void show( complex);
};
complex sum(complex c1, complex c2) //friend
{
      complex c3;
      c3.x = c1.x + c2.x;
      c3.y = c1.y + c2.y;
      return (c3);
}
void show(complex c)
{
      cout<<c.x<<"+j"<<c.y << "\n";
}
int main ()
{
      complex A(2.7, 3.5); // define & initialize
      complex B(1.6);       // define & initialize
      complex C;            // define
      C = sum(A, B);              // sum() is a friend
      cout << "A = ";show (A); // show() is also friend
      cout << "B = ";show (B);
      cout << "C = ";show (C);
// Another way to give initial values
      complex P,Q,R;        // define
      P = complex(2.5,3.9);
      Q = complex(1.6,2.5);
      R = sum(P,Q);
      cout <<"\n";
      cout << "P = ";show (P);
      cout << "Q = ";show (Q);
      cout << "R = ";show (R);
      return 0;
}
```

Output:

```
A = 2.7+j3.5        P = 2.5+j3.9
B = 1.6+j1.6        Q = 1.6+j2.5
C = 4.3+j5.1        R = 4.1+j6.4
```

Let us look at the first constructor again.

```
complex() { }
```

It contains the empty body and does not do anything. We just stated that this is used to create objects without any initial values. Remember, we have defined objects in the earlier examples without using such a constructor.

Why do we need this constructor now? As pointed out earlier, C++ compiler has an ***implicit constructor*** which creates objects, even though it was not defined in the class.

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

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 arguments. When called with no arguments, it becomes a default constructor. When both these forms are used in a class, it causes ambiguity for a statement such as

A a;

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

DYNAMIC INITIALIZATION OF OBJECTS

Class objects can be initialized dynamically too. That is to say, the initial value of an object may be provided during run time.

One advantage of dynamic initialization is that we can provide various initialization formats, using overloaded constructors. This provides the flexibility of using different format of data at run time depending upon the situation.

Consider the long-term deposit schemes working in the commercial banks. The banks provide different interest rates for different schemes as well as for different periods of investment.

Program shown below illustrates how to use the class variables for holding account details and how to construct these variables at run time using dynamic initialization.

```
// Long-term fixed deposit system
#include <iostream>
using namespace std;
class Fixed_deposit
{
        long int P_amount;      // Principal amount
        int    Years;           // Period of investment
        float  Rate;            // Interest rate
        float   R_value;        // Return value of amount
    public:
        Fixed_deposit(){ }
        Fixed_deposit(long int p, int y, float r=0.12);
        Fixed_deposit(long int p, int y, int r);
        void display(void);
 };
Fixed_deposit :: Fixed_deposit(long int p, int y, float r)
{
        P_amount = p;
        Years = y;
        Rate = r;
        R_value = P_amount;
        for(int i = 1; i <= y; i++)
            R_value = R_value * (1.0 + r);
}
```

```
Fixed_deposit :: Fixed_deposit(long int p, int y, int r)
{
     P_amount = p;
     Years = y;
     Rate = r;
     R_value = P_amount;
     for(int i=1; i<=y; i++)
       R_value = R_value*(1.0+float(r)/100);
}
void Fixed_deposit :: display(void)
{
     cout << "\n"
          << "Principal Amount = " << P_amount << "\n"
          << "Return Value     = " << R_value  << "\n";
}
int main()
{
     Fixed_deposit FD1, FD2, FD3;           // deposits created
     long int p;                            // principal amount
int    y;                                   // investment period, years
float   r;                                  // interest rate, decimal form
int    R;                                   // interest rate, percent form
cout << "Enter amount, period,interest rate(in percent)"<<"\n";
cin >> p >> y >> R;
FD1 = Fixed_deposit(p,y,R);
cout << "Enter amount, period, interest rate(decimal form)" << "\n";
cin >> p >> y >> r;
FD2 = Fixed_deposit(p,y,r);
cout << "Enter amount and period" << "\n";
cin >> p >> y;
FD3 = Fixed_deposit(p,y);
cout << "\nDeposit 1";
FD1.display();
cout << "\nDeposit 2";
FD2.display();
cout << "\nDeposit 3";
FD3.display();
return 0;
}
```

The output of Program would be:

```
Enter amount,period,interest rate(in percent)
10000 3 18
Enter amount,period,interest (in decimal form)
10000 3 0.18
Enter amount and period
10000 3
Deposit 1
Principal Amount = 10000
Return Value = 16430.3
Deposit 2
Principal Amount = 10000
Return Value = 16430.3
Deposit 3
Principal 'Amount = 10000
Return Value = 14049.3
```

COPY CONSTRUCTOR

We have used the copy constructor

```
integer (integer &i);
```

as one of the overloaded constructors.

As stated earlier, a copy constructor is used to declare.and initia.lize an object from another object.

For example, the statement

```
integer I2(I1);
```

would define the object **I2** and at the same time initialize it to the values of **I1.** Another form of this statement is

```
integer I2 = I1;
```

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

```
I2 = I1;
```

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

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

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

The output of above Program would be:

```
id of A: 100
id of B: 100
id of C: 100
id of D: 100
```

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

DYNAMIC CONSTRUCTORS

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus resulting in the saving of memory.

Allocation of memory to objects at the time of their construction is known as dynamic construction of objects. The memory is allocated with the help of the new operator.

Program shown below the use of new, in constructors that are used to construct strings in objects.

```
#include <iostream>
#include <string>
using namespace std;
class String
{
     char *name;
     int length;
     public:
     String()                          // constructor-1
     {
       length = 0;
       name = new char[length + 1];
     }
     String(char *s)  // constructor-2
     {
       length = strlen(s);
       name = new char[length + 1];  // one additional
                                     // character for \0
       strcpy(name, s);
     }
     void display(void)
     {cout << name << "\n";}
     void join(String &a, String &b);
};
void String :: join(String &a, String &b)
{
     length = a.length + b.length;
     delete name;
     name = new char[length+1];       // dynamic allocation
     strcpy(name, a.name);
     strcat(name, b.name);
};
int main()
{
     char *first = "Joseph ";
     String name1(first), name2("Louis "), name3 ("Lagrange"),
     s1,s2;
     s1.join(name1, name2);
     s2.join(s1, name3);
     name1.display();
     name2.display();
     name3.display();
     s1.display();
     s2.display();
     return 0;
}
```

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

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

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

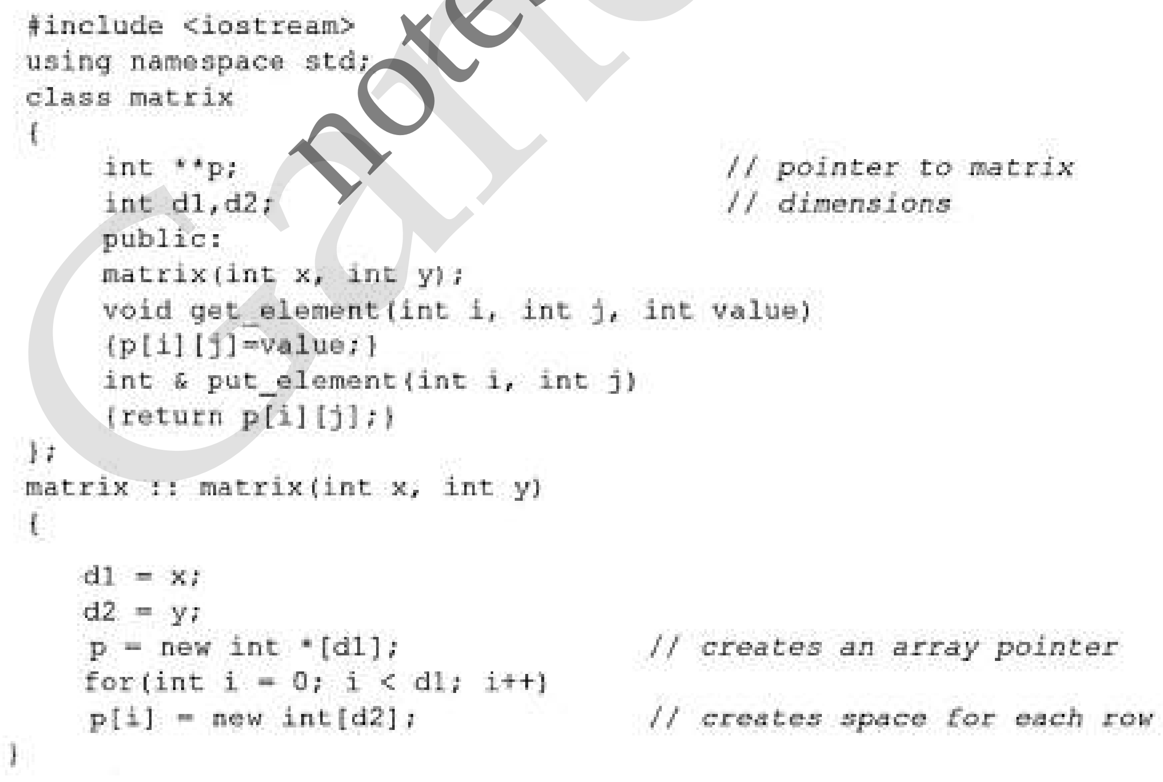
Note that in the function **join(**), **length** and **name** are members of the object that calls the function, while **a.length** and **a.name** are members of the argument object **a.**

The **main(**) function program concatenates three strings into one string. The output is as shown below:

```
Joseph Louis Lagrange
```

CONSTRUCTING TWO-DIMENSIONAL ARRAYS

The following illustrates how to construct a matrix of size m X n.



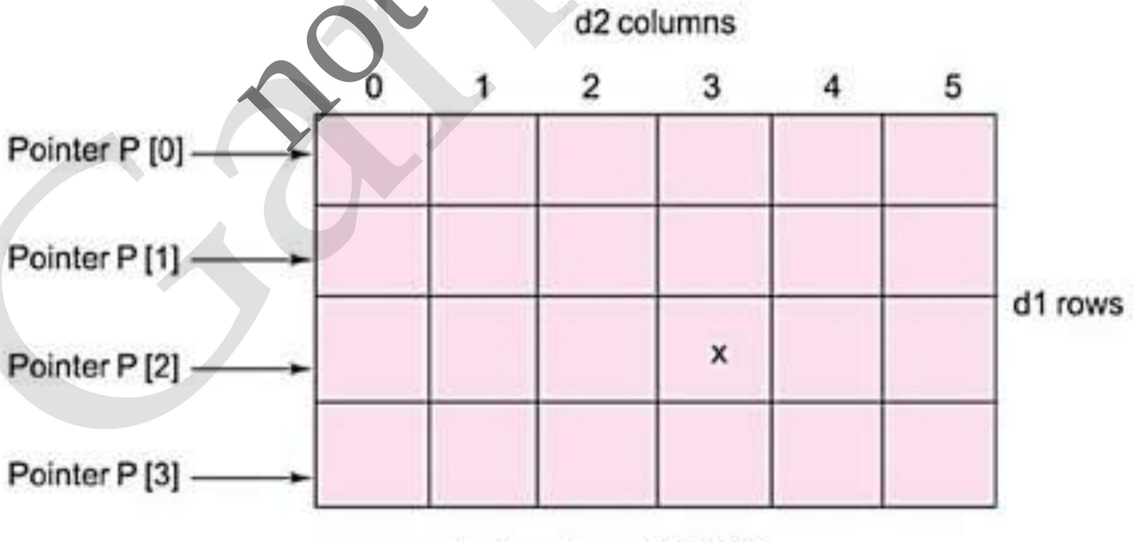
```
#include <iostream>
using namespace std;
class matrix
{
     int **p;                          // pointer to matrix
     int d1,d2;                        // dimensions
     public:
     matrix(int x, int y);
     void get_element(int i, int j, int value)
     {p[i][j]=value;}
     int & put_element(int i, int j)
     {return p[i][j];}
};
matrix :: matrix(int x, int y)
{
    d1 = x;
    d2 = y;
    p = new int *[d1];            // creates an array pointer
    for(int i = 0; i < d1; i++)
    p[i] = new int[d2];           // creates space for each row
}
```

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

Output

```
Enter size of matrix: 3 4
Enter matrix elements row by row
11 12 13 14
15 16 17 18
19 20 21 22
17
17 is the value of the element (1, 2).
```

The constructor first creates a vector pointer to an **int** of size **d1 .** Then, it allocates, iteratively an **int** type vector of size **d2** pointed at by each element **p[i].**



x represents the element P[2] [3]

Thus, space for the elements of a d1 x d2 matrix is allocated from free store as shown above.

const OBJECTS

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

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

Any attempt to modify the values of **m** and **n** will generate compile-time error. Further, a constant object can call only **const** member functions.

As we know, **a const** member is a function prototype or function definition where the keyword const appears after the function's signature. Whenever **const** objects try to invoke **nonconst** member functions, the compiler generates errors.

DESTRUCTORS

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

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

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

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no longer accessible.

It is a good practice to declare destructors in a program since it releases memory space for future use.

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

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

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

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

```
#include<iostream>
using namespace std;
int count=0;
class test
{
    public:
      test()
      {
      count++;
      cout<<"\n\nConstructor Msg: Object number "<<count<<
      "created..";
      }
      ~test()
      {
      cout<<"\n\nDestructor Msg: Object number "<<count<<"
      destroyed..";
      count--;
      }
};
int main()
{
    cout<<"Inside the main block..";
    cout<<"\n\nCreating first object T1..";
    test T1;
    {  //Block 1
       cout<<"\n\nInside Block 1..";
       cout<<"\n\nCreating two more objects T2 and T3..";
       test T2,T3;
       cout<<"\n\nLeaving Block 1..";
    }
    cout<<"\n\nBack inside the main block..";
    return 0;
}
```

```
Inside the main block ..
Creating first object T1 ..
Constructor Msg: Object number 1 created ..

Inside Block 1 ..
Creating two more objects T2 and T3 . .
Constructor Msg: Object number 2 created ..
Constructor Msg: Object number 3 created ..
```

```
Leaving Block 1 ..

Destructor Msg: Object number 3 destroyed .
Destructor Msg : Object number 2 destroyed.
Back inside the main block . .
Destructor Msg: Object number 1 destroyed . .
```

A class constructor is called every time an object is created. Similarly, as the program control leaves the current block the objects in the block start getting destroyed and destructors are called for each one of them.

Note that the objects are destroyed in the reverse order of their creation. Finally, when the main block is exited, destructors are called corresponding to the remaining objects present inside main.

Similar functionality can be attained by **using static data members with constructors and destructors.** We can declare a static integer variable count inside a class to keep a track of the number of its object instantiations.

Being static, the variable will be initialized only once, i.e., when the first object instance is created. During all subsequent object creations, the constructor will increment the count variable by one. Similarly, the destructor will decrement the count variable by one as and when an object gets destroyed.

To realize this scenario, the code in following program will change slightly, as shown below:

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

The primary use of destructors is in freeing up the memory reserved by the object before it gets destroyed. Program shown below demonstrates *how a destructor releases the memory allocated to an object:*

```
#include<iostream>
#include<conio.h>
using namespace std;
class test
{
     int *a;
     public:
     test(int size)
     {
       a = new int[size];
       cout<<"\n\nConstructor Msg: Integer array of size "<<size<<"
       created..";
     }
     ~test()
     {
       delete a;
       cout<<"\n\nDestructor Msg: Freed up the memory allocated for
       integer array";
     }
};
int main()
{
     int s;
     cout<<"Enter the size of the array..";
     cin>>s;
     cout<<"\n\nCreating an object of test class..";
     test T(s);
     cout<<"\n\nPress any key to end the program..";
     getch();
     return 0;
}
```

Output

```
Enter the size of the array .. 5
Creating an object of test class. :
Constructor Msg: Integer array of sizes created ..
Press any key to end the program ..
Destructor Msg: Freed up the memory allocated for integer array
```

Operator Overloading

C++ tries to make the user-defined data types behave in much the same way as the built-in types. For instance, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types.

This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as ***operator overloading***.

Operator overloading provides a flexible option for the creation of new definitions for most of the C++ operators. We can overload (***give additional meaning to***) all the C++ operators except the following:

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

The reason why we cannot overload these operators may be attributed to the fact that these operators take names (example class name) as their operand instead of values, as is the case with other normal operators.

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

Remember, when an operator is overloaded, its original meaning is not lost. For instance, the operator +, which has been overloaded to add two vectors, can still be used to add two integers.

DEFINING OPERATOR OVERLOADING

The general form of an operator function is:

```
return_type classname :: operator op (arglist)
{
        Function body // task defined
}
```

To define an additional task to an operator, we must specify what it means in relationto the class to which the operator is applied.

This is done with the help of a special function, called *operator function,* which describes the task.

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

Operator functions must be either member functions (**non-static**) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators.

This is because the object used to invoke the member function is passed implicitly and therefore is available for the member function. This is not the case with **friend** functions.

```
vector operator+ (vector);                  // vector addition
vector operator- ();                        // unary minus
friend vector operator+ (vector,vector);    // vector addition
friend vector operator- (vector);           // unary minus
vector operator-(vector &a);                // subtraction
int operator==(vector);                     // comparison
friend int operator==(vector,vector);       // comparison
```

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

<u>The process of overloading involves the following steps:</u>

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

Overloaded operator functions can be invoked by expressions such as

```
op x or x op
```

for unary operators and

```
x op y
```

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

```
operator op (x)
```

for **friend** functions.

Similarly, the expression **x op y** would be interpreted as either

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

in case of member functions, or

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

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

OVERLOADING UNARY OPERATORS

Let us consider the unary minus operator. A minus operator when used as a unary, takes just one operand. We know that this operator changes the sign of an operand when applied to a basic data item.

We will see here how to overload this operator so that it can be applied to an object in much the same way as is applied to an **int** or **float** variable. The unary minus when applied to an object should change the sign of each of its data items.

```
#include <iostream>

using namespace std;

class space
{
    int x;
    int y;
    int z;
  public:
    void getdata(int a, int b, int c);
    void display(void);
    void operator-();          // overload unary minus
};
void space :: getdata(int a, int b, int c)
{
    x = a;
    y = b;
    z = c;
}
void space :: display(void)
{
    cout<<"x = "<<x<<" ";
    cout<<"y = "<<y<<" ";
    cout<<"z = "<<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 : x = 10 y = -20 Z = 30
-S : x = - 10 y = 20 Z = - 30

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

Remember, a statement like

S2 = -S1 ;

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

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

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

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

OVERLOADING BINARY OPERATORS

we illustrated, how to add two complex numbers using a friend function. A statement like

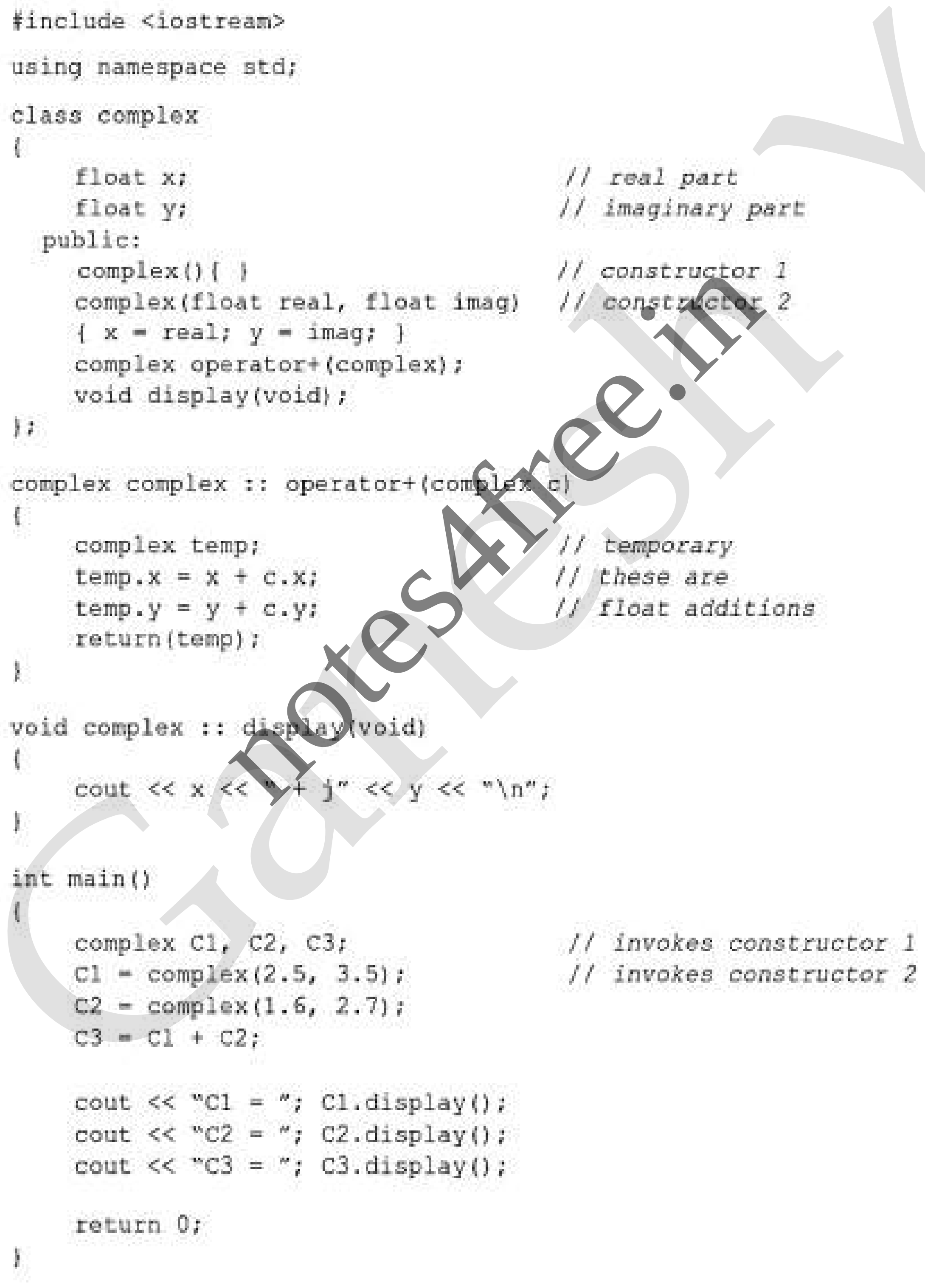
```
C = sum (A,B) ; // functional notation.
```

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

looking expression

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

by overloading the + operator using an operator+( ) function. The Program shown below illustrates how this is accomplished.



```
#include <iostream>

using namespace std;

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(void);
};

complex complex :: operator+(complex c)
{
    complex temp;                         // temporary
    temp.x = x + c.x;                     // these are
    temp.y = y + c.y;                     // float additions
    return(temp);
}

void complex :: display(void)
{
    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();
    cout << "C2 = "; C2.display();
    cout << "C3 = "; C3.display();

    return 0;
}
```

Output

```
C1= 2.5 + j3.5
C2=1.6 + j2.7
C3=4.1 + j6.2
```

Let us have a close look at the function **operator** +( ) and see how the operator overloading is implemented.

```
complex complex:: operator+(complex c)
{
      complex temp;
      temp.x = x + c.x;
      temp.y = y + c.y;
      return (temp);
}
```

We should note the following features of this function:

1. It receives only one **complex** type argument explicitly.

2. It returns a **complex** type value.

3. It is a member function of **complex.**

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

```
C3 =C1+ C2; // invokes operator+() function
```

We know that a member function can .be invoked only by an object of the same class. Here, the object **C1** takes the responsibility of invoking the function and **C2** plays the role of an argument that is passed to the function. The above **invocation** statement is equivalent to

```
C3 = C1.operator+(C2); // usual function call syntax
```

Therefore, in the **operator+()** function, the data members of **C1** are accessed directly and the data members of **C2** (that is passed as an argument) are accessed using the dot operator. Thus, both the objects are available for the function. For example, 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. Figure below shows how this is implemented.

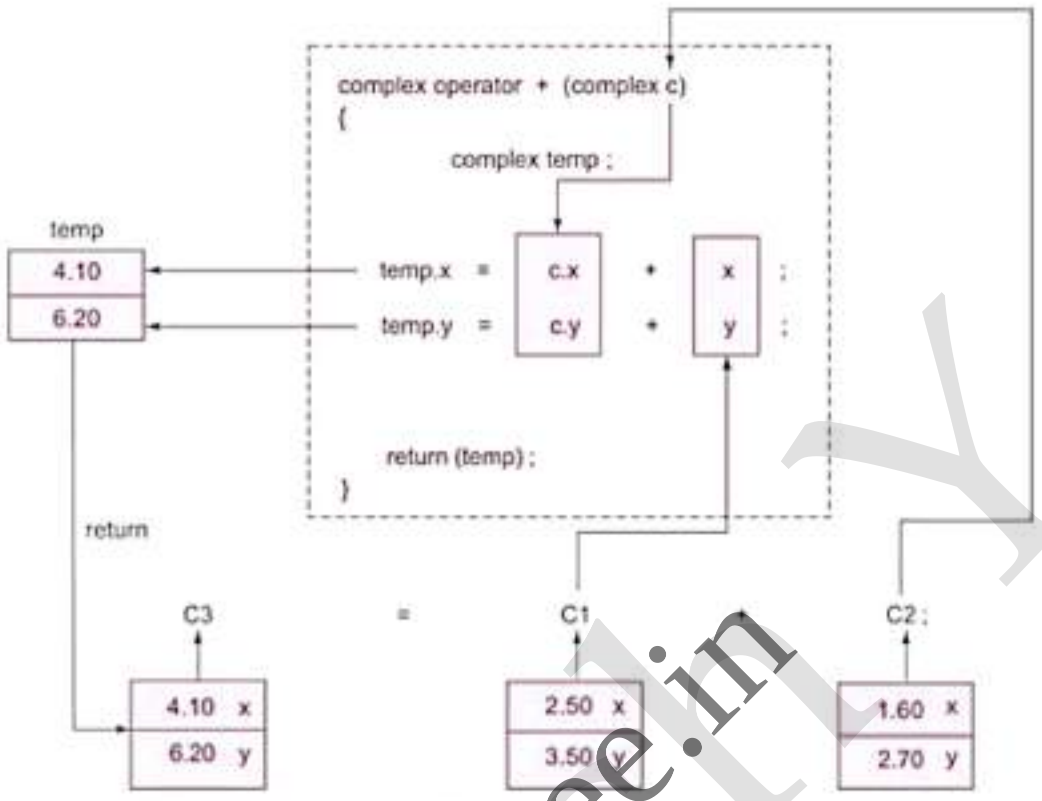


Fig. *Implementation of the overloded + operator*

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

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

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

What does it mean when we use a class name with an argument list? When the compiler comes across a statement like this, it invokes an appropriate constructor, initializes an object with no name and returns the contents for copying into an object.

Such an object is called a temporary object and goes out of space as soon as the contents are assigned to another object. Using *temporary objects* can make the code shorter, more efficient and better to read.

OVERLOADING BINARY OPERATORS USING FRIENDS

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

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

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

```
            friend complex operator+(complex , complex );
```

2. Redefine the operator function as follows:

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

In this case, the statement

`C3 = C1+ C2 ;`

is equivalent to

`C3 = operator +(C1 , C2 );`

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

For instance, consider a situation where we need to use two different types of operands for a binary operator, say, one an object and another a built-in type data as shown below,

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

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

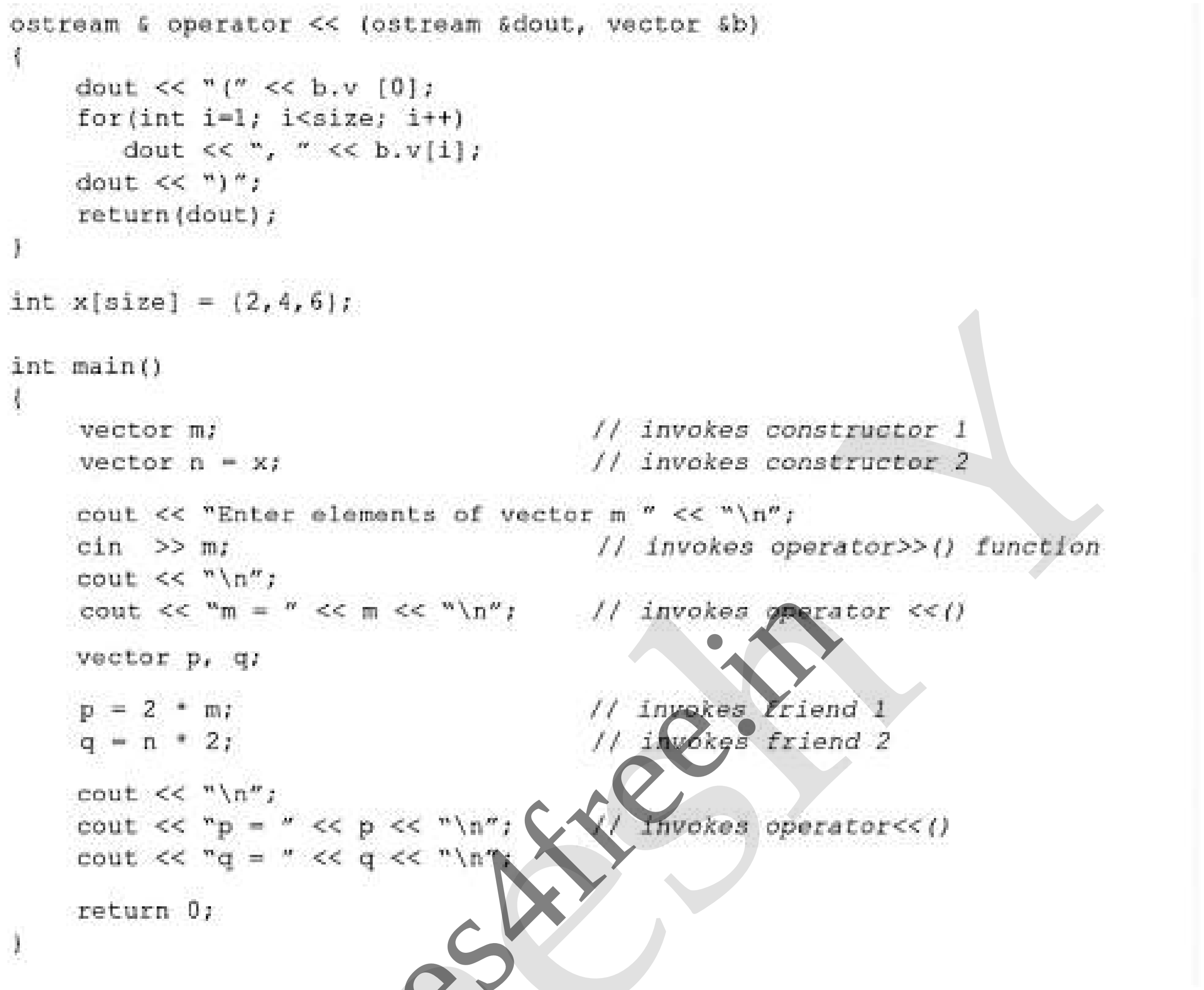
`A = 2 + B; ( or A = 2 * B)`

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

It may be recalled that an object need not be used to invoke a **friend** function but can be passed as an argument. Thus, we can use a friend function with a built-in type data as the *left-hand* operand and an object as the *right-hand* operand.

Program shown below illustrates this, using scalar *multiplication* of a vector. It also shows how to overload the input and output operators >> and <<.

```
#include <iostream.h>

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

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

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

vector operator *(int a, vector b)
{
    vector c;

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

vector operator *(vector b, int a)
{
    vector c;

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

istream & operator >> (istream &din, vector &b)

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



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

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

int main()
{
     vector m;                        // invokes constructor 1
     vector n = x;                    // invokes constructor 2

     cout << "Enter elements of vector m " << "\n";
     cin  >> m;                        // invokes operator>>() function
     cout << "\n";
     cout << "m = " << m << "\n";     // invokes operator <<()

     vector p, q;

     p = 2 * m;                       // invokes friend 1
     q = n * 2;                       // invokes friend 2

     cout << "\n";
     cout << "p = " << p << "\n";     // invokes operator<<()
     cout << "q = " << q << "\n";

     return 0;
}
```

Output

```
Enter elements of vector m
5 10 15
m = (5, 10 , 15)
p = (10 , 20, 30 )
q = ( 4, 8 , 12)
```

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

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

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

`vector() ;`

constructs a vector whose elements are all zero. Thus

```
vector m;
```

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

```
vector (int &x);
```

creates a vector and copies the ele1nents pointed to by the pointer argument x into it. Therefore, the statements

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

```
vector n = x;
```

create n as a vector with co1nponents 2, 4, and 6.

We have used vector variables like m and n in input and output statements just like simple variables. This has been made possible by overloading the operators >> and << using the functions:

```
        friend istream & operator >> (istream &, vector &) ;
        friend ostream & operator << (ostream &, vector &) ;
```

***istream** and **ostream** are classes defined in the **iostream.h** file which has been included in the program.*

MANIPULATION OF STRINGS USING OPERATORS

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

For example, we shall be able to use statements like

```
        string3 = string1 + string2;
        if(string1 >= string2) string = string1;
```

Strings can be defined as class objects which can be then manipulated like the built-in types. Since the strings vary greatly in size, we use *new* to allocate memory for each string and a pointer variable to point to the string array.

Thus we must create string objects that can hold these two pieces of information, namely, length and location which are necessary for string manipulations. A typical string class will look as follows:

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

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

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

class string
{
    char *p;
    int len;
  public:
    string() {len = 0; p = 0;}       // create null string
    string(const char * s);          // create string from arrays
    string(const string & s);        // copy constructor
    ~ string(){delete p;}            // destructor

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

    // <= operator
    friend int operator<=(const string &s, const string &t);
    friend void show(const string s);
};
string :: string(const char *s)
{
    len = strlen(s);
    p = new char[len+1];
    strcpy(p,s);
}


    string :: string(const string & s)
{
    len = s.len;
    p = new char[len+1];
    strcpy(p,s.p);
}
```

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

    if(m <= n) return(1);

    else return(0);
}
void show(const string s)
{
    cout << s.p;
}

int main()
{
    string s1 = "New ";
    string s2 = "York";
    string s3 = "Delhi";
    string string1, string2, string3;
    string1 = s1;
    string2 = s2;
    string3 = s1+s3;

    cout <<   "\nstring1 = "; show(string1);
    cout <<   "\nstring2 = "; show(string2);
    cout <<   "\n";
    cout <<   "\nstring3 = "; show(string3);
    cout <<   "\n\n";
    if(string1 <= string3)
    {
      show(string1);
      cout << " smaller than ";
      show(string3);
```

```
        cout << "\n";
      }
      else
      {
        show(string3);
        cout << " smaller than ";
        show(string1);
        cout << "\n";
      }

      return 0;
  }
```

output

```
string1 = New
string2 = York
string3 = New Delhi
New smaller than New Delhi
```

Chapter 6

Review Questions

1. They should be declared in the public section.
2. They are invoked automatically when the objects are created.
3. They do not have return type, not even void.
4. They cannot be inherited.
5. Like other C++ functions, they can have default arguments.
6. Constructors cannot be virtual.

6.4: **What is a parameterized constructor?**

Ans:The constructors that can take arguments are called parameterized constructors.

6.5: **Can we have more than one constructors in a class? If yes, explain the need for such a situation.**

Ans:Yes, we have when we need to overload the constructor, then we have to do this.

6.6: **What do you mean by dynamic initialization of objects? Why do we need to this?**

Ans:Initializing value of object during run time is called dynamic initialization of objects.
One advantage of dynamic initialization is that we can provide various initialization formats using overloaded constructors.

6.7: **How is dynamic initialization of objects achieved?**

Ans:Appropriate function of a object is invoked during run-time and thus dynamic initialization of object is achieved.
Consider following constructor:
santo (int p, int q, float r);
santo (int p, int q, int r);
It two int type value and one float type value are passed then sant (int p, int q, float r) is invoked.
It three int type value are passed then santo (int p, into q, int r) is invoked.

6.8: **Distinguish between the following two statements:**
time T2(T1);
time T2 = T1;
T1 and T2 are objects of time class.

Ans:
time T2 (T1); ==> explicitly called of copy constructor
time T2 = T1; ==> implicitly called of copy constructor.

6.9: **Describe the importance of destructors.**

Ans:Destructors are important to release memory space for future use.

6.10: **State whether the following statements are TRUE or FALSE.**
(a) Constructors, like other member functions, can be declared anywhere in the class.
(b) Constructors do not return any values.
(c) A constructor that accepts no parameter is known as the default constructor.
(d) A class should have at least one constructor.
(e) Destructors never take any argument.

Ans:
(a) FALSE
(b) TRUE
(c) TRUE
(d) TRUE
(e) TRUE

Debugging Exercises

6.1: **Identify the error in the following program.**

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

    int length;
    int width;
public:
    Room(int 1, int w=0):
        width(w),
        length(1)
    {
    }
};
void main()
{
Room objRooml;
Room objRoom2(12, 8);
}
1
</br>
<span class="a">Solution:</span>Here there is no default constructor, so object could not be
written without any argument.
```

<strong>Correction :</strong>

```
  Void main ( )
    {
                Room Objroom2(12,8);
    }.
```

6.2: **Identify the error in the following program.**

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

    int length;
    int width;
public:
  Room()
  {
        length=0;
        width=0;
  }
Room(int value=8)
 {
       length = width =8;
 }
void display()
 {
      cout<<length<< ' ' <<width;
 }
};
void main()
{
Room objRooml;
objRoom1.display();
```

Solution:Room() and Room(int value=8) Functions are same, so it show Ambiguity error.
Correction : Erase Room() function and then error will not show.

6.3: **Identify the error in the following program.**

```
#include <iostream.h>
class Room
{
    int width;
    int height;
public:
 void Room()
   {
          width=12;
          height=8;
  }
Room(Room& r)
 {
         width =r.width;
         height=r.height;
         copyConsCount++;
 }
void discopyConsCount()
 {
        cout<<copyConsCount;
 }
};
int Room::copyConsCount = 0;
void main()
{
Room objRooml;
Room objroom2(objRoom1);
Room objRoom3 = objRoom1;
Room objRoom4;
objRoom4 = objRoom3;
objRoom4.dicopyConsCount();
}
```

Solution: Just erase "objRoom4 = objRoom3; invalid to call copy constructor." for successfully run.

6.4: **Identify the error in the following program.**

```
#include <iostream.h>
class Room
{
     int width;
     int height;
     static int copyConsCount;
public:
 void Room()
   {
          width=12;
          height=8;
  }
Room(Room& r)
 {
         width =r.width;
```

```
        height=r.height;
        copyConsCount++;
 }
void discopyConsCount()
 {
       cout<<copyConsCount;
 }
};
int Room::copyConsCount = 0;
void main()
{
Room objRooml;
Room objroom2(objRoom1);
Room objRoom3 = objRoom1;
Room objRoom4;
objRoom4 = objRoom3;
objRoom4.dicopyConsCount();
}
```

Solution: Same as 6.3 problem solution.

Programming Exercises

6.1: **Design constructors for the classes designed in Programming Exercise 5.1 through 5.5 of Chapter 5.**

Solution: Study on Constructor and then see solution of chapter 5.

6.2: **Define a class String that could work as a user-defined string type. Include constructors that will enable us to create an uninitialized string:**

String s1; // string with length 0
And also initialize an object with a string constant at the time of creation like
String s2("Well done!");
Include a function that adds two strings to make a third string. Note that the statement
S2 = s1;
will be perfectly reasonable expression to copy one string to another.
Write a complete program to test your class to see that it does the following tasks:
(a) Creates uninitialized string objects.
(b) Creates objects with string constants.
(c) Concatenates two strings properly.
(d) Displays a desired string object.

Solution:

```
#include
#include
class string
{
char *str;
int length;

public:
string()
{
length = 0;
str = new char [length + 1] ;
}
string(char *s);
void concat(string &amp;m,string &amp;n);
string(string &amp;x);
void display();

};
string::string(string &amp;x)
{
length = x.length + strlen(x.str);
str = new char[length + 1];
strcpy(str, x.str);

}
void string:: concat(string &amp;m,string &amp;n)
{
length=m.length+n.length;
delete str;
str=new char[length+1];
strcpy(str,m.str);
strcat(str,n.str);
}
void string:: display()
{
cout&lt;&lt;str&lt;&lt;"\n";
}
string::string(char *s)
{
length = strlen(s);
str = new char[length + 1];
strcpy(str,s);
}

int main()
{
string s1;
string s2(" Well done ");
string s3(" Badly done ");
s2.display();
s1.concat(s2,s3);
s2=s3;
s2.display();
s1.display();
return 0;
}
```

output

```
Well done
Badly done
Well done Badly done
```

6.3: **A book shop maintains the inventory of books that are being sold at the shop. The list includes details such as author, title, price, publisher and stock position. Whenever a customer wants a book, the sales person inputs the title and author and the system searches the list and displays whether it is available or not. If it is not, an appropriate message is displayed. If it is, then the system displays the book details and requests for the number of copies required. If the requested copies are available, the total cost of the requested copies is displayed; otherwise "Required copies not in stock" is displayed.**
Design a system using a class called books with suitable member functions and constructors. Use new operator in constructors to allocate memory space required.

Solution:

```
#include
#include
#include
#include
#include

class book
{
char **author;
char **title;
float *price;
char **publisher;
int *stock_copy;
int size;

public:
book();
void book_detail(int i);
void buy(int i);
```

```
int search();
};

book :: book()
{
size=4;
author=new char*[80];
title=new char*[80];
publisher=new char*[80];

for(int i=0;i&lt;size;i++)
{
author[i]=new char[80];
title[i]=new char[80];
publisher[i]=new char[80];
}
stock_copy=new int[size];
price=new float[size];

title[0]="object oriented programming with c++";
title[1]="programming in ANCI";
title[2]="electronic circuit theory";
title[3]="computer algorithm";

author[0]="balagurusamy";
author[1]="balagurusamy";
author[2]="boyelstade";
author[3]="shahani";

stock_copy[0]=200;
stock_copy[1]=150;
stock_copy[2]=50;
stock_copy[3]=80;

price[0]=120.5;
price[1]=115.75;
price[2]=140;
price[3]=180.5;

}
void book::book_detail(int i)
{
cout&lt;&lt;" *********book detail **********\n";
cout&lt;&lt;setw(12)&lt;&lt;"Title"&lt;&lt;setw(25)&lt;&lt;"Author Name"
&lt;&lt;setw(18)&lt;&lt;"Stock copy\n";
cout&lt;&lt;setw(15)&lt;&lt;title[i]&lt;&lt;setw(16)&lt;&lt;author[i]&lt;&lt;setw(15)
&lt;&lt;stock_copy[i]&lt;&lt;"\n";

}
int book::search()
{
```

```
char name[80],t[80];
cout&lt;&lt;"Enter author name : ";

gets(name);
cout&lt;&lt;"and title of book in small letter : ";
gets(t);

int count=-1;
int a,b;
for(int i=0;i&lt;size;i++)
{

a=strcmp(name,author[i]);
b=strcmp(t,title[i]);
if(a==0 &amp;&amp; b==0)

count=i;

}

return count;
}

void book::buy(int i)
{
if(i&lt;0)
cout&lt;&lt;" This book is not available \n";

else
{
book_detail(i);
cout&lt;&lt;" How many copies of this book is required : ? "; int copy; cin&gt;&gt;copy;
int remaining_copy;
if(copy&lt;=stock_copy[i])
{
remaining_copy=stock_copy[i]-copy;
float total_price;
total_price=price[i]*copy;
cout&lt;&lt;"Total price = "&lt;&lt;total_price&lt;&lt;" TK\n";
}
else
cout&lt;&lt;" Sorry your required copies is not available \n";
}
}

int main()
{
book b1;
int result;

result=b1.search();
```

```
b1.buy(result);
return 0;
}
```

output

```
Enter author name : shahani

and title of book in small latter : computer algorithm

*********book detail *********

Title                          Author Name                 Stock copy

computer algorithm          shahani                        80

How many copies of this book is required : ?  78

Total price = 14079 TK
```

6.4: **Improve the system design in Exercise 6.3 to incorporate the following features:**
(a) The price of the books should be updated as and when required. Use a private member function to implement this.
(b) The stock value of each book should be automatically updated as soon as a transaction is completed.
(c) The number of successful and unsuccessful transactions should be recorded for the purpose of statistical analysis. Use static data members to keep count of transactions.

Solution:

```
#include
#include
#include
#include
#include

class book
{
static int successful,unsuccessful;
char **author;
char **title;
float *price;
char **publisher;
int *stock_copy;
int size;

public:
```

```
book();
void book_detail(int i);
void buy(int i);
int search();
void showtransaction();
void showdetail();
void edit_price();
};
int book::successful=0;
int book::unsuccessful=0;

book :: book()
{
size=5;
author=new char*[80];
title=new char*[80];
publisher=new char*[80];

for(int i=0;i&lt;size;i++)
{
author[i]=new char[80];
title[i]=new char[80];
publisher[i]=new char[80];
}
stock_copy=new int[size];

price=new float[size];

title[0]="object oriented programming with c++";
title[1]="programming in ANCI";
title[2]="electronic circuit theory";
title[3]="computer algorithm";
title[4]="complete solution of balagurusamy(c++)";

author[0]="balagurusamy";
author[1]="balagurusamy";
author[2]="boyelstade";
author[3]="shahani";
author[4]="abdus sattar";

stock_copy[0]=200;
stock_copy[1]=150;
stock_copy[2]=50;
stock_copy[3]=80;
stock_copy[4]=300;

price[0]=120.5;
price[1]=115.75;
price[2]=140;
price[3]=180.5;
price[4]=120;
```

```

}

void book::book_detail(int i)
{
cout&lt;&lt;" *********book detail **********\n";
cout&lt;&lt;setw(25)&lt;&lt;"Title"&lt;&lt;setw(30)&lt;&lt;"Author Name"
&lt;&lt;setw(18)&lt;&lt;"Stock copy\n";
cout&lt;&lt;setw(15)&lt;&lt;title[i]&lt;&lt;setw(16)&lt;&lt;author[i]&lt;&lt;setw(15)
&lt;&lt;stock_copy[i]&lt;&lt;"\n";

}

int book::search()
{
char name[80],t[80];
cout&lt;&lt;"Enter author name in small letter : ";
gets(name);
cout&lt;&lt;" title of book in small letter : ";
gets(t);

int count=-1;
int a,b;
for(int i=0;i&lt;size;i++)
{

a=strcmp(name,author[i]);
b=strcmp(t,title[i]);
if(a==0 &amp;&amp; b==0)

count=i;

}

return count;
}

void book::buy(int i)
{
if(i&lt;0)
{
cout&lt;&lt;" This book is not available \n";
unsuccessful++;
}

else
{
book_detail(i);
cout&lt;&lt;" How many copies of this book is required : ? "; int copy; cin&gt;&gt;copy;

if(copy&lt;=stock_copy[i])
{
stock_copy[i]=stock_copy[i]-copy;
float total_price;
total_price=price[i]*copy;
cout&lt;&lt;"Total price = "&lt;&lt;total_price&lt;&lt;" TK\n";
successful++;
}
else
{
cout&lt;&lt;" Sorry your required copies is not available \n";
unsuccessful++;
}
}
}

void book::edit_price()
{
cout&lt;&lt;" To edit price ";
int i;
i=search();
cout&lt;&lt;"Enter new price : "; float p; cin&gt;&gt;p;
price[i]=p;
}
void book::showdetail()
{
cout&lt;&lt;setw(22)&lt;&lt;"Title"&lt;&lt;setw(30)&lt;&lt;" stock copy "&lt;&lt;setw(20)
&lt;&lt;" Price per book "&lt;&lt;endl;
for(int i=0;i&lt;size;i++)
{
cout&lt;&lt;setw(35)&lt;&lt;title[i]&lt;&lt;setw(10)&lt;&lt;stock_copy[i]
&lt;&lt;setw(18)&lt;&lt;price[i]&lt;&lt;endl;
}
}
void book::showtransaction()
{
cout&lt;&lt;setw(22)&lt;&lt;"Successful transaction"&lt;&lt;setw(34)
&lt;&lt;"unsuccessful transaction "&lt;&lt;endl&lt;&lt;setw(10)
&lt;&lt;successful&lt;&lt;setw(32)&lt;&lt;unsuccessful&lt;&lt;endl;
}

int main()
{
book b1;
int result;

result=b1.search();
b1.buy(result);
b1.showdetail();
b1.showtransaction();
b1.edit_price();
cout&lt;&lt;"************details after edit price
```

```
*****************"&lt;&lt;endl;
b1.showdetail();

return 0;
}
```

output

```
Enter author name in small letter  :  abdus sattar

title of book in small letter  :  complete solution of balagurusamy(c++)

*********book detail **********

Title                                              Author Name              Stock copy

complete solution of balagurusamy(c++) abdus sattar              300

How many copies of this book is required : ? 100

Total price = 12000  TK

Title                                    stock copy                Price per book

object oriented programming with c++ 200              120.5

programming in ANCI              150                         115.75

electronic circuit theory          50                           140

computer algorithm                 80                           180.5

complete solution of balagurusamy(c++)  200          120

Successful transaction                        unsuccessful transaction

1                                                          0

To edit price Enter author name in small letter : shahani

title of book in small letter : computer algorithm

Enter new price : 200

************details after edit price*****************

Title                                     stock copy               Price per book
```

```
object oriented programming with c++   200                  120.5

programming in ANCI               150                            115.75

electronic circuit theory            50                             140

computer algorithm                   80                              200

complete solution of balagurusamy(c++)    200                120
```

Chapter 7

Review Questions

7.1: **What is operator overloading?**

Ans: The mechanism of giving special meaning to an operator is known as operator overloading.

7.2: **Why is it necessary to overload an operator?**

Ans: We can almost create a new language of our own by the creative use of the function and operator overloading techniques.

7.3: **What is an operator function? Describe the syntax of an operator function.**

Ans: To define an additional task to an operator, we must specify what it means in relation to the class to which the operator is applied. By which function this is done, is called operator function. Syntax of operator function:

```
return type class name : : operator OP (argument list)
  {
 function body // task defined
   }
```

7.4: **How many arguments are required in the definition of an overloaded unary operator?**

Ans: No arguments are required.

7.5: **A class alpha has a constructor as follows:**
alpha(int a, double b);
Can we use this constructor to convert types?

Ans: No. The constructors used for the type conversion take a single argument whose type is to be converted.

7.6: **What is a conversion function How is it created Explain its syntax.**

Ans: C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type. This is referred to conversion function.
Syntax:

```
Operator type name ( )
 {
    (Function Statements)
  }
```

7.7: **A friend function cannot be used to overload the assignment operator =. Explain why?**

Ans: A friend function is a non-member function of the class to which it has been defined as friend. Therefore it just uses the functionality (functions and data) of the class. So it does not consist the implementation for that class. That's why it cannot be used to overload the assignment operator.

7.8: **When is a friend function compulsory? Give an example.**

Ans: When we need to use two different types of operands for a binary operator, then we must use friend function.
Example:
A = B + 2;
or
A = B * 2;
is valid
But A = 2 + B
or
A = 2 * B will not work.
Because the left hand operand is responsible for invoking the member function. In this case friend function allows both approaches.

7.9: **We have two classes X and Y. If a is an object of X and b is an object of Y and we want to say a = b; What type of conversion routine should be used and where?**

Ans: We have to use one class to another class type conversion. The type-conversion function to be located in the source class or in the destination class.

7.10: **State whether the following statements are TRUE or FALSE.**
(a) Using the operator overloading concept, we can change the meaning of an operator.
(b) Operator overloading works when applied to class objects only.
(c) Friend functions cannot be used to overload operators.
(d) When using an overloaded binary operator, the left operand is implicitly passed to the member function.
(e) The overloaded operator must have at least one operand that is user-defined type.
(f)Operator functions never return a value.
(g) Through operator overloading, a class type data can be converted to a basic type data.
(h) A constructor can be used to convert a basic type to a class type data.

Ans:
(a) FALSE
(b) TRUE
(c) FALSE
(d) FALSE
(e) TRUE
(f) FALSE
(g) TRUE
(h) TRUE

Debugging Exercises

7.1: **Identify the error in the following program.**

```
#include <iostream.h>
class Space
{
    int mCount;
public:
    Space()
    {
        mCount = 0;
    }

     Space operator ++()
     {
         mCount++;
         return Space(mCount);
```

```
       }
};
void main()
{
     Space objSpace;
     objSpace++;
}
```

Solution: The argument of Space() function is void type, so when this function is called there are no argument can send to it. But ‘mCount’ argument is sending to Space() function through return space(mCount); Statement.
Here return space (mCount); replaced by return space();

7.2: **Identify the error in the following program.**

```
#include <iostream.h>
enum WeekDays
{
     mSunday'
     mMonday,
     mtuesday,
     mWednesday,
     mThursday,
     mFriday,
     mSaturday
};
bool op==(WeekDays& w1, WeekDays& w2)
{
     if(w1== mSunday && w2==mSunday)
          return 1;
     else if(w1==mSunday && w2==mSunday)
          return 1;
     else if(w1==mSunday && w2==mSunday)
          return 1;
     else if(w1==mSunday && w2==mSunday)
          return 1;
     else if(w1==mSunday && w2==mSunday)
          return 1;
     else if(w1==mSunday && w2==mSunday)
          return 1;
     else if(w1==mSunday && w2==mSunday)
          return 1;
    else
          return 0;
}
void main()
{
     WeekDays w1 = mSunday, w2 = mSunday;
     if(w1==w2)
     cout<<"Same day";
     else
     cout<<"Different day";
}
```

Solution: bool OP = = (WeekDays & w1, WeekDays & w2) replaced by bool operator = = (Weekdays & w1, WeekDays & w2). All other code will remain same.

7.3: **Identify the error in the following program.**

```
#include <iostream.h>
class Room
{
    float mWidth;
    float mLength;
public:
    Room()
    {
    }
    Room(float w, float h)
           :mWidth(w), mLength(h)
   {
   }
   operator float ()
   {
      return (float)mWidth * mLength;
   }

   float getWidth()
   {
   }
   float getLength()
   {
       return mLength;
   }
};

void main()
{
    Room objRoom1(2.5, 2.5)
    float fTotalArea;
    fTotalArea = objRoom1;
    cout<< fTotalArea;
}
```

Solution: The float getWidth() function return float type data, but there is no return statement in getWidth() function. So it should write as follows.

```
 float getWidth()
{
   return mWidth;
}
```

All other code will remain unchanged.

Programming Exercises

7.1: **Crate a class FLOAT that contains one float data member. Overload all the four arithmetic operators so that they operate on the objects of FLOAT.**

Solution:

```
#include<iostream.h>

class FLOAT
{
        float data;
         public:
         FLOAT(){};
         FLOAT(float d)
         { data=d;}
         FLOAT operator+(FLOAT f1);
         FLOAT operator-(FLOAT f2);
         FLOAT operator*(FLOAT f3);
         FLOAT operator/(FLOAT f4);
        void display();
};
FLOAT FLOAT::operator+(FLOAT f1)
{
         FLOAT temp;
         temp.data=data+f1.data;
        return (temp);
}
FLOAT FLOAT::operator-(FLOAT f2)
{
        FLOAT temp;
        temp.data=data-f2.data;
        return temp;
}
FLOAT FLOAT::operator*(FLOAT f3)
```

```
{
        FLOAT temp;
    temp.data=data*f3.data;
       return temp;
}
FLOAT FLOAT::operator/(FLOAT f4)
{
            FLOAT temp;
            temp.data=data/f4.data;
            return (temp);
}
void FLOAT:: display()
{
            cout<<data<<"\n";
}
int main()
{
                     FLOAT F1,F2,F3,F4,F5,F6;
                     F1=FLOAT(2.5);
                     F2=FLOAT(3.1);
                     F3=F1+F2;
                     F4=F1-F2;
                     F5=F1*F2;
                     F6=F1/F2;
                     cout<<" F1 = ";
                     F1.display();
                    cout<<" F2 = ";
                    F2.display();
                    cout<<" F1+F2 = ";
                    F3.display();
                   cout<<" F1-F2 = ";
                   F4.display();
                   cout<<" F1*F2 = ";
                   F5.display();
                   cout<<" F1/F2= ";
                   F6.display();
   return 0;
}
```

output

```
F1 = 2.5
F2 = 3.1
F1+F2 = 5.6
F1-F2 = -0.6
F1*F2 = 7.75
F1/F2= 0.806452
```

7.2: **Design a class Polar which describes a point in the plane using polar coordinates radius and angle. A point in polar coordinates is shown below figure 7.3**
Use the overload + operator to add two objects of Polar.
Note that we cannot add polar values of two points directly. This requires first the conversion of points into rectangular coordinates, then adding the respective rectangular coordinates and finally converting the result back into polar coordinates. You need to use the following trigonometric formula:
x = r *
cos(a);

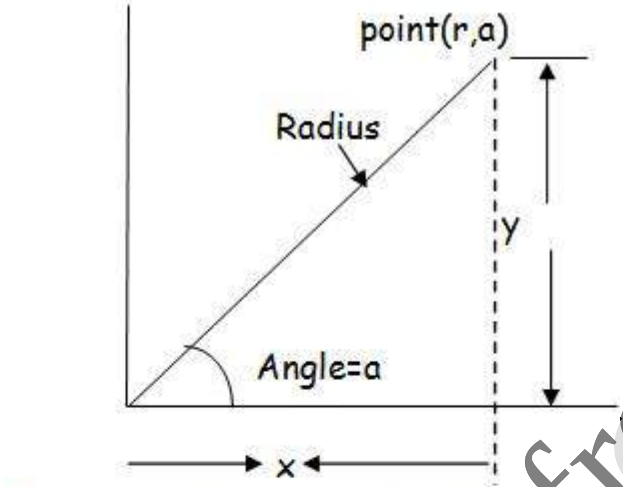


fig: polar coordinates of a point

y = r * sin(a);
a = atan(y/x); //arc tangent
r = sqrt(x*x + y*y);

Solution:

```
#include<iostream.h>
#include<math.h>
#define pi 3.1416
class polar
{
                float r,a,x,y;
                public:
                polar(){};
                polar(float r1,float a1);
                polar operator+(polar r1);
                void display(void);
};

polar :: polar(float r1,float a1)
{
           r=r1;
          a=a1*(pi/180);
```

```
            x=r*cos(a);
            y=r*sin(a);
}

polar polar :: operator+(polar r1)
{
            polar R;

            R.x=x+r1.x;
            R.y=y+r1.y;
            R.r=sqrt(R.x*R.x+R.y*R.y);
            R.a=atan(R.y/R.x);

  return R;
}

void polar::display()
{
          cout<<"radius = "<<r<<"\n angle = "<<a*(180/pi)<<"\n";
}

int main()
{
 polar p1,p2,p3;
 float r,a;
 cout<<" Enter radius and angle : ";
 cin>>r>>a;
 p1=polar(r,a);
 p2=polar(8,45);
 p3=p1+p2;
 cout<<" p1 : \n";
 p1.display();
 cout<<" p2 : \n ";
 p2.display();
 cout<<" p3 : \n ";
 p3.display();
 return 0;
}
```

output

```
Enter radius and angle : 10 45
P1:
radius = 10
angle = 44.999998
P2 :
radius = 8
angle = 44.999998
P3 :
radius = 18
angle = 44.999998
```

7.3: **Create a class MAT of size m * n. Define all possible matrix operations for MAT type objects.**

Solution:

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

class mat
{
             float **m;
             int rs,cs;
             public:
              mat(){}
              void creat(int r,int c);
              friend istream & operator >>(istream &,mat &);
              friend ostream & operator <<(ostream &,mat &);
              mat operator+(mat m2);
              mat operator-(mat m2);
              mat operator*(mat m2);
};

void mat::creat(int r,int c)
{
          rs=r;
          cs=c;
          m=new float *[r];
          for(int i=0;i<r;i++)
         m[i]=new float1;
}

istream &  operator>>(istream &din, mat &a)
{
        int r,c;
        r=a.rs;
        c=a.cs;
        for(int i=0;i<r;i++)
         {
                     for(int j=0;j<c;j++)
                        {
                             din>>a.m[i][j];
                        }
         }
    return (din);
}
ostream & operator<<(ostream &dout,mat &a)
{
              int r,c;
```

```
                r=a.rs;
                c=a.cs;
                        for(int i=0;i<r;i++)
                {
                     for(int j=0;j<c;j++)
                       {
                                           dout<<setw(5)<<a.m[i][j];
                                }
                                dout<<"\n";
                }
 return (dout);
}
mat mat::operator+(mat m2)
{
               mat mt;
               mt.creat(rs,cs);
              for(int i=0;i<rs;i++)
               {
                   for(int j=0;j<cs;j++)
                                {
                               mt.m[i][j]=m[i][j]+m2.m[i][j];
                      }
                }
     return mt;
}

mat mat::operator-(mat m2)
{
          mat mt;
          mt.creat(rs,cs);
           for(int i=0;i<rs;i++)
            {
                 for(int j=0;j<cs;j++)
                  {
                       mt.m[i][j]=m[i][j]-m2.m[i][j];
                  }
            }
    return mt;
}

mat mat::operator*(mat m2)
{
           mat mt;
                  mt.creat(rs,m2.cs);

    for(int i=0;i<rs;i++)
         {
              for(int j=0;j<m2.cs;j++)
               {
                     mt.m[i][j]=0;
                     for(int k=0;k<m2.rs;k++)
```

```
                    mt.m[i][j]+=m[i][k]*m2.m[k][j];
               }
          }

       return mt;
 }
int main()
{
         mat m1,m2,m3,m4,m5;
         int r1,c1,r2,c2;
         cout<<" Enter first matrix size : ";
         cin>>r1>>c1;
         m1.creat(r1,c1);
         cout<<"m1 = ";
         cin>>m1;
         cout<<" Enter second matrix size : ";
         cin>>r2>>c2;
         m2.creat(r2,c2);
         cout<<"m2 = ";
         cin>>m2;
         cout<<" m1:"<<endl;
         cout<<m1;
         cout<<" m2: "<<endl;
         cout<<m2;
         cout<<endl<<endl;
        if(r1==r2 && c1==c2)
          {
               m3.creat(r1,c1);
                     m3=m1+m2;
               cout<<" m1 + m2: "<<endl;
               cout<<m3<<endl;
              m4.creat(r1,c1);

               m4=m1-m2;
              cout<<" m1 - m2:"<<endl;
              cout<<m4<<endl<<endl;

        }
   else
    cout<<" Summation & substraction are not possible n"<<endl
        <<"Two matrices must be same size for summation &   substraction "<<endl<<endl;
if(c1==r2)
{
           m5=m1*m2;
           cout<<" m1 x m2: "<<endl;
           cout<<m5;
}
else
cout<<" Multiplication is not possible "<<endl
<<" column of first matrix must be equal to the row of second matrix ";
 return 0;
```

146}

output

```
Enter first matrix size : 2  2

m1 =

1     2

3     4

Enter second matrix size : 2    2

m2 =

5     6

7     8

m1 =

1     2

3     4

m2 =

5     6

7     8

m1+m2:

6       8

10     12

m1-m2:

-4       -4

-4      -4

m1 x m2:

19      22
```

```
43    50
```

7.4: **Define a class String. Use overload == operator to compare two strings.**

Solution:

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

class string
{
            char str[1000];
            public:
            void input(){gets(str);}
                  int operator==(string s2);
};
int string::operator==(string s2)
{
                int t= strcmp(str,s2.str);
        if(t==0)
                  t=1;
        else
        t=0;
      return t;
}

int main()
{

      char st1[1000],st2[1000];
      string s1,s2;
       cout<<" Enter 1st string : ";
       s1.input();
       cout<<" enter 2nd string : ";
       s2.input();

      if(s1==s2)
      cout<<" Two strings are equal ";
      else
      cout<<" Two string are not equal ";
   return 0;
}
```

output

```
Enter 1st string : our sweetest songs tel our saddest thought
enter 2nd string : a burning desire lead to success.
Two string are not equal
```

7.5: **Define two classes Polar and Rectangle to represent points in the polar and rectangle systems. Use conversion routines to convert from one system to the other.**

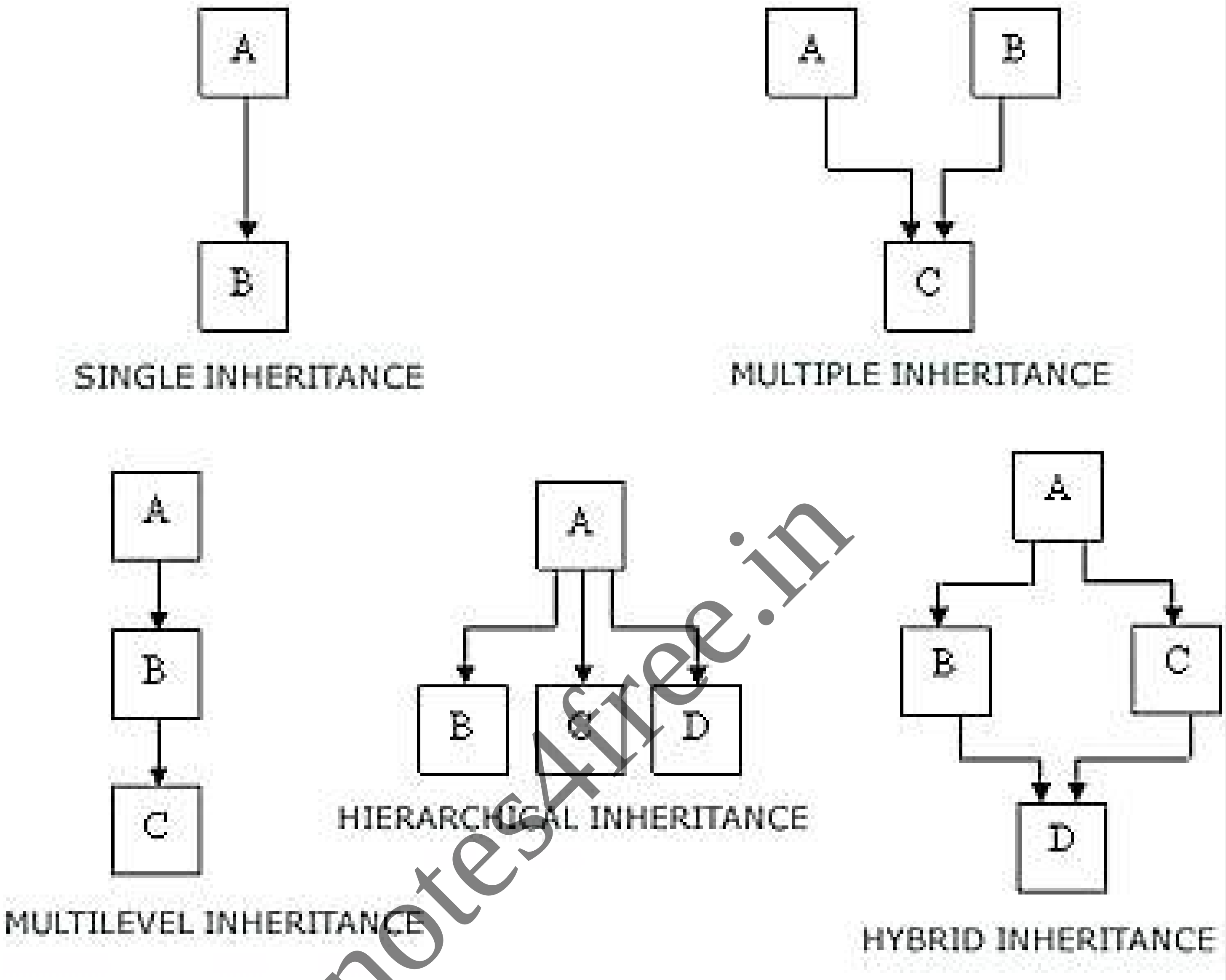
Solution:

```
#include<iostream.h>
#include<math.h>
#define pi 3.1416
class conversion_point
{
          float x,y,r,theta;
           public:
            void set_xy();
            void set_r_theta();
            void show_xy();
            void show_r_theta();
            void conversion(int t);
};
    void conversion_point::set_xy()
{
     cout<<"Enter the value of x & y : ";
     cin>>x>>y;
}
     void conversion_point::set_r_theta()
{
       cout<<"Enter the value of r & theta :";
       cin>>r>>theta;
       theta=(pi/180)*theta;
}

     void conversion_point::show_xy()
{
         cout<<" CERTECIAN FORM :\n"
                <<" x = "<<x<<"\n"
                <<" y = "<<y<<"\n";
}
void conversion_point::show_r_theta()
{
          cout<<" POLAR FORM :\n"
                <<" r = "<<r<<"\n"
                <<" theta = "<<(180/pi)*theta<<" degree \n";
}

void conversion_point::conversion(int t)
```

```
{
        if(t==1)
         {
                r=sqrt(x*x+y*y);

               if(x!=0)
                {
                     theta=atan(y/x);
                     show_r_theta();
                }

              else
                {
                      cout<<" POLAR FORM :\n"
                             <<" r = "<<r<<"\n"
                            <<" theta = 90 degree\n";
                }

          }
     else if(t==2)
       {
              x=r*cos(theta);
              y=r*sin(theta);
             show_xy();
       }
}

int main()
{
           conversion_point santo;
            int test;
           cout<<" press 1 to input certecian point \n"
                 <<" press 2 to input polar point \n "
                 <<" what is your input ? : ";
               cin>>test;
               if(test==1)
               santo.set_xy();
               else if(test==2)
               santo.set_r_theta();
               santo.conversion(test);

     return 0;
}
```

output

```
Press 1 to input certecian point
Press 2 to input polar point
what is your input ? 1
Enter the value of x & y : 4 5                r = 6.403124
POLAR FORM :                                  theta = 51.340073 degree
```



MODULE -4

Inheritance, Pointers, Virtual Functions, Polymorphism

GANESH Y
Dept. of ECE RNSIT

MODULE -4
Inheritance, Pointers, Virtual Functions, Polymorphism

SYLLABUS

Derived Classes, Single, multilevel, multiple inheritance, Pointers to objects and derived classes, this pointer, Virtual and pure virtual functions (Selected topics from Chap-8, 9 of Text).

Introduction

Reusability is yet another important feature of OOP. It is always nice if we could reuse something that already exists rather than trying to create the same all over again. It would not *only* save time and money but also reduce frustration and increase reliability. For instance, the reuse of a class that has already been tested, debugged and used many times can save us the effort of developing and testing the same again.

Fortunately, C++ strongly supports the concept of *reusability.* The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements. This is basically done by creating new classes, reusing the properties of the existing ones. The mechanism of deriving a new class from an old one is called *inheritance (or derivation).* The old class is referred to as the *base class* and the new one is called the *derived class or subclass.*

Definition: The capability of a class to derive properties and characteristics from another class is called **Inheritance**. The derived class inherits some or all of the traits from the base class.

Derived class or Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

Base class or Super Class: The class whose properties are inherited by sub class is called Base Class or Super class.

Different Forms of Inheritance

A class can also inherit properties from more than one class or from more than one level.

A derived class with only one base class, is called ***single inheritance*** and one with several base classes _is called ***multiple Inheritance.***

On the other hand, the traits of one class may be inherited by more than one class. This process is known as ***hierarchical inheritance.*** The mechanism of deriving a class from another 'derived class' is known as ***multilevel inheritance****.*

Figure below shows various forms of Inheritance that could be used for writing extensible programs. The direction of arrow Indicates the direction of inheritance.

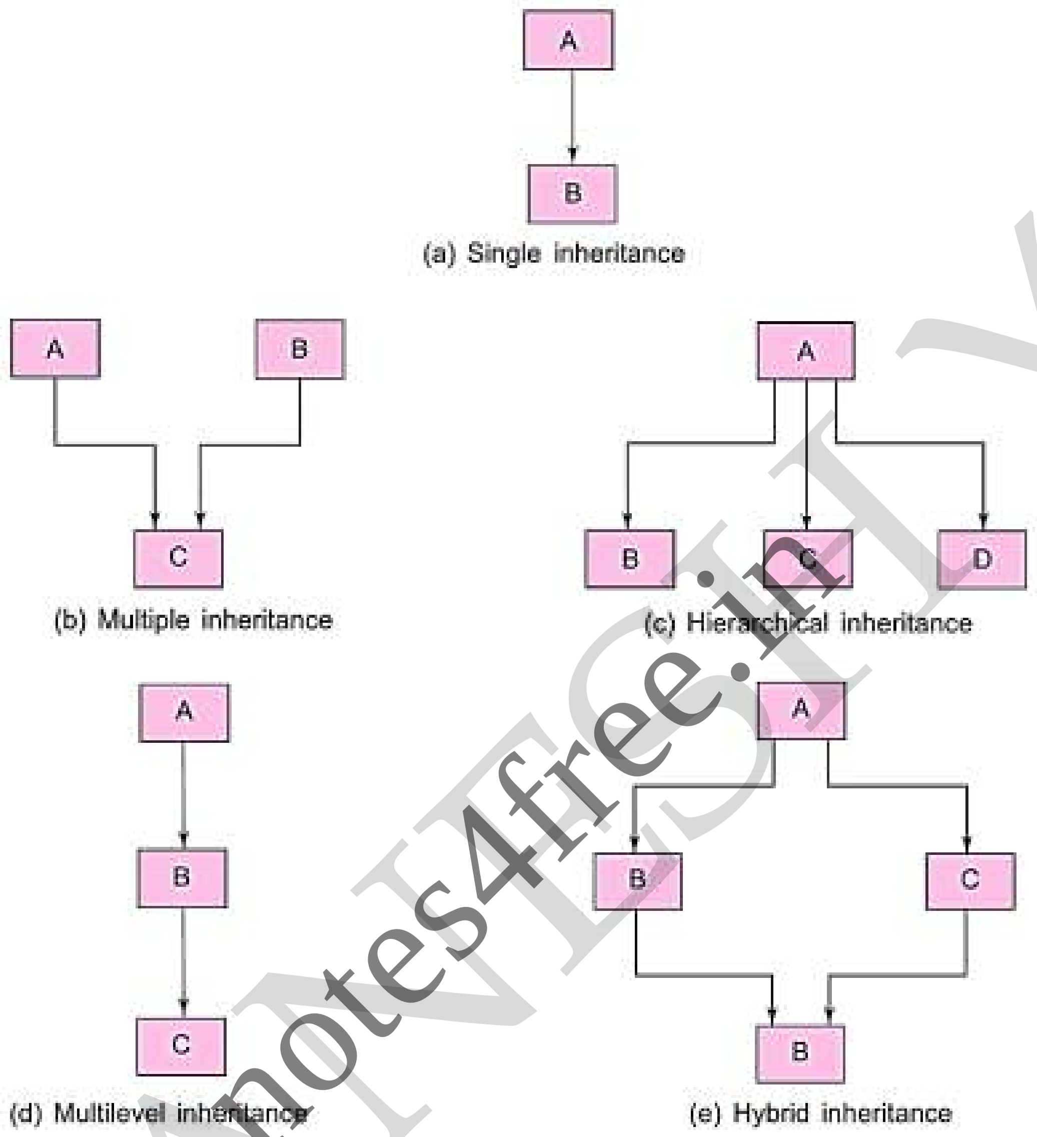


Fig. *Forms of inheritance*

Defining Derived Classes

For creating a sub-class which is inherited from the base class we have to follow the below syntax.

```
class derived_class_name: visiblity_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.*

```
class ABC:public XYZ // public derivation
{
        members of ABC;
};
class ABC:private XYZ // private derivation
{
        members of ABC;
};
class ABC:protected XYZ // protected derivation
{
        members of ABC;
};
class ABC: XYZ // private derivation by default
{
        members of ABC;
};

Syntax for multiple inheritance

class C: public A, public B
{
        ......;
};
```

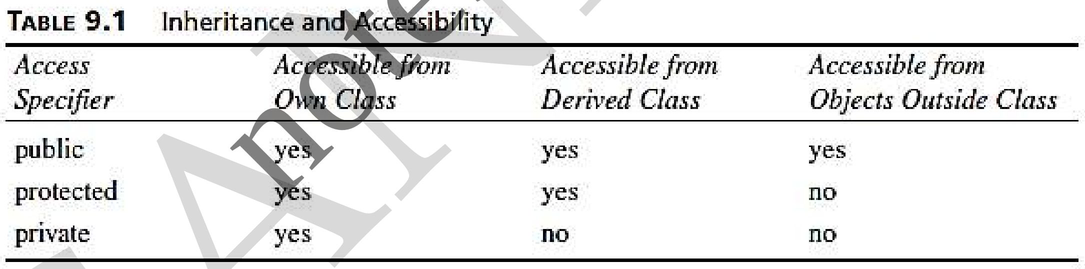


TABLE 9.1 Inheritance and Accessibility

Access Specifier	*Accessible from Own Class*	*Accessible from Derived Class*	*Accessible from Objects Outside Class*
public	yes	yes	yes
protected	yes	yes	no
private	yes	no	no

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. Remember, a public member of a class can be accessed by its own objects using the *dot operator.* The result is that no member of the base class is accessible to the objects of the derived class.

On the other hand, when the base class is *publicly inherited,* 'public members' of the base class become 'public members' of the derived class and therefore they are accessible to the objects of the derived class. In both the cases, the *private members* are not inherited and therefore, the private members of a base class will never become the members of its derived class.

In inheritance, some of the base class data elements and member functions are 'inherited' into the derived class. We can add our own data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

<u>Summary of above explanation</u>

Access in Base Class	Base Class Inherited as	Access in Derived Class
Public	Public	Public
Protected		Protected
Private		No access
Public	Protected	Protected
Protected		Protected
Private		No access
Public	Private	Private
Protected		Private
Private		No access

Single Inheritance

Let us consider a simple example to illustrate inheritance. Program 8.1 shows a base class **B** and a derived class **D.** The class **B** contains one private data member, one public data member, and three public member functions. The class **D** contains one private data member and two public member functions.

Program 8.1 **Single Inheritance: Public**

```
#include <iostream>

using namespace std;

class B
{
        int a;    // private; not inheritable
    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.set_ab();
        d.mul();
        d.show_a();
        d.display();

        d.b = 20;
        d.mul();
        d.display();

        return 0;
}
```

The output of Program 8.1 would be:

```
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.

The class D, in effect, will have more members than what it contains at the time of declaration as shown in Fig. below.

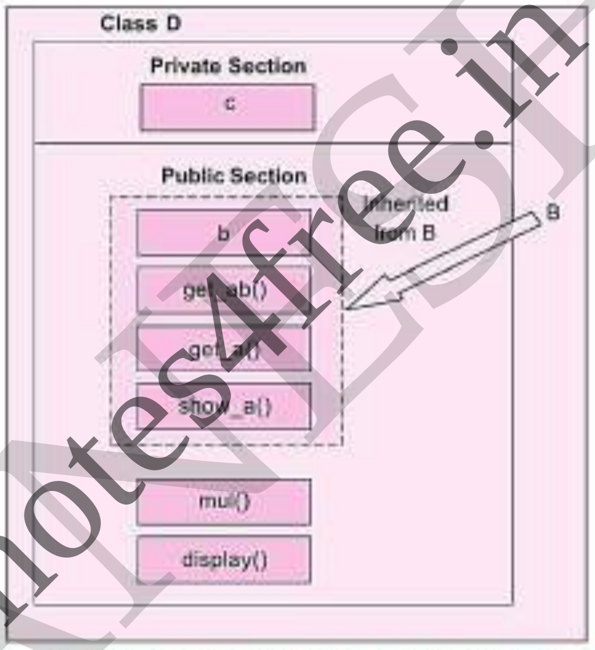


Fig. Adding more members to a class (by public derivation)

The program illustrates that the objects of class **D** have access to all the public members of **B.** Let us have a look at the functions **show_a()** and **mul():**

```
void show_a()
{
      cout << "a = " <<a<< "\n";
}
void mul ()
{
      c = b * get_a(); // c = b * a
}
```

Although the data member **a** is private in **B** and cannot be inherited, objects of **D** are able to access it through an inherited member function of **B.**

Let us now consider the case of private derivation.

```
class B
{
      int a;
   public:
      int b;
      void get_ab();
void get_a();
      void show_a();
};

class D : private B                  // private derivation
{
      int c;
   public:
      void mul();
      void display();
};
```

The membership of the derived class D is shown in Fig. below. 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.

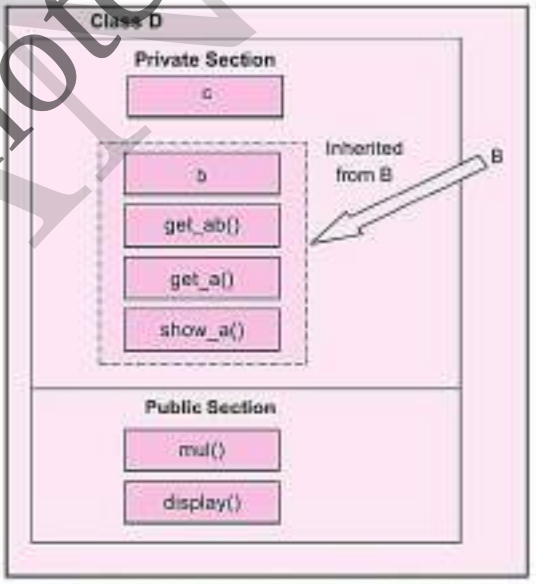


Fig. *Adding more members to a class (by private derivation)*

The statements such as `d.get_ab(); d.get_a(); d.show_a();` will not work. However, these functions can be used inside `mul()` and `display()` like the normal functions as shown below:

```
void mul()
{
    get_ab();
    c = b * get_a();
}
void display()
{
    show_a();                    // outputs value of 'a'
    cout << "b = " << b << "\n"
       << "c = "   << c << "\n\n";
}
```

Program 8.2 incorporates these modifications for private derivation. Please compare this with Program 8. 1.

Program 8.2 **Single Inheritance : Private**

```
#include <iostream>

using namespace std;

class B
{
    int a;                       // private; not inheritable
  public:
    int b;                       // public; ready for inheritance
    void get_ab();
    int get_a(void);
    void show_a(void);
};

class D : private B              // private derivation
{
    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"
       << "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; b has become private
        d.mul();
        d.display();

        return 0;
}
```

Suppose a base class and a derived class define a function of the same name. What will happen when a derived class object invokes the function? In such cases, the derived class function supersedes the base class definition. However, if the derived class does not redefine the function, then the base class function will be called.

Making a Private Member Inheritable

We have seen that a private member of a base class cannot be inherited and therefore it is not available for the derived class directly. What do we do if the **private** data needs to be inherited by a derived class? This can be accomplished by modifying the visibility limit of the **private** member by making it **public.** This would make it accessible to all the other functions of the program, thus taking away the advantage of data hiding.

C++ provides a third *visibility modifier,* **protected,** which serve a limited purpose in inheritance. **A** member declared as **protected** is accessible by the member functions within its class and any class immediately derived from it. It *cannot* be accessed by the functions outside these two classes. A class can now use all the three visibility modes as illustrated below:

```
class alpha
{
  private:                    // optional
    .....                     // visible to member functions
    .....                     // within its class
  protected:
    .....                     // visible to member functions
    .....                     // of its own and that of derived class
  public:
    .....                     // visible to all functions
    .....                     // in the program
};
```

Figure below shows the pictorial representation for the two levels of derivation.

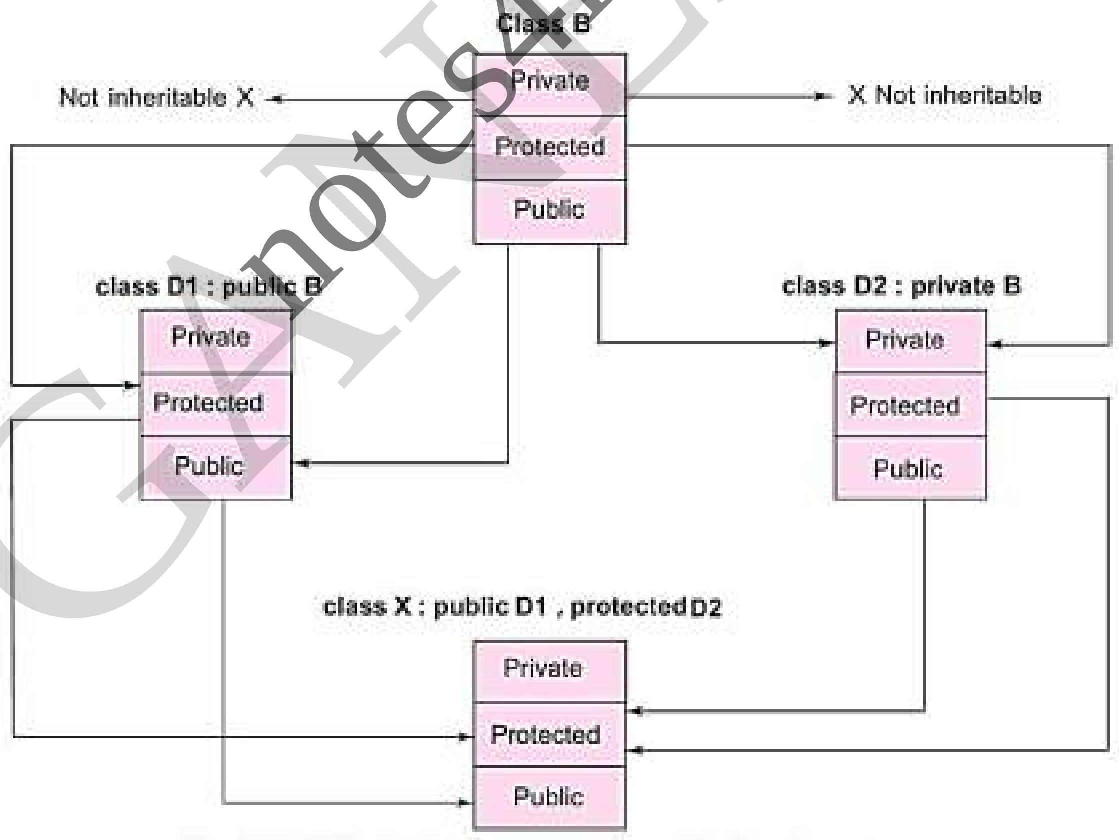


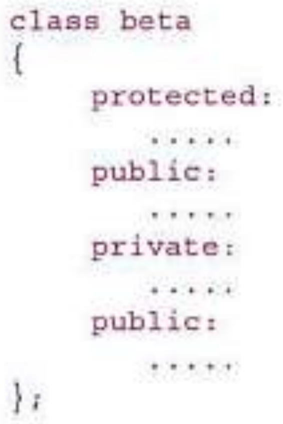
Fig. *Effect of inheritance on the visibility of members*

When a **protected** member is inherited in **public** mode, it becomes **protected** in the derived class too and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance.

A **protected** member, inherited in the **private** mode derivation, becomes **private** in the derived class. Although **it** is available to the member functions of the derived class, it is not available for further inheritance (since **private** members cannot be inherited).

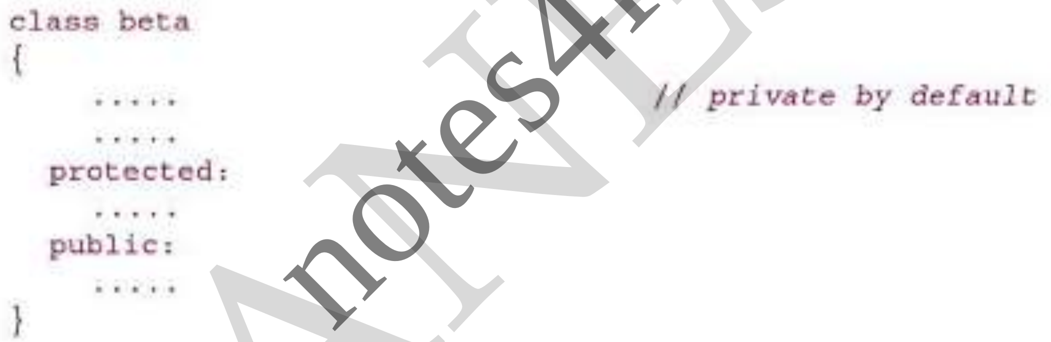
(visibility mode summery table explained once again ⇑)

The keywords **private, protected,** and **public** may appear in any order and any number of times in the declaration of a class. For example,



```
class beta
{
    protected:
        .....
    public:
        .....
    private:
        .....
    public:
        .....
};
```

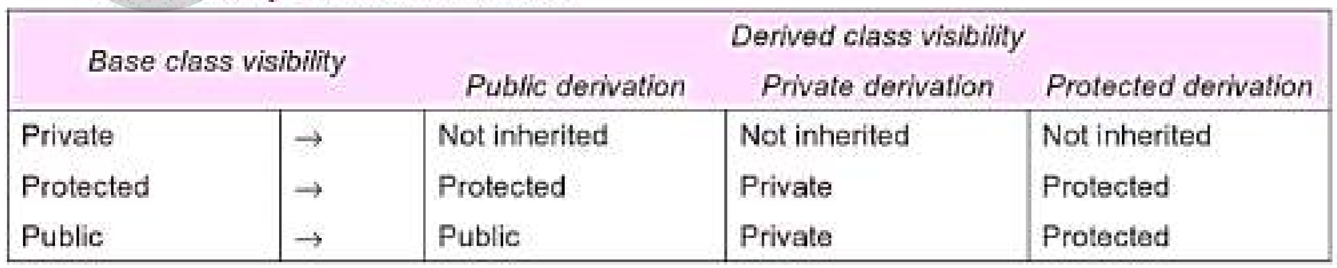
is a valid class definition. However, the normal practice is to use them as follows:



```
class beta
{
    .....                        // private by default
    .....
  protected:
    .....
  public:
    .....
}
```

It is also possible to inherit a base class in **protected** mode (known as *protected derivation).* In protected derivation, both the **public** and **protected** members of the base class become **protected** members of the derived class. Table below (same as visibility summary table shown earlier) summarizes how the visibility of base class members undergoes modifications in all the three types of derivation.

Table *Visibility of inherited members*



Base class visibility		*Derived class visibility*		
		Public derivation	*Private derivation*	*Protected derivation*
Private	→	Not inherited	Not inherited	Not inherited
Protected	→	Protected	Private	Protected
Public	→	Public	Private	Protected

Now let us review the access control to the **private** and **protected** members of a class. What are the various functions that can have access to these members? They could be:

1. A function that is a friend of the class.

2. A member function of a class that is a friend of the class.

3. A member function of a derived class.

While the friend functions and the member functions of a friend class can have direct access to both the **private** and **protected** data, the member functions of a derived class can directly access only the **protected** data. However. they can access the **private** data through the member functions of the base class.

Figure 8.5 illustrates how the access control mechanism works in various situations. A simplified view of access control to the members of a class is shown in Fig. 8.6.

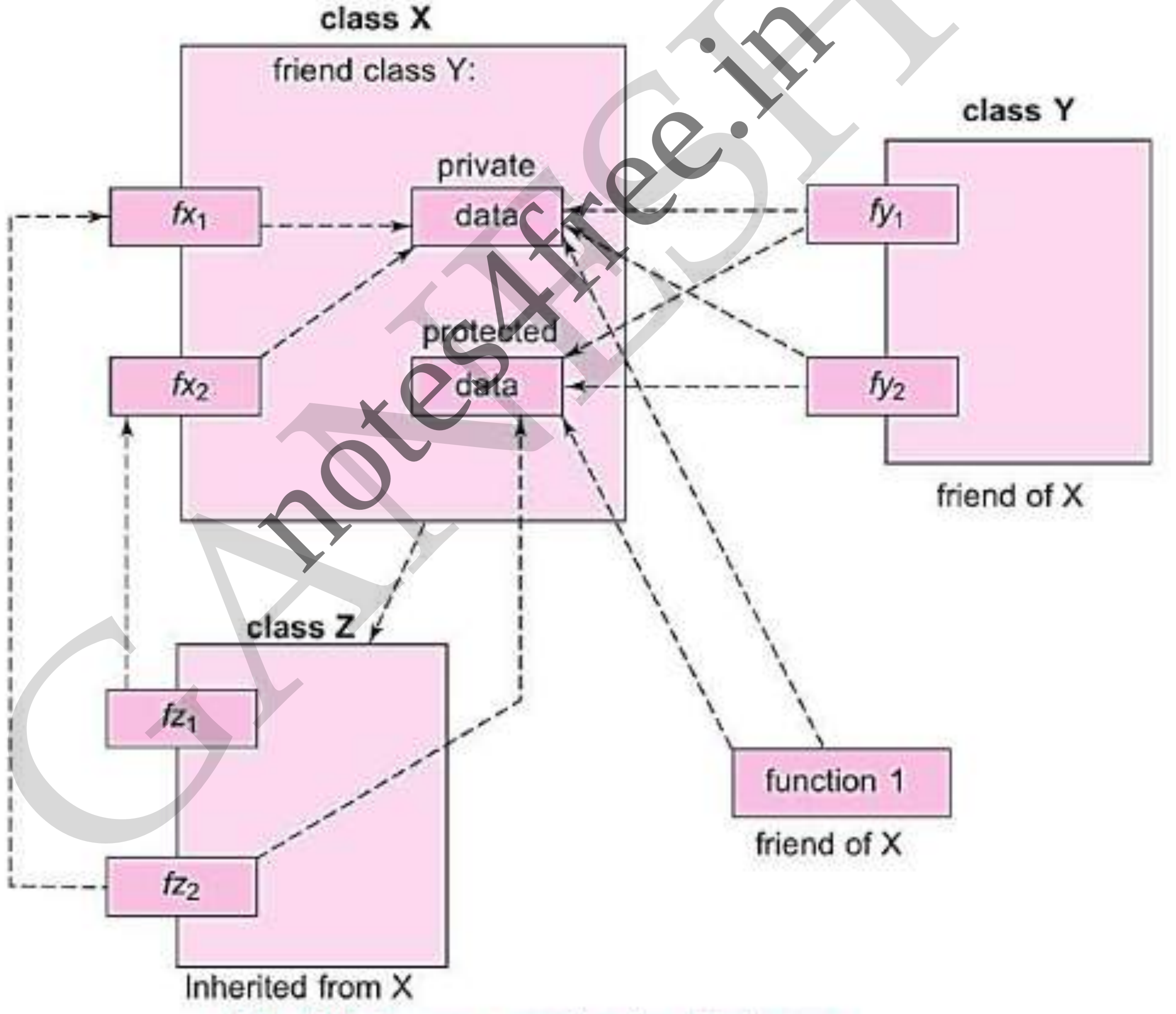


Fig. 8.5 *Access mechanism in classes*

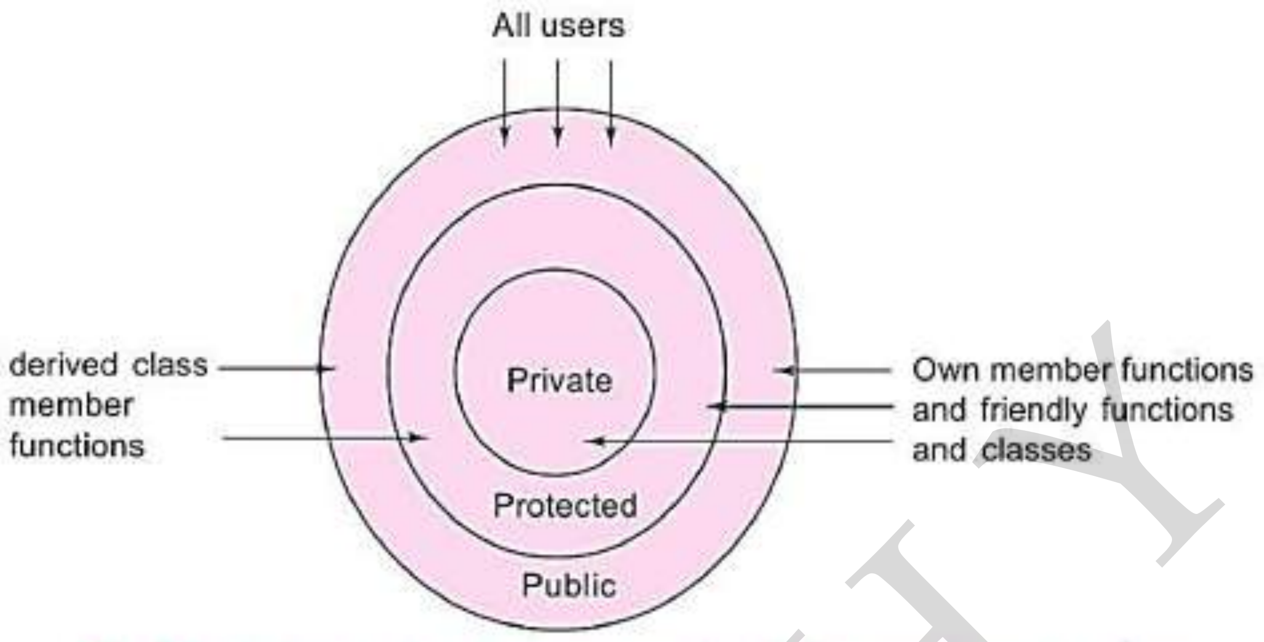

All users
derived class member functions
Private
Protected
Public
Own member functions and friendly functions and classes

Fig. 8.6 A simple view of access control to the members of a class

Multilevel Inheritance

It is not uncommon that a class is derived from another derived class as shown in Fig. 8. 7.

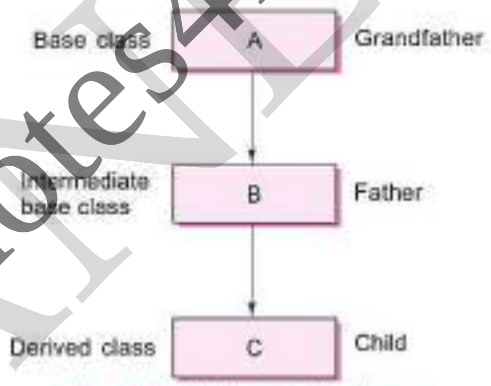


Fig. 8.7 Multilevel inheritance

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. The chain **ABC** is known as inheritance *path.* This process can be extended to any number of levels.

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
```

Let us consider a simple example. Assume that the test results of a batch of students are stored in three different classes.
Class **student** stores the roll number, class **test** stores the marks obtained in two subjects and class **result** contains the total marks obtained in the test.
The class result can inherit the details of the marks obtained in the test and the roll number of students through multilevel inheritance. Example:

```
 class student
 {
      protected:
      int roll_number;
    public:
      void get_number(int);
      void put_number(void);
 };

void student :: get_number(int a)
{
     roll_number = a;
}
void student :: put_number()
{
     cout << "Roll Number: " << roll_number << "\n";
}

class test : public student        // First level derivation
{
   protected:
     float sub1;
     float sub2;
   public:
     void get_marks(float, float);
     void put_marks(void);
};
void test :: get_marks(float x, float y)
{
     sub1 = x;
     sub2 = y;
}
void test :: put_marks()
{
     cout << "Marks in SUB1 = " << sub1 << "\n";
     cout << "Marks in SUB2 = " << sub2 << "\n";
}
class result : public test                    // Second level derivation
{
     float total;                             // private by default
   public:
     void display(void);
};
```

The class **result**, after inheritance from 'grandfather' through 'father', would contain the following members:

```
private:
    float total;                        // own member
protected:
    int roll_number;                    // inherited from student
                                        // via test
    float sub1;                         // inherited from test
    float sub2;                         // inherited from test
public:
    void get_number(int);               // from student via test
    void put_number(void);              // from student via test
    void get_marks(float, float);       // from test
    void put_marks(void);               // from test
    void display(void);                 // own member
```

The inherited functions `put_number()` and `put_marks()` can be used in the definition of `display()` function:

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

Here is a simple **main()** program:

```
int main()
{
        result student1;    // student1 created
        student1.get_number(111);
        student1.get_marks(75.0, 59.5);
        student1.display();

        return 0;
}
```

This will display the result of student1. The complete program is shown in Program8.3.

Program 8.3 Multilevel Inheritance

```
#include <iostream>

using namespace std;

class student
{
     protected:
          int roll_number;
     public:
          void get_number(int);
          void put_number(void);
};

void student :: get_number(int a)
{
          roll_number = a;
}
void student :: put_number()
{
          cout << "Roll Number: " << roll_number << "\n";
}
class test : public student         // First level derivation
{
     protected:
          float sub1;


        float sub2;
   public:
           void get_marks(float, float);
           void put_marks(void);
};

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

void test :: put_marks()
{
         cout << "Marks in SUB1 = " << sub1 << "\n";
         cout << "Marks in SUB2 = " << sub2 << "\n";
}
```

```
class result : public test         // Second level derivation
{
      float total;                 // private by default
    public:
      void display(void);
};

void result :: display(void)
{
        total = sub1 + sub2;
        put_number();
        put_marks();
        cout << "Total = " << total << "\n";
}
int main()
{
        result student1;           // student1 created

        student1.get_number(111);

        student1.get_marks(75.0, 59.5);

        student1.display();

        return 0;
}
```

The output of Program 8.3 would be:

```
Roll Number: 111

Marks in SUB1= 75

Marks in SUB2 = 59.5

Total =134.5
```

Multiple Inheritance

A class can inherit the attributes of two or more classes as shown in Fig. 8.8. This is known as ***multiple inheritance***. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent and the intelligence of another.

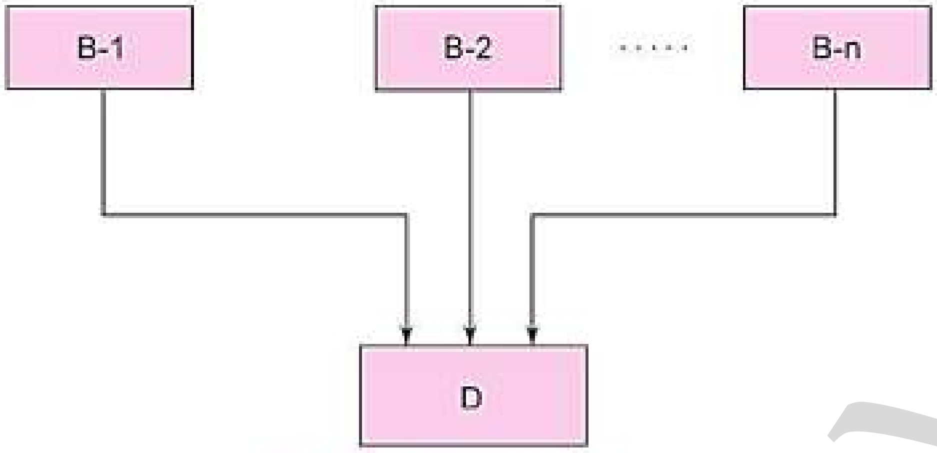


Fig. 8.8 *Multiple inheritance*

The syntax of a derived class with multiple base classes is as follows:

```
class D: visibility B-1, visibility B-2 ...
{
    .....
    ..... (Body of D)
    .....
};
```

where, *visibility* may be either **public** or **private.** The base classes are separated by commas. Example:

```
class P : public M, public N
{
     public:
         void display(void);
};
```

Classes **M** and **N** have been specified as follows:

```
class M
{
     protected:
         int m;
     public:
       void get_m(int);
};
void M :: get_m(int x)
{
        m = x;
}
class N
{
     protected:
          int n;
     public:
          void get_n(int);
};
void N :: get_n(int y)
{
          n = y;
}
```

The derived class **P**, as declared above, would, in effect, contain all the members of **M** and **N** in addition to its own members as shown below:

```
class P
{
     protected:
          int m;                  // from M
          int n;                  // from N

     public:
          void get_m(int);        // from M
          void get_n(int);        // from N
          void display(void);     // own member
};
```

The member function display() can be defined as follows:

```
void P :: display(void)
{
          cout << "m = " << m << "\n";
          cout << "n = " << n << "\n";
          cout << "m*n =" << m*n << "\n";
};
```

The main() function which provides the user-interface may be written as follows:

```
main()
{
          P p;
          p.get_m(10);
          p.get_n(20);
          p.display();
}
```

Program 8.4 shows the entire code illustrating how all the three classes are implemented in multiple inheritance mode.

Program 8.4 Multiple Inheritance

```
#include <iostream>

using namespace std;

class M
{
    protected:
        int m;
    public:
        void get_m(int);
};

class N
{
    protected:
        int n;
    public:
        void get_n(int);
};

class P : public M, public N
{
    public:
        void display(void);
};

void M :: get_m(int x)
{
        m = x;
}

void N :: get_n(int y)
{
        n = y;
}

void P :: display(void)
{
        cout << "m = " << m << "\n";
        cout << "n = " << n << "\n";
        cout << "m*n = " << m*n << "\n";
}
int main()
{
        P p;
        p.get_m(10);
        p.get_n(20);
        p.display();

        return 0;
}
```

The output of Program 8.4 would be:

```
m = 10
n = 20
m*n = 200
```

Ambiguity Resolution in Inheritance

Occasionally, we may face a problem in using the multiple inheritance, when a function with the same name appears in more than one base class. Consider the following two classes.

```
class M
{
    public:
        void display(void)
        {
            cout << "Class M\n";
        }
};

class N
{
    public:
        void display(void)
        {
            cout << "Class N\n";
        }
};
```

Which **display ( )** function is used by the derived class when we inherit these two classes? We can solve this problem by defining a named instance within the derived class, using the class resolution operator with the function as shown below:

```
class P : public M, public N
{
    public:
        void display(void)      // overrides display() of M and N
        {
            M :: display(); }
        };
```

We can now use the derived class as follows:

```
int main()
{
        P p;
        p.display();
}
```

Ambiguity may also arise in single inheritance applications. For instance, consider the following situation:

```
class A
{
  public:
        void display()
        {
                cout<<"A\n";
        }
};
class B:public A
{
  public:
        void display()
        {
                cout<<"B\n";
        }
};
```

In this case, the function in the derived class overrides the inherited function and, therefore, a simple call to **display ( ) by B** type object will invoke function defined in **B** only. However, we may invoke the function defined in **A** by using the scope resolution operator to specify the class.

```
int main()
{
        B b; // derived class object
        b.display(); // invokes display() in B
        b.A::display();// invokes display() in A
        b.B::display();// invokes display() in B
        return 0;
}
```

```
This will produce the following output:
B
A
B
```

Pointers to Objects

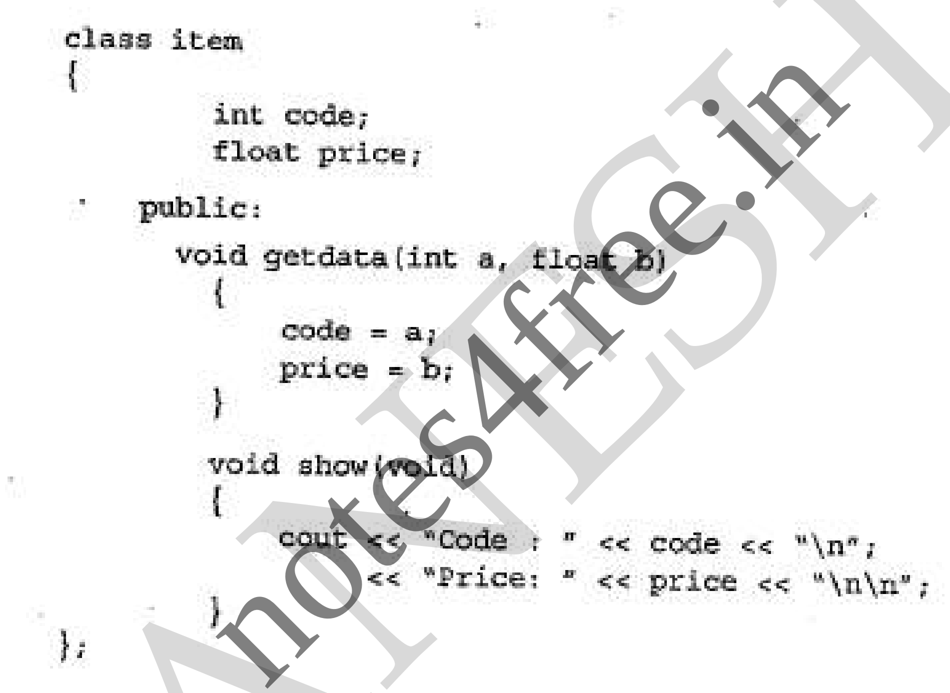
We have already seen how to use pointers to access the class members. As stated earlier, a pointer can point to an object created by a class. Consider the following statement:

```
item x;
```

where **item** is a class and x is an object defined to be of type item. Similarly, we can define a pointer **it_ptr** of type **item** as follows:

```
item *it_ptr;
```

Object pointers are useful in creating objects at run time. We can also use an object pointer to access the public members of an object. Consider a class **item** defined as follows:



```
class item
{
        int code;
        float price;

    public:
      void getdata(int a, float b)
        {
            code = a;
            price = b;
        }

        void show(void)
        {
            cout << "Code : " << code << "\n";
                 << "Price: " << price << "\n\n";
        }
};
```

Let us declare an **item** variable **x** and a pointer **ptr** to **x** as follows:

```
item x;
item *ptr = &x;
```

The pointer **ptr** is initialized with the address of **x.**

We can refer to the member functions of **item** in two ways, one by using the *dot operator* and *the object,* and another by using the *arrow operator* and the *object pointer.* The statements

```
x.getdata(100,75.50);
x.show();
```

are equivalent to

```
ptr->getdata(100, 75.50);
ptr->show();
```

Since ***ptr** is an alias of **x,** we can also use the following method:

```
(*ptr).show();
```

The parentheses are necessary because the dot operator has higher precedence than the *indirection operator**.

We can also create the objects using pointers and **new** operator as follows:

```
item *ptr = new item;
```

This statement allocates enough memory for the data members in the object structure and assigns the address of the memory space to **ptr.** Then **ptr** can be used to refer to the members as shown below:

```
ptr -> show();
```

If a class has a constructor with arguments and does not include an empty constructor, then we must supply the arguments when the object is created.

We can also create an array of objects using pointers. For example, the statement

```
item *ptr = new item[10];   // array of 10 objects
```

creates memory space for an array of 10 objects of **item.** Remember, in such cases, if the class contains constructors, it must also contain an empty constructor.

Program 9.8 **Pointers to Objects**

```
#include <iostream>
using namespace std;
class item
{
        int code;
        float price;
    public:
        void getdata(int a, float b)
        {
            code = a;
            price = b;
        }

        void show(void)
        {
            cout << "Code : " << code << "\n";
            cout << "Price: " << price << "\n";
        }
};

const int size = 2;
```

```
int main()
{
        item *p = new item [size];
        item *d = p;
        int x, i;
        float y;

        for(i=0; i<size; i++)
        {
            cout << "Input code and price for item" << i+1;
            cin >> x >> y;
            p->getdata(x,y);
            p++;
        }

        for(i=0; i<size; i++)
        {
            cout << "Item:" << i+1 << "\n";
            d->show();
            d++;
        }
        return 0;
}
```

The output of Program 9.8 would be:

```
Input code and price for item1 40 500
Input code and price for item2 50 600
Item:1
Code : 40
Price: 500
Item:2
Code : 50
Price: 600
```

In Program 9.8 we created space dynamically for two objects of equal size. But this may not be the case always. For example, the objects of a class that contain character strings would not be of the same size. In such cases, we can define an array of pointers to objects that can be used to access the individual objects. This is illustrated in Program 9.9.

Program 9.9 Array of Pointers to Objects

```
#include <iostream>
#include <cstring>

using namespace std;

class city
{
        protected:
            char *name;
            int len;
      public:
            city()

            {
                len = 0;
                name = new char[len+1];
            }
              void getname(void)
              {
                 char *s;
                 s = new char[30];

                 cout << "Enter city name:";
                 cin >> s;
                 len = strlen(s);
                 name = new char[len + 1];
                 strcpy(name, s);
              }
              void printname(void)
              {
                 cout << name << "\n";
              }
};

int main()
{
           city *cptr[10]; // array of 10 pointers to cities

           int n = 1;
           int option;

           do
           {
                 cptr[n] = new city; // create new city
                 cptr[n]->getname();
                 n++;
                 cout << "Do you want to enter one more name?\n";
                 cout << "(Enter 1 for yes 0 for no):";
            cin >> option;
        }
        while(option);

        cout << "\n\n";
        for(int i=1; i<=n; i++)
        {
            cptr[i]->printname();
        }

        return 0;
}
```

The output of Program 9.9 would be:

```
Enter city name:Hyderabad
Do you want to enter one more name?
(Enter 1 for yes 0 for no);1
Enter city name:Secunderabad
Do you want to enter one more name?
(Enter 1 for yes 0 for no);1
Enter city name:Malkajgiri
Do you want to enter one more name?
(Enter 1 for yes 0 for no);0

Hyderabad
Secunderabad
Malkajgiri
```

this POINTER

C++ uses a unique keyword called **this** to represent an object that invokes a member function. **This** is a pointer that points to the object for which *this* function was called.

For example, the function call **A.max()** will set the pointer **this** to the address of the object ***A***. The starting address is the same as the address of the first variable in the class structure.

This unique pointer is automatically passed to a member function when it is called. The pointer **this** acts as an *implicit* argument to all the member functions. Consider the following simple example:

```
class ABC
{
        int a;
        .....
        .....
};
```

The private variable 'a' can be used directly inside a member function, like

```
a= 123;
```

We can also use the following statement to do the same job:

```
this->a =123;
```

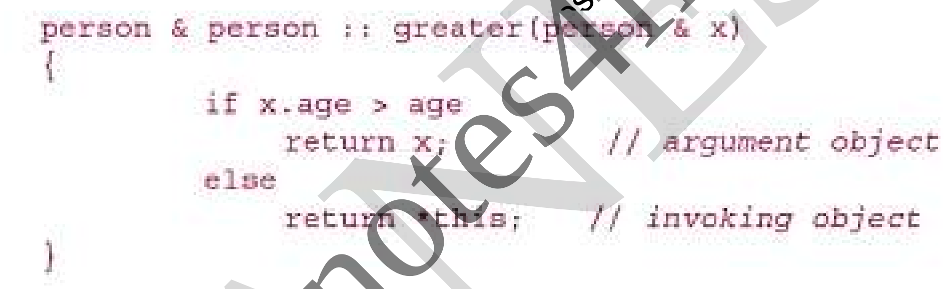
Since C++ permits the use of shorthand form **a** = **123,** we have not been using the pointer **this** explicitly so far. However, we have been implicitly using the pointer **this** when overloading the operators using member function.

Recall that, when a binary operator is overloaded using a member function, we pass only one argument to the function. The other argument is implicitly passed using the pointer **this.**

One important application of the pointer **this** is to return the object it points to. For example, the statement

```
return *this;
```

inside a member function will return the object that invoked the function. This statement assumes importance when we want to compare two or more objects inside a member function and return the invoking object as a result. Example:



```
person & person :: greater(person & x)
{
        if x.age > age
            return x;        // argument object
        else
            return *this;    // invoking object
}
```

Suppose we invoke this function by the call

```
max= A.greater(B);
```

The function will return the object **B** (argument object) if the age of the person **B** is greater than that of **A,** otherwise, it will return the object **A** (invoking object) using the pointer **this.**

Remember, the dereference operator * produces the contents at the address contained in the pointer. A complete program to illustrate the use of **this** is given in Program 9.10.

Program 9.10 this Pointer

```
#include <iostream>
#include <cstring>

using namespace std;

class person
{
          char name[20];
          float age;
     public:
          person(char *s, float a)
          {
               strcpy(name, s);
               age = a;
          }
          person & person :: greater(person & x)
          {
               if(x.age >= age)
                   return x;
               else
                   return *this;
          }

          void display(void)
          {
               cout << "Name: " << name << "\n"
                    << "Age: "  << age  << "\n";
          }
};

int main()
{
          person P1("John", 37.50),
                 P2("Ahmed", 29.0),
                 P3("Hebber", 40.25);

          person P = P1.greater(P3);       // P3.greater(P1)
          cout << "Elder person is: \n";
          P.display();

          P = P1.greater(P2);             // P2.greater(P1)
          cout << "Elder person is: \n";
          P.display();

          return 0;
}
```

The output of Program 9.10 would be:

```
Elder person is:
Name: Hebber
Age: 40.25
Elder person is:
Name: John
Age: 37.5
```

Pointers to Derived Classes

We can use pointers not only to the base objects but also to the objects of derived classes. Pointers to objects of a base class are type-compatible with pointers to objects of a derived class.

Therefore, a single pointer variable can be made to point to objects belonging to different classes. For example,

if **B** is a base class and **D** is a derived class from **B,** then a pointer declared as a pointer to **B** can also be a pointer to **D.** Consider the following declarations:

```
B *cptr;     // pointer to class B type variable
B b;         // base object
D d;         // derived object
cptr = &b;  // cptr points to object b
```

We can make **cptr** to point to the object **d** as follows:

```
cptr = &d;  // cptr points to object d
```

This is perfectly valid with C++ because **d** is an object derived from the class **B.**

However, there is a problem in using **cptr** to access the public members of the derived class **D.** Using **cptr,** we can access only those members which are inherited from **B** and not the members that originally belong to **D.**

In case a member of **D** has the same name as one of the members of **B,** then any reference to that member by **cptr** will always access the base class member.

Although C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. We may have to use another pointer *declared* as pointer to the derived type.

Program 9.11 illustrates how pointers to a derived object are used.

Program 9.11 Pointers to Derived Objects

```
#include <iostream>

using namespace std;

class BC
{
        public:
            int b;
            void show()
            { cout << "b = " << b << "\n";}
};
class DC : public BC
{
        public:
            int d;
            void show()
            { cout << "b = " << b << "\n"
                   << "d = " << d << "\n";
            }
};

int main()
{
        BC *bptr;              // base pointer
        BC base;
        bptr = &base;          // base address

        bptr->b = 100;         // access BC via base pointer
        cout << "bptr points to base object \n";

        bptr -> show();
        // derived class
        DC derived;
        bptr = &derived;       // address of derived object
        bptr -> b = 200;       // access DC via base pointer

        /* bptr -> d = 300;*/  // won't work
        cout << "bptr now points to derived object \n";
        bptr -> show();        // bptr now points to derived object

        /* accessing d using a pointer of type derived class DC */

        DC *dptr;              // derived type pointer
        dptr = &derived;
        dptr->d = 300;
```

```
            cout << "dptr is derived type pointer\n";
            dptr -> show();

            cout << "using ((DC *)bptr)\n";
            ((DC *)bptr) -> d = 400;
            ((DC *)bptr) -> show();

            return 0;
    }
```

The output of Program 9.11 would be:

```
    bptr points base object
    b = 100
    bptr now points to derived object
    b = 200
    dptr is derived type pointer
    b = 200
    d = 300
    using ((DC *)bptr)
    b = 200
    d = 400
```

Note *We have used the statement*

```
        bptr -> show();
```

two times. First, when ***bptr*** *points to the base object, and second when* ***bptr*** *is made to point to the derived object. But, both the times, it executed* ***BC::show()*** *function and displayed the content of the base object. However, the statements*

```
        dptr -> show();
        ((DC *) bptr) -> show();    // cast bptr to DC type
```

display the contents of the ***derived*** *object. This shows that, although a base pointer can be made to point to any number of derived objects, it cannot directly access the members defined by a derived class.*

Polymorphism

Polymorphism is one of the crucial features of OOP. It simply means 'one name, multiple forms'. We have already seen how the concept of *polymorphism* is implemented using the overloaded functions and operators.

The overloaded member functions are 'selected' for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called *early binding* or *static binding* or *static linking.* Also known as ***compile time polymorphism***, early binding simply means that an object is bound to its function call at compile time.

Now let us consider a situation where the function name and prototype is the same in both the base and derived classes. For example, consider the following class definitions:

```
class A
{
        int x;
     public:
        void show() {....}    // show() in base class
};
class B: public A
{
          int y;
     public:
          void show() {....}    // show() in derived class
};
```

How do we use the member function **show()** to print the values of objects of both the classes **A** and **B?** Since the prototype of **show()** is the same in both the places, the function is not overloaded and therefore static binding does not apply.

We have seen earlier that, in such situations. we may use the class resolution operator to specify the class while invoking the functions with the derived class objects.

It would be nice if the appropriate member function could be selected while the program is running. This is known as ***run time polymorphism.*** How could it happen? C++ supports a mechanism known as ***virtual function*** to achieve run time polymorphism. Please refer Fig. 9.1.

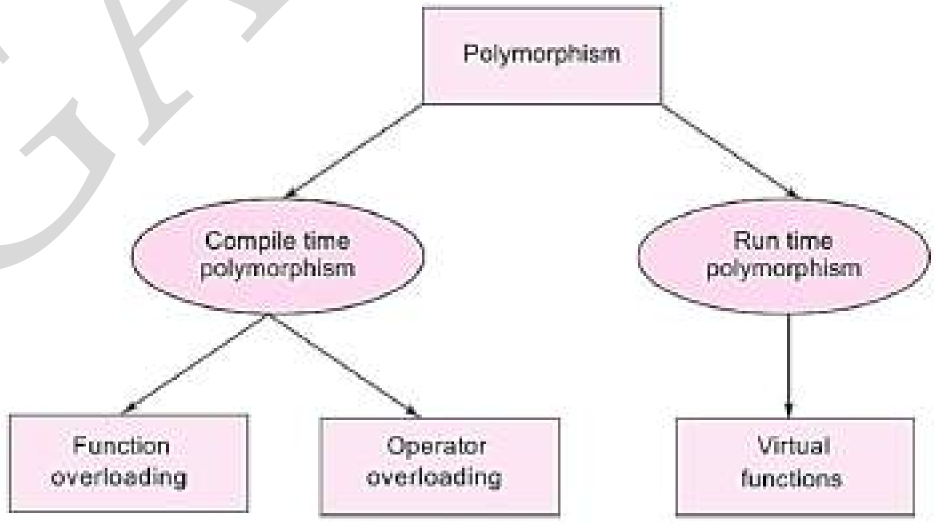


Fig. 9.1 *Achieving polymorphism*

At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as *late binding.* It is also known as *dynamic binding* because the selection of the appropriate function is done dynamically at run time.

Dynamic binding is one of the powerful features of C++. This requires the use of pointers to objects. We shall discuss in detail how the object pointers and virtual functions are used to implement dynamic binding.

Virtual Functions

As mentioned earlier, polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes.

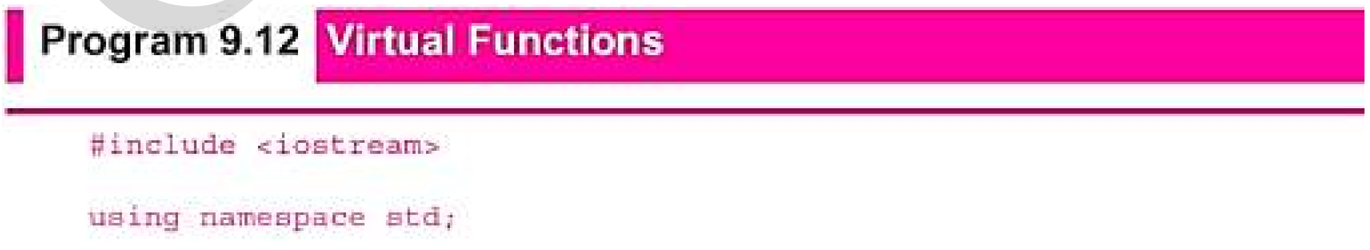
This necessitates the use of a single pointer variable to refer to the objects of different classes. Here, we use the pointer to base class to refer to all the derived objects. But, we just discovered that a base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class.

The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer. How do we then achieve polymorphism? It is achieved using what is known as **'virtual' functions.**

When we use the same function name in both the base and derived classes, the function in base class is declared as *virtual* using the keyword **virtual** preceding its normal declaration.

When a function is made **virtual, C++** determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer.

Thus, by making the base pointer to point to different objects, we can execute different versions of the **virtual** function. Program 9.12 illustrates this point.



Program 9.12 Virtual Functions

```
#include <iostream>

using namespace std;
```

```
class Base
{
        public:
            void display() {cout << "\n Display base ";}
            virtual void show() {cout << "\n show base";}
};
class Derived : public Base
{
        public:
            void display() {cout << "\n Display derived";}
            void show() {cout << "\n show derived";}
};

int main()
{
            Base B;
            Derived D;
            Base *bptr;

            cout << "\n bptr points to Base \n";
            bptr = &B;
            bptr -> display();          // calls Base version
            bptr -> show();             // calls Base version

            cout << "\n\n bptr points to Derived\n";
            bptr = &D;
            bptr -> display();          // calls Base version
            bptr -> show();             // calls Derived version

            return 0;
}
```

The output of Program 9.12 would be:

```
bptr points to Base

Display base
Show base

bptr points to Derived

Display base
Show derived
```

Note *When **bptr** is made to point to the object **D**, the statement*

```
bptr -> display();
```

*calls only the function associated with the **Base** (i.e., **Base :: display( )**), whereas the statement*

```
bptr -> show();
```

*calls the **Derived** version of **show()**. This is because the function **display()** has not been made **virtual** in the **Base** class.*

One important point to remember is that, we must access **virtual** functions through the use of a pointer declared as a pointer to the base class. Why can't we use the object name (with the dot operator) the same way as any other member function to call the virtual functions? We can, but remember, run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

Let us take an example where **virtual** functions are implemented in practice. Consider a book shop which sells both books and video-tapes. We can create a class known as. **media** that stores the title and price of a publication. We can then create two derived classes, one for storing the number of pages in a book and another for storing the playing time of a tape. Figure below shows the class hierarchy for the book shop.

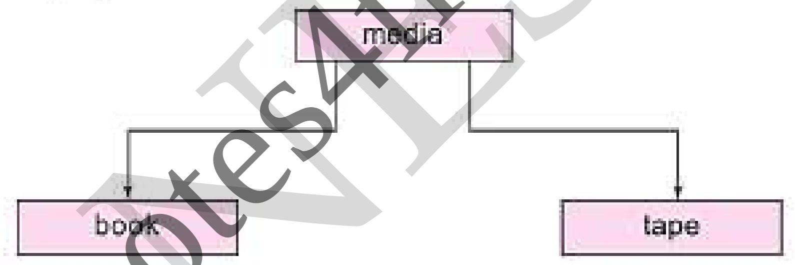


Fig. 9.2 The class hierarchy for the book shop

The classes are implemented in Program 9.13. A function **display()** is used in all the classes to display the class contents. Notice that the function **display()** has been declared virtual in media, the base class.

In the **main** program we create a heterogeneous list of pointers. of type **media** as shown below:

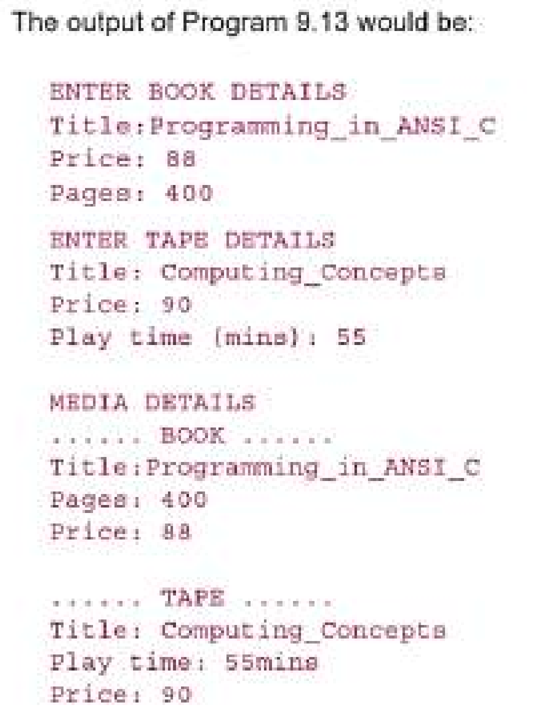
```
media *list[2]={&book1,&tape1};
```

The base pointers **list[0]** and **list[1]** are initialized with the addresses of objects **book1** and **tape1** respectively.

Program 9.13 Runtime Polymorphism

```
#include <iostream>
#include <cstring>

using namespace std;

class media
{
        protected:
            char title[50];
            float price;
        public:
            media(char *s, float a)
            {
               strcpy(title, s);
               price = a;
            }

            virtual void display() { } // empty virtual function
};

class book: public media
{
            int pages;
        public:
            book(char *s, float a, int p):media(s,a)
            {
               pages = p;
            }
            void display();
};
class tape :public media
{
            float time;
        public:
            tape(char * s, float a, float t):media(s, a)
            {
               time = t;
            }
            void display();
};

void book :: display()
{
            cout << "\n Title: " << title;
            cout << "\n Pages: " << pages;
```

```
                cout << "\n Price: " << price;
    }

    void tape :: display()
    {
                cout << "\n Title: " << title;
                cout << "\n play time: " << time << "mins";
                cout << "\n price: " << price;
    }

    int main()
    {
                char * title = new char[30];
                float price, time;
                int pages;

                // Book details
                cout << "\n ENTER BOOK DETAILS\n";
                cout << " Title: "; cin >> title;
                cout << " Price: "; cin >> price;
                cout << " Pages: "; cin >> pages;

                book book1(title, price, pages);

                // Tape details
                cout << "\n ENTER TAPE DETAILS\n";
                cout << " Title: "; cin >> title;
                cout << " Price: "; cin >> price;
                cout << " Play time (mins): "; cin >> time;
                tape tape1(title, price, time);

                media* list[2];
                list[0] = &book1;
                list[1] = &tape1;

                cout << "\n MEDIA DETAILS";

                cout << "\n ...... BOOK ...... ";
                list[0] -> display(); // display book details

                cout << "\n ...... TAPE ...... ";

                list[1] -> display(); // display tape details

                result 0;
    }
```


The output of Program 9.13 would be:

```
ENTER BOOK DETAILS
Title:Programming_in_ANSI_C
Price: 88
Pages: 400

ENTER TAPE DETAILS
Title: Computing_Concepts
Price: 90
Play time (mins): 55

MEDIA DETAILS
...... BOOK ......
Title:Programming_in_ANSI_C
Pages: 400
Price: 88

...... TAPE ......
Title: Computing_Concepts
Play time: 55mins
Price: 90
```

<u>Rules for Virtual Functions</u>

When virtual functions are created for implementing late binding, we should observe some basic rules that satisfy the compiler requirements:

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes,

C++ considers them as overloaded functions, and the virtual function mechanism is ignored.

7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Extra program

```
// virtual functions accessed from pointer
#include <iostream>
using namespace std;
class Base //base class
{
public:
      virtual void show() //virtual function
      {
            cout << "Base\n";
      }
};
class Derv1 : public Base //derived class 1
{
public:
      void show()
      {
            cout << "Derv1\n";
      }
};
class Derv2 : public Base //derived class 2
{
public:
      void show()
      {
            cout << "Derv2\n";
      }
};
int main()
{
      Derv1 dv1; //object of derived class 1
      Derv2 dv2; //object of derived class 2
      Base* ptr; //pointer to base class
      ptr = &dv1; //put address of dv1 in pointer
      ptr->show(); //execute show()
      ptr = &dv2; //put address of dv2 in pointer
      ptr->show(); //execute show()
      return 0;
}
```

Pure Virtual Functions

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a *placeholder.*

For example, we have not defined any object of class media and therefore the function display() in the base class has been defined 'empty'. Such functions are called "do-nothing" functions.

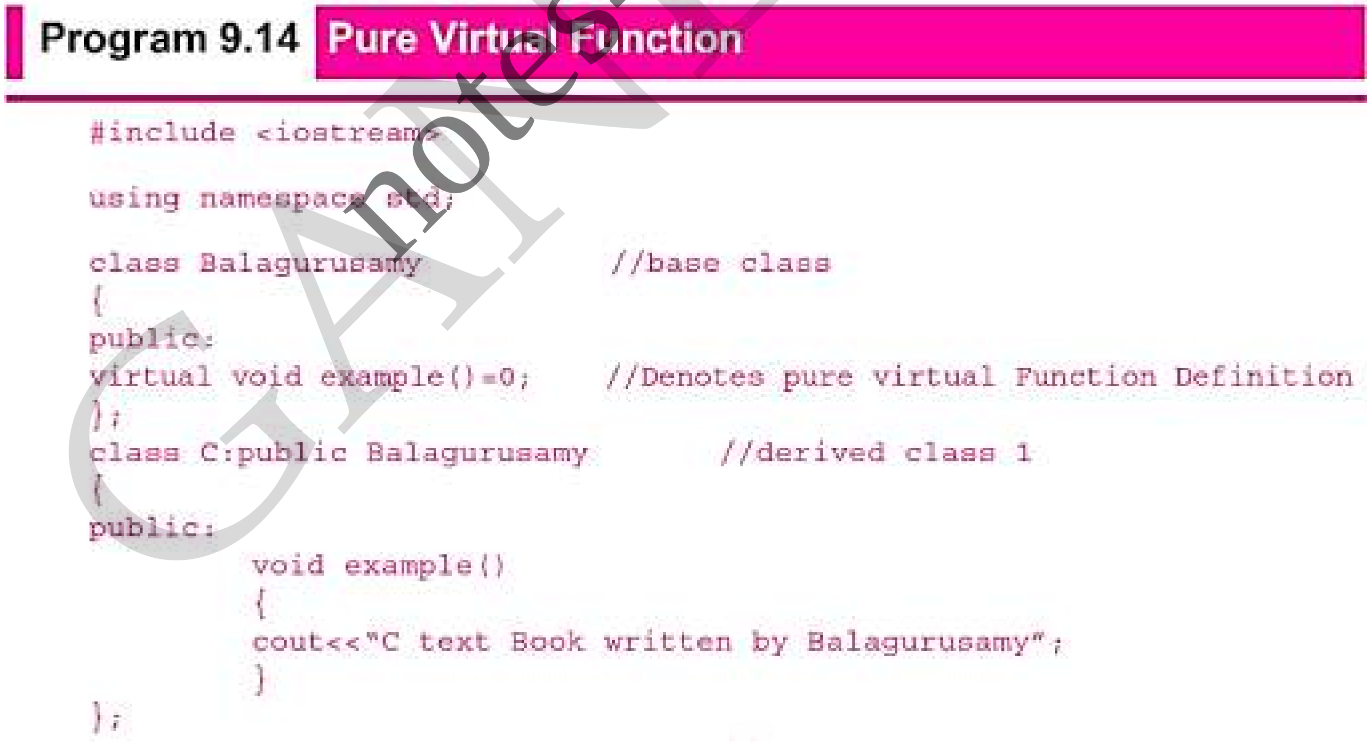
A "do-nothing" function may be defined as follows:

virtual void display ( ) = 0;

Such functions are called ***pure virtual* functions**. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function.

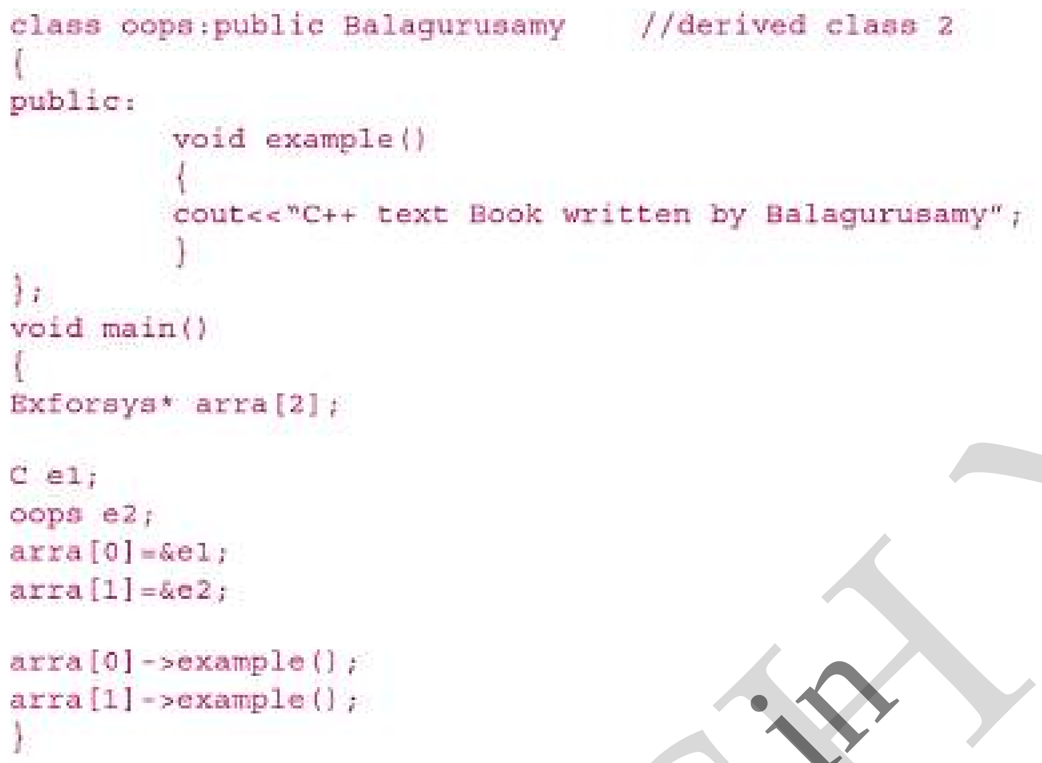
Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called *abstract base classes.* The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

Program 9.14 demonstrates the use of pure virtual function:

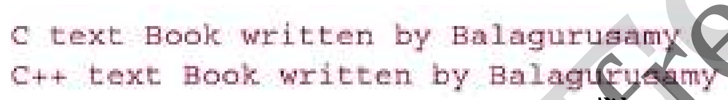


Program 9.14 Pure Virtual Function

```
#include <iostream>

using namespace std;

class Balagurusamy          //base class
{
public:
virtual void example()=0;    //Denotes pure virtual Function Definition
};
class C:public Balagurusamy       //derived class 1
{
public:
          void example()
          {
          cout<<"C text Book written by Balagurusamy";
          }
};
```



```
class oops:public Balagurusamy     //derived class 2
{
public:
          void example()
          {
          cout<<"C++ text Book written by Balagurusamy";
          }
};
void main()
{
Exforsys* arra[2];

C e1;
oops e2;
arra[0]=&e1;
arra[1]=&e2;

arra[0]->example();
arra[1]->example();
}
```



```
C text Book written by Balagurusamy
C++ text Book written by Balagurusamy
```

Extra program

```
#include <iostream>
using namespace std;
class A
{
public:
     void virtual display()=0;
};
class B:public A
{
public:
     void display()
     {
          cout<<"derived class";
     }
};
int main()
{
     A* ap;
     B b;
     ap=&b;
     ap->display();//invokes derived function.
     return 0;
}
```

r = 6.403124
theta = 51.340073 degree

Chapter 8

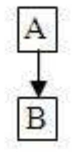
Review Questions

8.1: **What does inheritance mean in C++?**

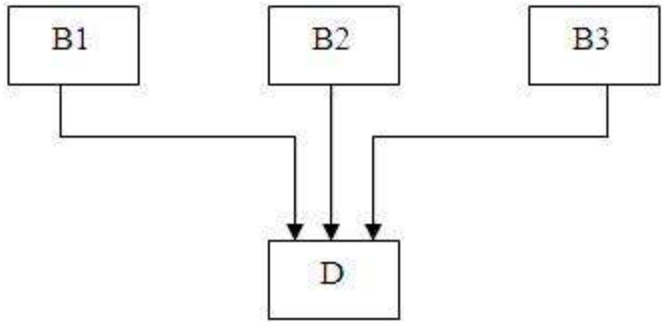
Ans:The mechanism of deriving a new class from an old one is called inheritance. The old class is referred to as the base class and the new one is called derived class.

8.2: **What are the different forms of inheritance? Give an example for each.**

Ans:Different forms of inheritence:
1. Single inheritence : Only one derived class inherited from only one base class is called single inheritence.
Example: Let A is a base class
and B is a new class derived from A



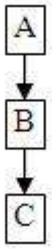
This is written in program as following:
class A {……….};
class B : Public A {……..};
2.Multiple inheritence : A class can inherit the attributes of two or more classes. This is known as multiple inheritence.
Example :



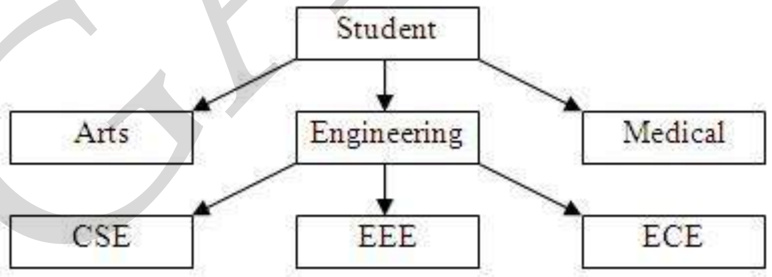
Class D : visibilityB1, visibility.B2, visibility B3

```
            {
                    (Body of D)
                  }
```

* visibility may be public or private.
3.Multilevel inheritance : It a class is derived from a base class, and another class is derived from this derived class and so on, then it is called multilevel inheritance.



Example :
Class A { ….. };
Class B : Public A { ….. };
Class C : Private B{ ….. };
4.Hierarchical inheritance: It the inheritance follows the hierarchical design of a program, then it is called hierarchical inheritance.
Example :



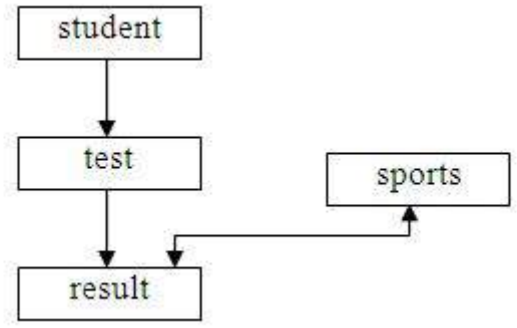
This design is implemented in program as follows:
Class student { …… }; // base class,
Class Arts : Public student { ……};
Class Medical : Public student {…….};
Class Engineering : Public student { ……};
Class CSE : Public Engineering { ……. };

Class EEE : Public Engineering { …… };
Class ECE : Public Engineering { …… };

* here all inheritance are considered as public you can private inheritance also. as you wish.

5.Hybrid inheritance: When multi level and multiple inheritances are applied to an inheritance, then it is called Hybrid inheritance.

Example :



In program :
Class student {……};
Class test : public student {……};
Class result : public test {…….};
Class result : public sports {…….};

8.3: **Describe the syntax of the single inheritance in C++.**

Ans:Syntax of single inheritance:
Class Derived name : visibility mode Base_class name

```
        {
            Body of derived class
        };
```

* visibility mode may be public or private.
or protected

8.4: **We know that a private member of a base class is not inheritable. Is it anyway possible for the objects of a derived class to access the private members of the base class? If yes, how? Remember, the base class cannot be modified.**

Ans:Yes. It is possible for the objects of derived class to access the private member of the base class by a member function of base class which is public. The following example explains this :

```
#include<iostream.h>
class B
{
     int a; // a is private that can not be inherited.
     public:
            int get_a();
            void set_a();
};
class D:public B
{
        int b;
        public:
          void display_a();
};

void D :: display_a()
{
    cout<<" a = "<<get_a()<<"\n"; // a is accessed by member function get_a().
}

void B :: set_a()
{
    a=156271;
}

int B :: get_a()
{
    return a;
}

void main()
{
         D d;
         d.set_a();
         d.display_a();
}
```

8.5: **How do the properties of the following two derived classes differ?**
(a) class D1: private B(// ….);
(b) class D2: public B(//….);

Ans:(a) Private member of B can not be inherited in D1 Protected member of B become private in D1 public member of B become private in D1.
(b) Private member of B can not be inherited in D2 Protected member of B remains protected in D2 Public member of B remains public in D2

8.6: **When do we use the protected visibility specifier to a class member?**

Ans:When we want access a data member by the member function within its class and by the member functions immediately derived from it, then we use visibility modifier protected.

8.7: **Describe the syntax of multiple inheritance. When do we use such an inheritance?**

Ans: **Syntax :**

```
Class D : Visibility B1, Visibility B2, ... ... ...
  {
     (Body of D)
  }
```

Then we want of combine several feature to a single class then we use multiple inheritance.

8.8: **What are the implications of the following two definitions?**
(a) class A: public B, public C(//….);
(b) class A: public C, public B(//….);

Ans:Two are same.

8.9: **WWhat is a virtual base class?**

Ans:Whey multiple paths between a bass class and a derived class are exist then this base class is virtual base class. A base class can be made as virtual just adding 'virtual' keyword before this base class.

8.10: **When do we make a class virtual?**

Ans:To avoid the duplication of inherited members due to multiple paths between base and derived classes we make base class virtual.

8.11: **What is an abstract class?**

Ans:An abstract class is one that is not used to create objects.

8.12: **In what order are the class constructors called when a derived class object is created?**

Ans:According to the order of derived class header lines

8.13: **Class D is derived from class B. The class D does not contain any data members of its own. Does the class D require constructors? If yes, why?**

Ans:D does not require any construct or because a default constructor is always set to class by default.

8.14: **What is containership? How does it differ from inheritance?**

Ans:Containership is another word for composition. That is, the HAS-A relationship where A has-a member that is an object of class B.
Difference : In inheritance the derived class inherits the member data and functions from the base class and can manipulate base public/protected member data as its own data. By default a program which constructs a derived class can directly access the public members of the base class as well. The derived class can be safely down cast to the base class, because the derived is-a" base class.

Container : a class contains another object as member data. The class which contains the object cannot access any protected or private members of the contained class(unless the container it was made a friend in the definition of the contained class).The relationship between the container and the contained object is "has-a" instead of "is-a".

8.15: **Describe how an object of a class that contains objects of other classes created?**

Ans:By inheriting an object can be created that contains the objects of other class.

Example :

```
class  A
{
        int a;
       public:
             void dosomething();
};

class B: class A
{
         int  b;
        public:
        void donothing();
};
```

Now if object of B is created ; then if contains:
1.   void dosomething ( );
2.   int b;
3.   void donothing ( );

8.16: **State whether the following statements are TRUE or FALSE:**
(a) Inheritance helps in making a general class into a more specific class.
(b) Inheritance aids data hiding.
(c) One of the advantages of inheritance is that it provides a conceptual framework.
(d) Inheritance facilitates the creation of class libraries.
(e) Defining a derived class requires some changes in the base class.
(f) A base class is never used to create objects.
(g) It is legal to have an object of one class as a member of another class.
(h) We can prevent the inheritance of all members of the base class by making base class virtual in the definition of the derived class.

Ans:
(a) TRUE
(b) FALSE
(c) TRUE
(d) TRUE
(e) FALSE
(f) TRUE
(g) TRUE
(h) FALSE

Debugging Exercises

8.1: **Identify the error in the following program.**

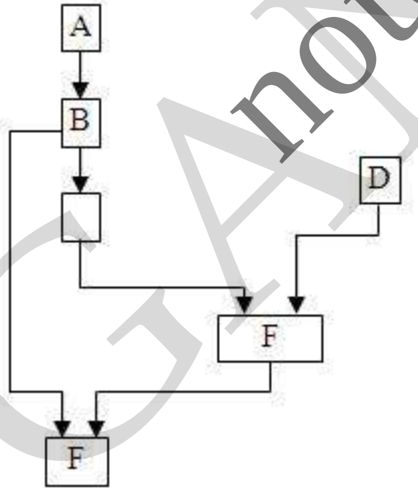
```
#include <iostream.h>;
class Student {
      char* name;
      int rollNumber;
public:
      Student() {
           name = "AlanKay";;
           rollNumber = 1025;
      }

       void setNumber(int no) {
            rollNumber = no;
       }
       int getRollNumber() {
           return rollNumber;
      }
};

class AnualTest: Student {
          int mark1, mark2;
public:
      AnualTest(int m1, int m2)
            :mark1(m1), mark2(m2) {
     }
     int getRollNumber() {
        return Student::getRollNumber();
     }
};
void main()
{
    AnualTest test1(92 85);
    cout<< test1.getRollNumber();
}
```

Solution: Constructor and Private (data & function) can not be inherited.

8.2: **Identify the error in the following program.**

```
#include <iostream.h>;
class A
{
public:
    A()
    {

     cout<< "A";
    }
};

class B: public A
{
public:
    B()
     {
         cout<< "B";
     }
};
class C: public B
{
public:
    C()
     {
         cout << "C";
     }
};
class D
{
public:
      D()
       {
             cout << "D";
       }
};
class E: public C, public D
{
public:
      E()
       {
             cout<< "D";
       }
};
class F: B, virtual E
{
public:
      F()
       {
             cout<< "F";
       }
};
void main()
{
     F f;
}
```

Solution: The inheritance can be represented as follows :



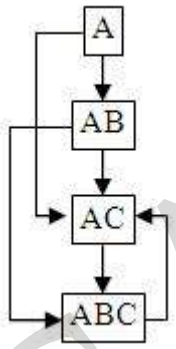
Here B is virtual, but not E.

8.3: **Identify the error in the following program.**

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

```
class A
{
    int i;
};

class AB: virtual A
{
        int j;
};
class AC: A, ABAC
{
        int k;
};
class ABAC: AB, AC
{
        int l;
};
void main()
{
    ABAC abac;
    cout << "sizeof ABAC:" << sizeof(abac);
}
```

Solution: The inheritance can be represented as follows:



Class AC: A, Here there is no identification of ABAC. If we write class ABAC; after #include it will not show any error massage.

8.4: **Find errors in the following program. State reasons.**

```
// Program test
#include <iostream.h>
class X
{
     private:
        int x1;
     Protected:
        int x2;
      public:
        int x3;
};

class Y: public X
{
        public:
          void f()
          {
                int y1,y2,y3;
                y1 = x1;
                y2 = x2;
                y3 = x3;
          }
};
class Z: X
{
       public:
          void f()
          {
                int z1,z2,z3;
                z1 = x1;
                z2 = x2;
                z3 = x3;
          }
};
main()
{
    int m,n,p;
    Y y;
    m = y.x1;
    n = y.x2;
    p = y.x3;
    Z z;
    m = z.x1;
    n = z.x2;
    p = z.x3;
}
```

Solution: Here x1 is private, so x1 cannot be inherited.
y1 = x1; is not valid
z1 = x1; is not valid
m = y, x1; is not valid
m = z, x1; is not valid

8.5: **Debug the following program.**

```
// Test program
#include <iostream.h>
class B1
{
      int b1;
      public:
        void display();
        {
               cout << b1 <<"\n";
        }
};

class B2
{
      int b2;
      public:
        void display();
        {
               cout << b2 <<"\n";
        }
};
class D: public B1, public B2
{
       //nothing here
};
main()
{
    D d;
    d.display()
    d.B1::display();
    d.B2::display();
}
```

Programming Exercises

8.1: **Assume that a bank maintains two kinds of accounts for customers, one called as savings and the other as current account. The savings account provides compound interest and withdrawal facilities but no cheque book facility. The current account provides cheque book facility but no interest. Current account holders should also maintain a minimum balance and if the balance falls below this level a service charge is imposed.**
Create a class account that stores customer name, account number and type of account. From this derive the classes cur_acct and sav_acct to make them more specific to their requirements. Include necessary member functions in order to achieve the following tasks:
(a) Accept the deposit from a customer and update the balance.
(b) Display the balance.
(c) Compute and deposit interest.
(d) Permit withdrawal and update the balance.
(e) Check for the minimum balance, impose penalty, necessary and update the balance.

Do not use any constructors. Use member functions to initialize class members.

Solution:

```
#include<iostream.h>
#include<stdio.h>
#include<string.h>
#include<math.h>
#define minimum 500
#define service_charge 100
#define r 0.15

class account
{
        protected:
         char name[100];
                  int ac_number;
         char ac_type[100];
               public:
        void creat( char *t);
};

void account::creat(char *t)
{

       cout<<" Enter customer name :";
        gets(name);
        strcpy(ac_type,t);
        cout<<" Enter account number :";
        cin>>ac_number;
}
class cur_acct: public account
{
           private:
            float balance;
           public:
            void deposite(float d);
            void withdraw(float w);
            void display();
};
void cur_acct::deposite(float d)
{
        balance=d;
}

void cur_acct::withdraw(float w)
{
          if(balance<w)
        cout<<" sorry your balance is less than your withdrawal amount \n";
         else
```

```
            {
                  balance-=w;
                  if(balance<minimum)
 cout<<"\n your current balance is :"<<balance<<" which is less    than"<<minimum<<"\n
your account is discharged by "<<service_charge<<"TK \n"<<" You must store
"<<minimum<<"TK to avoid discharge \n "<<" Do you want to withdraw ? press 1 for yes
press 0 for no \n"<<" what is your option ?";
                int test;
                         cin>>test;
                  if(test==0)
                    balance+=w;
         }

}

void cur_acct::display()
{
               cout<<"\n Now balance = "<<balance<<"\n";
}
class sav_acct:public account
{
                         float balance;
                        int d,m,y;
               public:
               void deposite(float d);
               void withdraw(float w);
              void display();
              void set_date(int a,int b,int c){d=a;m=b;y=c;}
              void interest();
};

void sav_acct::deposite(float d)
{
          int x,y,z;
          cout<<" Enter today's date(i,e day,month,year) : ";
         cin>>x>>y>>z;
          set_date(x,y,z);
          balance=d;
}

       void sav_acct::withdraw(float w)
{
                if(balance<w)
              cout<<" sorry your balance is less than your withdrawal amount \n";
               else
               {
                      balance-=w;

                               if(balance<minimum)
                       {
 cout<<"\n your current balance is :"<<balance<<" which is   less than"<<minimum<<"\n
your account is discharged by "<<service_charge<<"TK \n"<<" You must store
"<<minimum<<"TK to avoid discharge \n "<<" Do you want to withdraw ? press 1 for yes
press 0 for no \n"<<" what is your option ?";

                    int test;
                     cin>>test;
                    if(test==0)
                     balance+=w;
               }
         }

}
void sav_acct::display()
{
          cout<<"\n Now balance = "<<balance;
}
void sav_acct::interest()
{
       int D[12]={31,28,31,30,31,30,31,31,30,31,30,31};
       int d1,y1,m1;
       cout<<" Enter today's date :(i,e day,month,year) ";
       cin>>d1>>m1>>y1;
       int iday,fday;
       iday=d;
      fday=d1;
     for(int i=0;i<m1;i++)
     {
            fday+=D[i];
     }
    for(i=0;i<m;i++)
    {
             iday+=D[i];
    }
  int tday;
    tday=fday-iday;
    float ty;
    ty=float(tday)/365+y1-y;
    float intrst;

     intrst=ty*r*balance;
     cout<<" Interest is : "<<intrst<<"\n";
     balance+=intrst;
}

int main()
{
      sav_acct santo;
      santo.creat("savings");
      float d;
      cout<<" Enter your deposit amount : ";
      cin>>d;
      santo.deposite(d);
      santo.display();
      int t;
     cout<<"\n press 1 to see your interest : \n"
          <<" press 0 to skip : ";

      cin>>t;

    if(t==1)
     santo.interest();

    cout<<"\n Enter your withdrawal amount :";
    float w;
    cin>>w;
     santo.withdraw(w);
      santo.display();
     return 0;
}
```

output

```
Enter customer name :Rimo
Enter account number :10617
Enter your deposit amount : 10000
Enter today's date(i,e day,month,year) : 15   7   2010

Now balance = 10000
press 1 to see your interest :
press 0 to skip : 1
Enter today's date :(i,e day,month,year) 15   8   2010

Interest is : 135.61644
Enter your withdrawal amount :500
Now balance = 9635.616211
```

8.2: **Modify the program of exercise 8.1 to include constructors for all three classes.**

Solution:

```
#include<iostream.h>
#include<stdio.h>
#include<string.h>
#include<math.h>
#define minimum 500
#define service_charge 100
#define r 0.15

```

```
class account
{
         protected:
          char name[100];
           int ac_number;
          char ac_type[100];
        public:
         account( char *n,char *t,int no);
};
 account::account(char *n,char *t,int no)
{
          strcpy(name,n);
          strcpy(ac_type,t);
          ac_number=no;

}

       class cur_acct: public account
{
      private:
       float balance,d,w;
      public:
                void withdraw(float ww);
                void deposit(float d){balance=d;}
     cur_acct(char   *n,char  *t,int  number,float  dp,float  wd):
     account(n,t,number)
         {
                    d=dp;
                    w=wd;
                    deposit(d);
                    withdraw(w);

                  }
                 void display();
};

void cur_acct::withdraw(float ww)
{

       if(balance<ww)
       cout<<" sorry your balance is less than your withdrawal amount \n";
       else
         {
                  balance-=ww;
                 if(balance<minimum)
                  {
 cout<<"\n your current balance is :"<<balance<<" which is less than"<<minimum<<"\n your
account is discharged by "<<service_charge<<"TK \n"<<" You must store
"<<minimum<<"TK to avoid discharge \n "<<" Do you want to withdraw ? press 1 for yes
press 0 for no \n"<<" what is your option ?";
                              int test;
                              cin>>test;
                              if(test==0)
                               balance+=w;
                    }
                             else
                     ;
                }
}

        void cur_acct::display()
{
             cout<<"\n Now balance = "<<balance<<"\n";
}
class sav_acct:public account
{
             float balance;
             int d,m,y;
             public:
             void deposite(float d){balance=d;set_date();}
             void withdraw(float w);
             void display();
             void set_date(){d=12;m=1;y=2010;}
             void interest();
             sav_acct(char  *n,char  *t,int  number,float  dp,float  wd):
          account(n,t,number)
              {
                      float d,w;
                                d=dp;
                      w=wd;
                      deposite(d);
                     interest();
                      withdraw(w);

             }
};
     void sav_acct::withdraw(float w)
{
                       if(balance<w)
               cout<<" sorry your balance is less than your withdrawal amount \n";
                      else
                      {
                              balance-=w;
                              if(balance<minimum)
                              {
cout<<"\n your current balance is :"<<balance<<" which is less than"<<minimum<<"\n your
account is discharged by "<<service_charge<<"TK \n"<<" You must store
"<<minimum<<"TK to avoid discharge \n "<<" Do you want to withdraw ? press 1 for yes
press 0 for no \n"<<" what is your option ?";
                                     int test;
                                     cin>>test;
                                     if(test==0)
```

```
                        balance+=w;
                    }
                       else
                    ;
               }

}
void sav_acct::display()
{
        cout<<"\n Now balance = "<<balance;
}
void sav_acct::interest()
{
      int D[12]={31,28,31,30,31,30,31,31,30,31,30,31};
      int d1,y1,m1;
      cout<<" Enter today's date :(i,e day,month,year) ";
       cin>>d1>>m1>>y1;
      int iday,fday;
      iday=d;
      fday=d1;
      for(int i=0;i<m1;i++)
       {
               fday+=D[i];
              }
      for(i=0;i<m;i++)
      {
            iday+=D[i];
      }
     int tday;
     tday=fday-iday;
     float ty;
     ty=float(tday)/365+y1-y;
     balance=balance*pow((1+r),ty);
}

int main()
{

     float d;
      cout<<" Enter customer name :";
      char name[100];
      gets(name);
      cout<<" Enter account number :";
      int number;
      cin>>number;
      cout<<" Enter your deposit amount : ";
      cin>>d;

     cout<<" Enter your withdrawal amount :";
     float w;
     cin>>w;
```

```
             //cur_acct s("current",name,number,d,w);
            //s.display();
             sav_acct c("savings",name,number,d,w);
            c.display();
         return 0;
   }
```

output

```
Enter customer name :mehedi

Enter account number :1457

Enter your deposit amount : 5000

Enter your withdrawal amount :1200

Enter today's date :(i,e day,month,year)   13   7   2010
Now balance = 4160.875977
```

8.3: **An educational institution wishes to maintain a database of its employees. The database is divided into a number of classes whose hierarchical relationships are shown in following figure. The figure also shows the minimum information required for each class. Specify all classes and define functions to create the database and retrieve individual information as and when required.**

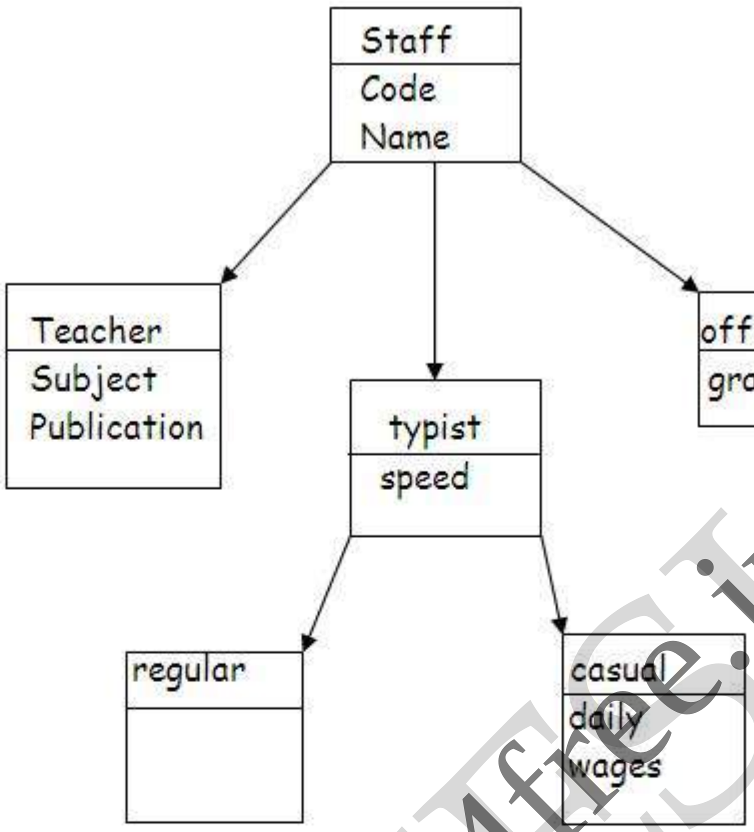


fig: class relationships (for exercise 8.3)

Solution:

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

class staff
{
          public:
            int code;
            char name[100];
                   public:
          void set_info(char *n,int c)
            {
                    strcpy(name,n);
                    code=c;
                    }
};

class teacher : public staff
```

```
{
            protected:
           char sub[100],publication[100];
            public:
            void set_details(char *s,char *p)
            {
                   strcpy(sub,s);strcpy(publication,p);
            }
           void show()
            {
                                       cout<<"name"<<setw(8)<<"code"<<setw(15)<<"subject"<<setw(25)
<<"publication"<<endl<<name<<setw(8)<<code<<setw(25)<<sub<<setw(22)<<publication<<endl;
           }
};

class officer:public staff
{
         char grade[100];
          public:
         void set_details(char *g)
          {
                 strcpy(grade,g);
          }

         void show()
                     {
            cout<<" name "<<setw(15)<<"code"<<setw(15)<<"Category "<<endl
            <<name<<setw(10)<<code<<setw(15)<<grade<<endl;
         }
};
       class typist: public staff
{
        protected:
        float speed;
          public:
          void set_speed(float s)
            {
                   speed=s;
            }
};
class regular:public typist
{
     protected:
     float wage;
      public:
      void set_wage(float w){wage=w;}
      void show()
       {
             cout<<"name"<<setw(16)<<"code"<<setw(15)<<"speed"<<setw(15)
<<"wage"<<endl<<name<<setw(10)<<code<<setw(15)<<speed
<<setw(15)<<wage<<endl;
```

```
        }
};
class causal:public typist
{
                float wage;
        public:
        void set_wage(float w){wage=w;}
        void show()
        {
                cout<<"name"<<setw(16)<<"code"<<setw(15)<<"speed"<<setw(15)
<<"wage"<<endl<<name<<setw(10)<<code<<setw(15)<<speed
<<setw(15)<<wage<<endl;
        }

};

int main()
{

    teacher t;
        t.set_info("Ataur",420);
        t.set_details("programming with c++","Tata McGraw Hill");

        officer o;
        o.set_info("Md. Rashed",222);
        o.set_details("First class");

                regular rt;
        rt.set_info("Robiul Awal",333);
        rt.set_speed(85.5);
        rt.set_wage(15000);

    causal ct;
ct.set_info("Kawser Ahmed",333);
ct.set_speed(78.9);
ct.set_wage(10000);
    cout<<"  About teacher: "<<endl;
    t.show();
    cout<<" About officer:"<<endl;
     o.show();
    cout<<" About regular typist :"<<endl;
    rt.show();
    cout<<" About causal typist :"<<endl;
    ct.show();

    return 0;
}
```

output

About teacher:

```
name          code          subject                              publication

Ataur          420          programming with c++        Tata McGraw Hill

About officer:

name                    code            Category

Md. Rashed          222              First class

About regular typist :

name                  code          speed               wage

Robiul Awal       333            85.5                  15000

About causal typist :

name                  code          speed                  wage

Kawser Ahmed   333          78.900002         10000
```

8.4: **The database created in exercise 8.3 does not include educational information of the staff. It has been decided to add this information to teachers and officers (and not for typists) which will help management in decision making with regard to training, promotions etc. Add another data class called education that holds two pieces of educational information namely highest qualification in general education and highest professional qualification. This class should be inherited by the classes teacher and officer. Modify the program of exercise 8.19 to incorporate these additions.**

Solution:

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

class staff
{
          protected:
               int code;
               char name[100];
              public:
              void set_info(char *n,int c)
              {
                          strcpy(name,n);
                           code=c;
```

```
            }
};
class education:public staff
{
   protected:
          char quali[100];
           public:
           void set_qualification(char *q){strcpy(quali,q);}
};

class teacher : public education
{
             protected:
             char sub[100],publication[100];
             public:
                     void set_details(char *s,char *p)
              {
                  strcpy(sub,s);strcpy(publication,p);
              }
              void show()
              {
                     cout<<" name "<<setw(8)<<"code"<<setw(15)
                         <<"subject"<<setw(22)<<"publication"
                         <<setw(25)<<"qualification"<<endl
                         <<name<<setw(8)<<code<<setw(25)
                   <<sub<<setw(18)<<publication<<setw(25)<<quali<<endl;
               }
};

    class officer:public education
{
            char grade[100];
            public:
            void set_details(char *g)
            {
                 strcpy(grade,g);

            }

            void show()
            {
                 cout<<" name "<<setw(15)<<"code"<<setw(15)<<"Catagory "
                      <<setw(22)<<"Qualification"<<endl<<name<<setw(10)
                       <<code<<setw(15)<<grade<<setw(25)<<quali<<endl<<endl;
            }
};

class typist: public staff
{
      protected:
        float speed;
```

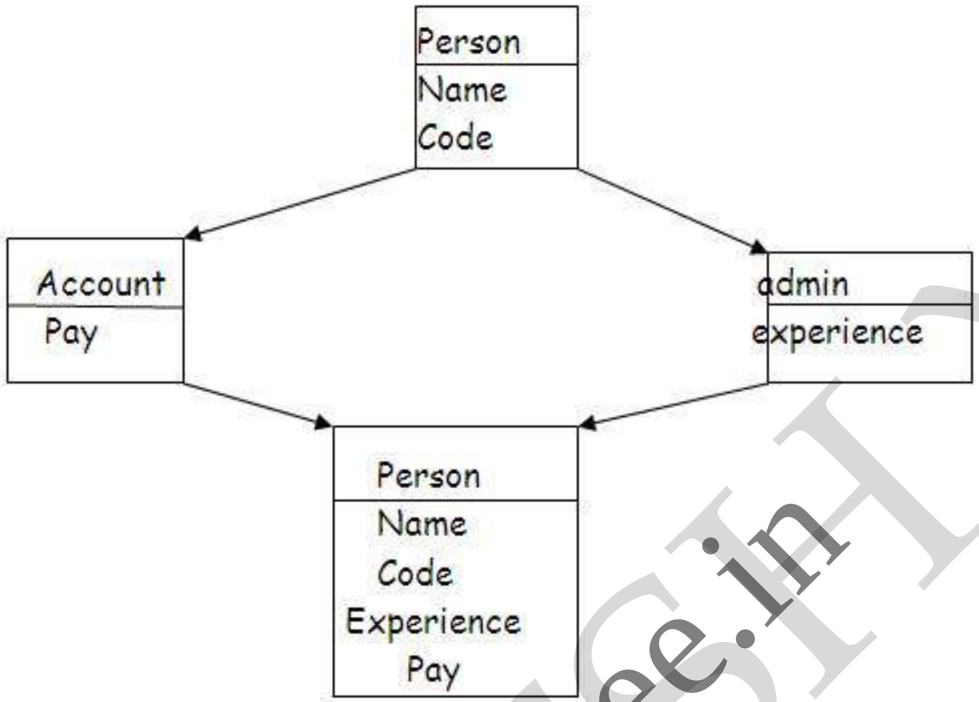
```
          public:
          void set_speed(float s)
                {
                     speed=s;
                }
};
class regular:public typist
{
      protected:
          float wage;
       public:
         void set_wage(float w){wage=w;}
         void show()
         {
                cout<<" name "<<setw(16)<<"code"<<setw(15)<<"speed"
                   <<setw(15)<<"wage"<<endl<<name<<setw(10)<<code
                   <<setw(15)<<speed<<setw(15)<<wage<<endl<<endl;
         }
};
class causal:public typist
{
       float wage;
        public:
         void set_wage(float w){wage=w;}
         void show()
         {
               cout<<" name "<<setw(16)<<"code"<<setw(15)<<"speed"
                   <<setw(15)<<"wage"<<endl<<name<<setw(10)<<code
                   <<setw(15)<<speed<<setw(15)<<wage<<endl<<endl;
         }

};

int main()
{

     teacher t;                          t.set_info("Ataur",420);
t.set_details("programming with c++"," Tata McGraw Hill");
t.set_qualification("PHD from programming ");

      officer o;

       o.set_info("Md. Rashed",222);
       o.set_details("First class");
       o.set_qualification("2 years experienced");

               regular rt;
                    rt.set_info("Robiul Awal",333);
rt.set_speed(85.5);
rt.set_wage(15000);

```

```
      causal ct;
          ct.set_info("Kawser Ahmed",333);
       ct.set_speed(78.9);
       ct.set_wage(10000);

    cout<<"  About teacher: "<<endl;
      t.show();

     cout<<" About officer:"<<endl;
      o.show();

      cout<<" About regular typist :"<<endl;
       rt.show();
      cout<<" About causal typist :"<<endl;
       ct.show();

    return 0;
}
```

output

```
About teacher:

name    code       subject                publication           qualification

Ataur    420  programming with c++   Tata McGraw Hill   PHD from programming-
About officer:

name                 code       Catagory                 Qualification

Md. Rashed      222        First class               2 years experienced

About regular typist :

name               code        speed            wage

Robiul Awal     333         85.5               15000
About causal typist :

name              code        speed            wage

Kawser           333         78.900002    10000
```

8.5: **Consider a class network of the following figure. The class master derives information from both account and admin classes which in turn derives information from the class person. Define all the four classes and write a program to create, update and display the information contained in master objects.**



Solution:

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

class staff
{
        protected:
         int code;
         char name[100];
         public:
         void set_info(char *n,int c)
         {
             strcpy(name,n);
             code=c;
         }
};

class education:public staff
{
        protected:
        char quali[100];
        public:
        void set_qualification(char *q){strcpy(quali,q);}
};

```

```
class teacher : public education
{
          protected:
         char sub[100],publication[100];
                  public:
          void set_details(char *s,char *p)
                  {
                strcpy(sub,s);strcpy(publication,p);

          }

                void show()
          {
                  cout<<"name"<<setw(8)<<"code"<<setw(15)<<"subject"<<setw(22)
<<"publication"<<setw(25)<<"qualification"<<endl<<name<<setw(8)
<<code<<setw(25)<<sub<<setw(18)<<publication<<setw(25)<<quail
<<endl;
         }
};

class officer:public education
{
        char grade[100];
         public:
         void set_details(char *g)
                  {
                  strcpy(grade,g);
         }

          void show()
          {
             cout<<"name"<<setw(15)<<"code"<<setw(15)<<"Catagory"
                <<setw(22)<<"Qualification"<<endl<<name<<setw(10)
               <<code<<setw(15)<<grade<<setw(25)<<quali<<endl<<endl;
                  }
};

class typist: public staff
{
       protected:
        float speed;
      public:
      void set_speed(float s)
                {
           speed=s;
       }
};
       class regular:public typist
{
          protected:
          float wage;
        public:
        void set_wage(float w){wage=w;}
        void show()
         {
                 cout<<"name"<<setw(16)<<"code"<<setw(15)<<"speed"<<setw(15)
<<"wage"<<endl<<name<<setw(10)<<code<<setw(15)<<speed
<<setw(15)<<wage<<endl<<endl;
         }
};
class causal:public typist
{
        float wage;
        public:
        void set_wage(float w){wage=w;}

        void show()
        {
                   cout<<"name"<<setw(16)<<"code"<<setw(15)<<"speed"<<setw(15)
<<"wage"<<endl<<name<<setw(10)<<code<<setw(15)<<speed
<<setw(15)<<wage<<endl<<endl;
       }

};

int main()
{

        teacher t;
          t.set_info("Ataur",420);
         t.set_details("programming with c++"," Tata McGraw Hill");
         t.set_qualification("PHD from programming ");
       officer o;
      o.set_info("Md. Rashed",222);
      o.set_details("First class");
      o.set_qualification("2 years experienced");
      regular rt;
      rt.set_info("Robiul Awal",333);
      rt.set_speed(85.5);
      rt.set_wage(15000);
          causal ct;
     ct.set_info("Kawser Ahmed",333);
     ct.set_speed(78.9);
     ct.set_wage(10000);
     cout<<"  About teacher: "<<endl;
      t.show();
      cout<<" About officer:"<<endl;
      o.show();
      cout<<" About regular typist :"<<endl;
      rt.show();
      cout<<" About causal typist :"<<endl;
      ct.show();
```

```
 
return 0;
}
```

output

```
name        code         Experience            payment

Hasibul    111            3 years                    1500tk
```

8.6: **In exercise 8.3 the classes teacher, officer, and typist are derived from the class staff. As we know we can use container classes in place of inheritance in some situations. Redesign the program of exercise 8.3 such that the classes teacher, officer and typist contain the objects of staff.**

Solution:

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

class staff
{
            public:
                     int code;
            char name[100];
            public:
            void set_info(char *n,int c)
             {
                    strcpy(name,n);
                    code=c;
            }
};

class teacher : public staff
{
        protected:
        char sub[100],publication[100];
        public:
        void set_details(char *s,char *p)
         {
              strcpy(sub,s);strcpy(publication,p);
         }
                void show()
         {
                  cout<<"name"<<setw(8)<<"code"<<setw(15)<<"subject"<<setw(25)
<<"publication"<<endl<<name<<setw(8)<<code<<setw(25)<<sub
<<setw(22)<<publication<<endl;
          }
};

       class officer:public staff
{
         char grade[100];
         public:
         void set_details(char *g)
         {
              strcpy(grade,g);
         }
        void show()
        {
           cout<<" name "<<setw(15)<<"code"<<setw(15)<<"Catagory "<<endl
                <<name<<setw(10)<<code<<setw(15)<<grade<<endl;
       }
};

class typist: public staff
{
      protected:
       float speed;
       public:
        void set_speed(float s)
        {
                speed=s;
       }
                void show()
                 {
                cout<<" name "<<setw(15)<<"code"<<setw(15)<<"speed"<<endl
                      <<name<<setw(10)<<code<<setw(15)<<speed<<endl<<endl;
        }
};

      int main()
{


       teacher t;
       t.set_info("Ataur",420);
       t.set_details("programming with c++"," Tata McGraw Hill");

       officer o;
       o.set_info("Md. Rashed",222);
       o.set_details("First class");

               typist tp;
      tp.set_info("Robiul Awal",333);
      tp.set_speed(85.5);

                cout<<"  About teacher: "<<endl;
```

```
     t.show();
     cout<<" About officer:"<<endl;
     o.show();
     cout<<" About  typist :"<<endl;
     tp.show();
    return 0;
}
```

output

```
About teacher:

name      code      subject                    publication

Ataur   420     programming with c++   Tata McGraw Hill

About officer:

name              code           Catagory

Md. Rashed     222            First class

About typist :

name               code           speed

Robiul Awal     333             85.5
```

8.7: **We have learned that OOP is well suited for designing simulation programs. Using the techniques and tricks learned so far, design a program that would simulate a simple real-world system familiar to you**

Solution:

```
#include<iostream.h>
#include<stdio.h>
#include<string.h>
#include<iomanip.h>
#include<conio.h>

char *sub[10]={"Bangla 1st paper","Bangla 2nd paper","English 1st paper",
                  "English 2nd paper","Mathematics","Religion",
                  "Physics","Chemistry","Sociology","Higher Mathematics"};

class student_info
{

        public:
          char  name[40];
           char  roll[20];
        public:
          void set_info();
};

void student_info::set_info()
{
         cout<<"Enter student name : ";
          gets(name);
          cout<<"Enter roll: ";
          gets(roll);
}

        class subject :public student_info
{

        public:
         float mark[10];

         public:
         void set_mark();
};

void subject::set_mark()
{
         cout<<" marks  of  : \n";
            for(int i=0;i<10;i++)
               {
                        cout<<sub[i]<<" = ? ";
                                  cin>>mark[i];
               }

}
      class conversion :public  subject
{
                 float gpa[10];
        char grade[20][20];
         public:
        void convert_to_gpa();
         void show();
};
        void conversion::convert_to_gpa()
{
                for(int i=0;i<10;i++)
                   {
                        if(mark[i]>=80)
                          {
                                         gpa[i]=5.00;
                                  strcpy(grade[i],"A+");
                             }
                            else if(mark[i]>=70 && mark[i]<80)
                            {
                                   gpa[i]=4.00;
                                            strcpy(grade[i],"A");
                             }
                             else if(mark[i]>=60 && mark[i]<70)
                              {
                                               gpa[i]=3.50;
                                     strcpy(grade[i],"A-");
                               }
                            else if(mark[i]>=50 && mark[i]<60)
                              {
                                       gpa[i]=3.00;
                                                strcpy(grade[i],"B");
                              }
                            else if(mark[i]>=40 && mark[i]<50)
                              {
                                                 gpa[i]=2.00;
   strcpy(grade[i],"C");
                               }
                                      else if(mark[i]>=33 && mark[i]<40)
                               {
                                          gpa[i]=1.00;
                                          strcpy(grade[i],"D");
                                }
                              else
                                {
                                             gpa[i]=0.00;
                                             strcpy(grade[i],"Fail");
                                 }
                     }
     }

 void conversion::show()
{
             cout<<" result of \n";
             cout<<"name :"<<name<<"\n";
             cout<<"Roll : "<<roll<<"\n";
          cout<<setw(25)<<"Subject"<<setw(17)<<"Marks"
                <<setw(14)<<"GPA"<<setw(12)<<"Grade \n";
             for(int i=0;i<10;i++)
              {
                           cout<<setw(25)<<sub[i]<<setw(15)<<mark[i]
                        <<setw(15)<<gpa[i]<<setw(10)<<grade[i]<<"\n";
              }
}
int main()
{
          clrscr();
```

```
    conversion A;
    A.set_info();
    A.set_mark();
    A.convert_to_gpa();
     A.show();
     getch();
    return 0;
}
```

output

```
Enter student name : santo

Enter roll: 156271

marks of  :

Bangla 1st paper = ? 74

Bangla 2nd paper = ? 87

English 1st paper = ? 45

English 2nd paper = ? 56

Mathematics = ? 87

Religion = ? 59

Physics = ? 75

Chemistry = ? 65

Sociology = ? 39

Higher Mathematics = ? 86

result of

name :santo

Roll : 156271

Subject                 Marks    GPA      Grade

Bangla 1st paper        74        4         A

Bangla 2nd paper        87       5        A+
```

English 1st paper	45	2	C
English 2nd paper	56	3	B
Mathematics	87	5	A+
Religion	59	3	B
Physics	75	4	A
Chemistry	65	3.5	A-
Sociology	39	1	D
Higher Mathematics	86	5	A+

Chapter 9

Review Questions

9.1: **What does polymorphism mean in C++ language?**

Ans:In short, polymorphism means one thing with several district forms.
–In details, using operators or functions in different ways, depending on what they are Operating on, is called polymorphism.

9.2: **How is polymorphism achieved at (a) compile time, and (b) run time?**

Ans:Polymorphism can be achieved at compile time by early binding. Early binding means an object is bound to its function call at compile time.
And we can achieve run time polymorphism by a mechanism known as virtual function.

9.3: **Discuss the different ways by which we can access public member functions of an object.**

Ans:We can access public member functions of an object by
(i) Object name and dot membership operator.
(ii) Pointer to object and function name.

9.4: **Explain, with an example, how you would create space for an array of objects using pointers.**

Ans:We can also create an array of objects using pointers. For example, the statement
item *ptr = new item [10]; // array of 10 objects.
creates memory space for an array of 10 objects of item.

9.5: **What does this pointer point to?**

Ans:'this' pointer point to invoking object.

9.6: **What are the applications of this pointer?**

Ans:One important application of the pointer this is to return the object it points to. For example, the statement.
return * this;
inside a member function will return the object that invoked the function.

9.7: **What is a virtual junction?**

Ans:When we use the same function name in both the base and derived classes the function in the base class is declared as virtual using the keyword virtual preceding its normal declaration.

9.8: **Why do we need virtual functions?**

Ans:It we need same function name at base class and derived class then, we need virtual function.

9.9: **When do we make a virtual function "pure"? What are the implications of making a function a pure virtual function?**

Ans:When a function is defined as empty, then this function is called do nothing function.
The implications of making a function a pure virtual function is to achieve run time polymorphism.

9.10: **State which of the following statements are TRUE or FALSE.**
(a) Virtual functions are used to create pointers to base classes.
(b) Virtual functions allow us to use the same junction call to invoke member functions of objects of different classes.

(c) A pointer to a base class cannot be made to point to objects of derived class.
(d) this pointer points to the object that is currently used to invoke a function.
(e) this pointer can be used like any other pointer to access the members of the object it points to.
(f) this pointer can be made to point to any object by assigning the address of the object.
(g) Pure virtual functions force the programmer to redefine the virtual function inside the derived classes.

Ans:
(a) TRUE
(b) TRUE
(c) FALSE
(d) TRUE
(e) TRUE
(f) TRUE
(g) TRUE

Debugging Exercises

9.1: **Identify the error in the following program.**

```
#include <iostream.h>;
class  Info
{
     char* name;
     int Number;
public:
     void getInfo()
     {
         cout << "Info::getInfo";
         getName();
     }

      void getName()
      {
          cout << "Info::getName";
      }
};

class Name: public Info
{
         char *name;
public:
        void getName()
      {
              cout << "Name::getName";
      }
};
```

```
void main()
{
    Info *P;
    Name n;
    P = n;
    p->getInfo();
}
/*
```

Solution: Here P=n will replace with P=&n in the main() function. Because P is a pointer.

9.2: **Identify the error in the following program.**

```
#include <iostream.h>;
class  Person
{
     int age;
public:
     Person()
     {
     }

      Person(int age)
      {
          this.age = age;
      }
      Person& operator < (Person &p)
      {
          return age < p.age? p: *this;
      }
      int getAge()
      {
         return age;
      }
};

void main()
{
    Person P1 (15);
    Person P2 (11);
    Person P3;
    //if P1 is less than P2
     P3 =  P1 < P2; P1.lessthan(P2)
     cout << P3.getAge();
}
/*
```

Solution: The function

```
person (int age)
{
        this.age = age;
}
```

should write like as…

```
person (int age)
{
        –this > age = age;
}
```

9.3: **Identify the error in the following program.**

```
#include <iostream.h>;
class  Human
{
public:
   Human()
   {
   }

    virtual -Human()
    {
      cout << "Human::-Human";
    }
};
class Student: public Human
{
public:
   Student()
   {
   }
   -Student()
  {
      cout << "Student::-Student()";
  }
};

void main()
{
  Human *H = new Student();
  delete H;
}
```

Solution: Here we cannot write Human *H = new student(); in the main() function because base class's member includes in derived class's object so we should write this as follow
student *H = new Student();

9.4: **Correct the errors in the following program.**

```
class  Human
{
private:
      int m;
public:
      void getdata()
      {
            cout << " Enter number:";
            cin >> m;
      }
};
main()
{
    test T;
    T->getdata();
    T->display();

    test *p;
    p = new test;
    p.getdata();
    (*p).display();
}
```

Solution: Here T->getdata replace with T.getdata and T->display replace with T.display in the main() function. Because in this program T is object to pointer.

9.5: **Debug and run the following program. What will be the output?**

```
#include<iostream.h>
class  A
{
protected:
      int a,b;
public:
      A(int x = 0, int y)
      {
            a = x;
            b = y;
      }
      virtual void print ();
};
class B: public A
{
     private:
        float p,q;
     public:
        B(intm, int n, float u, float v)
         {
              p = u;
              q = v;
         }
         B() {p = p = 0;}
         void input(float u, float v);
         virtual void print(loat);
};
void A::print(void)
{
     cout << A values: << a <<""<< b << "\n";
}
void B:print(float)
{
     cout << B values: << u <<""<< v << "\n";
}
void B::input(float x, float y)
{
     p = x;
     q = y;
}
main()
{
     A a1(10,20), *ptr;
     B b1;
     b1.input(7.5,3.142);

     ptr = &a1;
     ptr->print();

     ptr = &b1;
     ptr->print();
}
```

Programming Exercises

9.1: **Create a base class called shape. Use this class to store two double type values that could be used to compute the area of figures. Derive two specific classes called triangle and rectangle from the base shape. Add to the base class, a member function get_data() to initialize base class data members and another member function display_area() to compute and display the area of figures. Make display_area() as a virtual function and redefine this function in the derived classes to suit their requirements.**

Using these three classes, design a program that will accept dimensions of a triangle or a rectangle interactively, and display the area.

Remember the two values given as input will be treated as lengths of two sides in the case of rectangles and as base and height in the case of triangles, and used as follows:

Area of rectangle = x * y

Area of triangle = ½ * x * y

Solution:

```
#include<iostream.h>
#include<iomanip.h>
class shape
{
       public:
       double x,y;
       public:
        void get_data()
         {
             cin>>x>>y;

         }
       double get_x(){return x;}
                double get_y(){return y;}
       virtual void display_area(){}
};

class triangle:public shape
{
    public:
    void display_area()
           {
                   double a;
             a=(x*y)/2;
             cout<<" Area of triangle = "<<a<<endl;

       }
};
class rectangle:public shape
{
      public:
      void display_area()
       {
              double a;
              a=x*y;
              cout<<" Area of rectangle = "<<a<<endl;
      }
};
        int main()
{

       shape *s[2];
       triangle t;
       s[0]=&t;
              rectangle r;
      s[1]=&r;
             cout<<" Enter the value of x & y for triangle: ";
      s[0]->get_data();
      cout<<" Enter the value of x & y for rectangle: ";
      s[1]->get_data();
             s[0]->display_area();
      s[1]->display_area();
            return 0;
 }
```

output

```
Enter the value of x & y for triangle: 12   26

Enter the value of x & y for rectangle: 24    14

Area of triangle = 156

Area of rectangle = 336
```

9.2: **Extend the above program to display the area of circles. This requires addition of a new derived class 'circle' that computes the area of a circle. Remember, for a circle we need only one value, its radius, but the get_data() function in base class requires two values to be passed.(Hint: Make the second argument of get_data() function as a default one with zero value.)**

Solution:

```
#include<iostream.h>
#include<iomanip.h>
#define pi 3.1416
class shape
```

```
{
      public:
      double x,y;
      public:
       void get_data(double a,double b)
        {
               x=a;
               y=b;

                 }
       double get_x(){return x;}
       double get_y(){return y;}
      virtual void display_area(){}
};

class triangle:public shape
{
    public:
      void display_area()
       {
               double a;
               a=(x*y)/2;
              cout<<" Area of triangle = "<<a<<endl;

       }
};

       class rectangle:public shape
{
     public:
      void display_area()
        {
                double a;
                a=x*y;
                cout<<" Area of rectangle = "<<a<<endl;
        }
};
class circle:public shape
{
     public:
     void display_area()
      {
                    double a;
             a=pi*x*x;
             cout<<" Area of circle = "<<a<<endl;
       }
};

int main()
{

```

```
            shape *s[3];

        triangle t;
        s[0]=&t;

        rectangle r;
        s[1]=&r;

        circle c;
        s[2]=&c;
        double x,y;
         cout<<" Enter the value of x & y for triangle: ";
         cin>>x>>y;
         s[0]->get_data(x,y);
         cout<<" Enter the value of x & y for rectangle: ";
         cin>>x>>y;
         s[1]->get_data(x,y);
         cout<<" Enter the radius of circle : ";
         double rd;
         cin>>rd;
         s[2]->get_data(rd,0);
         cout<<endl<<endl;
         s[0]->display_area();
         s[1]->display_area();
         s[2]->display_area();

        return 0;
}
```

output

```
Enter the value of x & y for triangle: 10      24

Enter the value of x & y for rectangle: 14      23

Enter the radius of circle : 12



Area of triangle = 120

Area of rectangle = 322

Area of circle = 452.3904
```

9.3: **Run the program above with the following modifications:**
(a) Remove the definition of display_area() from one of the derived classes.

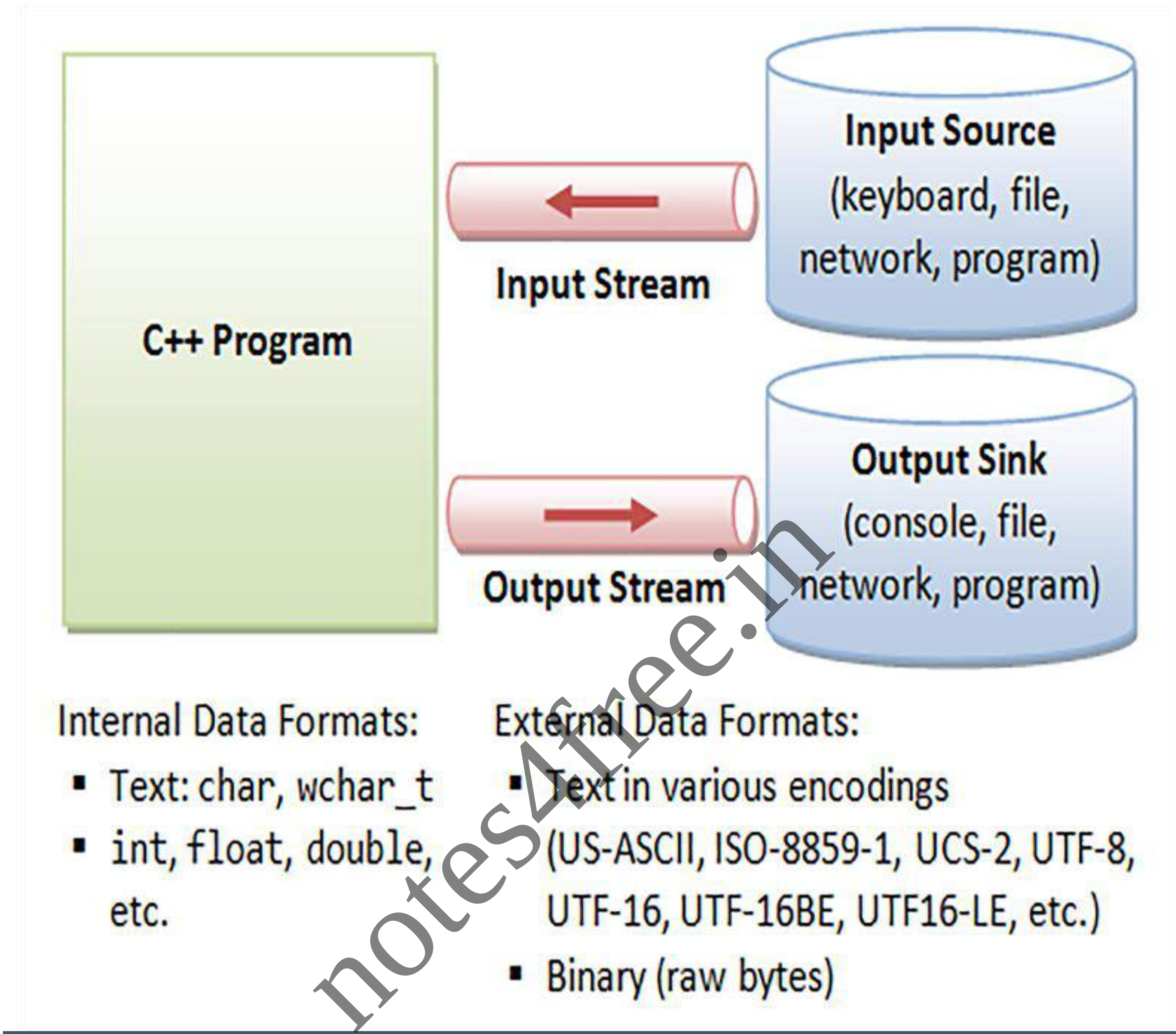
(b) In addition to the above change, declare the display_area() as virtual in the base class shape.
Comment on the output in each case.

Solution:

```
#include<iostream.h>
#include<iomanip.h>
#define pi 3.1416
class shape
{
  public:
  double x,y;
  public:
  void get_data(double a,double b)
 {
      x=a;
      y=b;

}
double get_x(){return x;}
double get_y(){return y;}
virtual void display_area(){}
};
class triangle:public shape
{
   public:
   void display_area()
  {
                           double a;
       a=(x*y)/2;
       cout<<" Area of triangle = "<<a<<endl;
 }
};
class rectangle:public shape
{
      public:
       void display_area()
        {
              double a;
              a=x*y;
             cout<<" Area of rectangle = "<<a<<endl;
        }
};
class circle:public shape
{
   public:
  void display_area()
   {
          double a;
        a=pi*x*x;
        cout<<" Area of circle = "<<a<<endl;
   }
};

int main()
{

   shape *s[3];
   triangle t;
  s[0]=&t;

 rectangle r;
 s[1]=&r;
circle c;
s[2]=&c;
double x,y;
cout<<" Enter the value of x & y for triangle: ";
cin>>x>>y;
s[0]->get_data(x,y);
cout<<" Enter the value of x & y for rectangle: ";
cin>>x>>y;
s[1]->get_data(x,y);
cout<<" Enter the radius of circle : ";
double rd;
cin>>rd;
s[2]->get_data(rd,0);
cout<<endl<<endl;
s[0]->display_area();
s[1]->display_area();
s[2]->display_area();

return 0;
}
```

output

```
Enter the value of x & y for triangle: 28 32

Enter the value of x & y for rectangle: 25 36

Enter the radius of circle : 20



Area of triangle = 448

Area of rectangle = 900

Area of circle = 1256.64
```


C++ Program
Input Stream
Input Source (keyboard, file, network, program)
Output Stream
Output Sink (console, file, network, program)
Internal Data Formats:
Text: char, wchar_t
int, float, double, etc.
External Data Formats:
Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
Binary (raw bytes)

MODULE -5

STREAMS AND WORKING WITH FILES

GANESH Y
Dept. of ECE RNSIT

MODULE -5
Streams and Working with files

Syllabus

C++ streams and stream classes, formatted and unformatted I/O operations, Output with manipulators, Classes for file stream operations, opening and closing a file, EOF (Selected topics from Chap-10, 11 of Text).

Introduction

Every program takes some data as input and generates processed data as output following the familiar input-process-output cycle. It is, therefore, essential to know how to provide the input data and how to present the results in a desired form.

We have, in the earlier chapters, used **cin** and **cout** with the operators >> and << for the input and output operations. But we have not so far discussed as to how to control the way the output is printed. C++ supports a rich set of I/O functions and operations to do this.

Since these functions use the advanced features of C++ (such as classes, derived classes and virtual functions), we need to know a lot about them before really implementing the C++ I/O operations.

Remember, C++ supports all of C's rich set of I/O functions. We can use any of them in the C++ programs. But we restrained from using them due to two reasons. First, I/O methods in C++ support the concepts of 00P and secondly, I/O methods in C cannot handle the user-defined data types such as class objects.

C++ uses the concept of ***stream*** and ***stream classes*** to implement its I/O operations with the console and disk files. We will discuss in this module, how stream classes support the console-oriented input-output operations and File-oriented I/O operations.

C++ Streams

The I/O system in C++ is designed to work with a wide variety of devices including ***terminals, disks, and tape drives.*** Although each device is very different, the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as ***stream.***

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.***

In other words, a program *extracts* the bytes from an input stream and *inserts* bytes into an output stream as illustrated in Fig. 10.1.

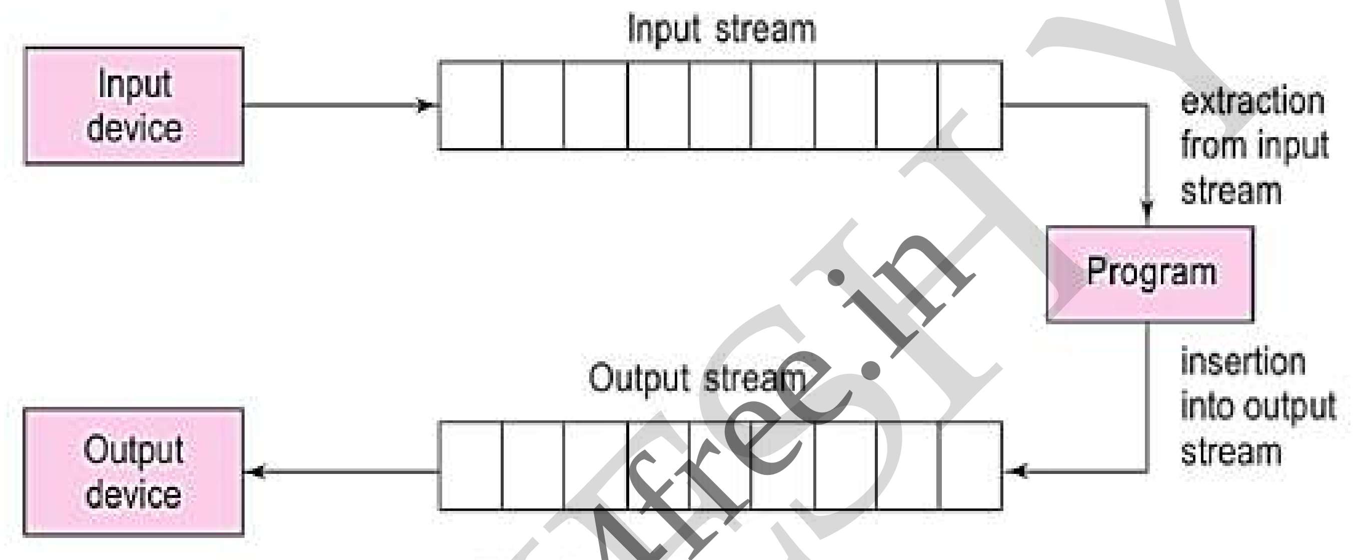


Fig. 10.1 *Data streams*

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.

Hence, a stream acts as an interface between the program and the input/output device. Therefore, a C++ program handles data (input or output) independent of the devices used.

C++ contains several pre-defined streams that are automatically opened when a program begins its execution. These include cin and cout, we know that cin represents the input stream connected to the standard input device (usually the keyboard) and cout represents the output stream connected to the standard output device (usually the screen).

Note that the keyboard and the screen are default options. We can redirect streams to other devices or files, if necessary.

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.***

Figure 10.2 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.

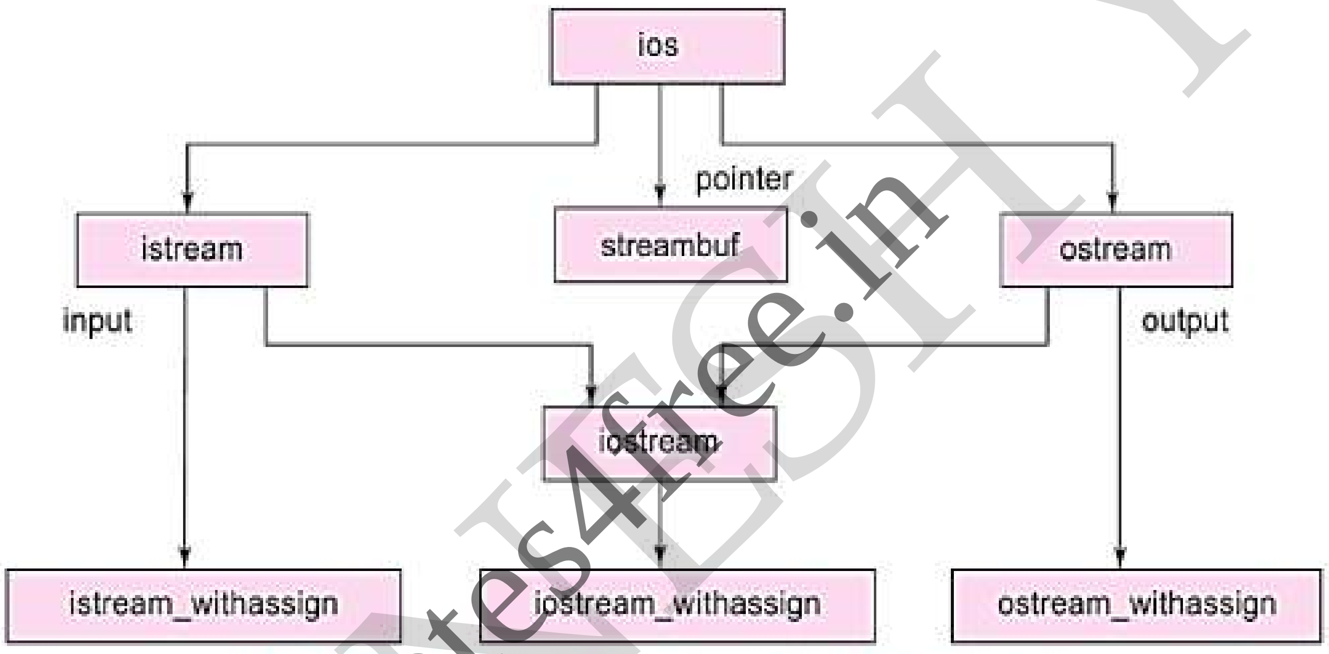


Fig. 10.2 Stream classes for console I/O operations

As seen in the Fig. 10.2, **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 <u>*virtual base class*</u> so that only one copy of its members are inherited by the **iostream.**

The class **ios** provides the basic support for formatted and unformatted I/O operations.

The class **istream** provides the facilities for formatted and unformatted input while the class **ostream** (through inheritance) provides the facilities for formatted output.

The class **iostream** provides the facilities for handling both input and output streams. Three classes, namely, **istream_withassign, ostream_withassign,** and **iostream_withassign** add assignment operators to these classes. Table 10.1 gives the details of these classes.

Table 10.1 *Stream classes for console operations*

Class name	*Contents*
ios (General input/output stream class)	• Contains basic facilities that are used by all other input and output classes • Also contains a pointer to a buffer object (**streambuf** object) • Declares constants and functions that are necessary for handling formatted input and output operations
istream (input stream)	• Inherits the properties of **ios** • Declares input functions such as **get(), getline()** and **read()** • Contains overloaded extraction operator >>
ostream (output stream)	• Inherits the properties of ios • Declares output functions **put()** and **write()** • Contains overloaded insertion operator <<
iostream (input/output stream)	• Inherits the properties of **ios istream** and **ostream** through multiple inheritance and thus contains all the input and output functions
streambuf	• Provides an interface to physical devices through buffers • Acts as a base for **filebuf** class used ios files

Unformatted I/O Operations

Overloaded Operators >> and <<

We have used the objects **cin** and **cout** (pre-defined in the *iostream* file) for the input and output of data of various types. This has been made possible by overloading the operators >> and << to recognize all the basic C++ types.

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
```

variable 1, variable 2, ... 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 data2 ..... dataN
```

The input data are separated by *white spaces* and should match the type of variable in the cin list. Spaces, newlines and tabs will be skipped.

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 example, consider the following code:

```
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.

Note: Whitespace is a term that refers to characters that are used for formatting purposes. In C++, this refers primarily to spaces, tabs, and (sometimes) newlines.

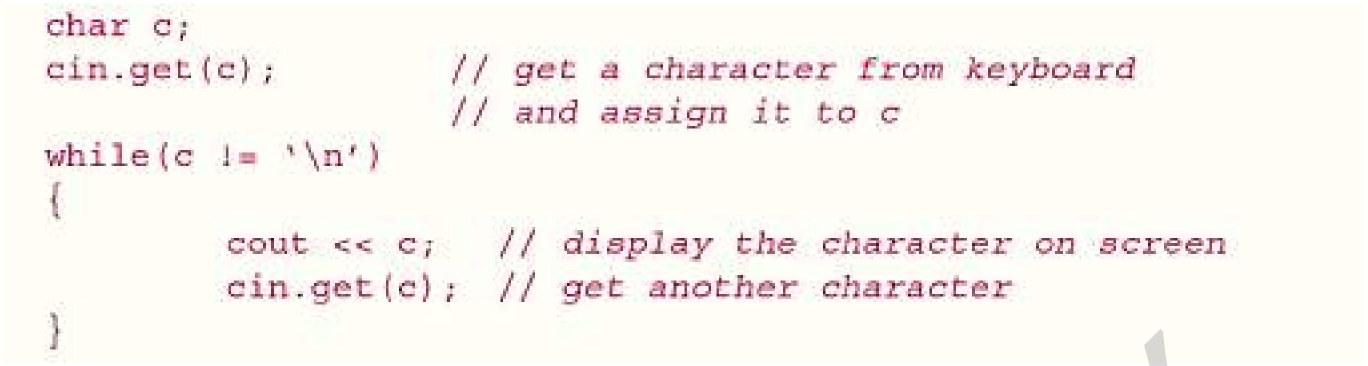
put( ) and get( ) Functions

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(void)** 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(void)** version returns the input character.

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



```
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). Remember, the operator >> can also be used to read a character but it will *skip* the white spaces and newline character. The above **while** loop will not work properly if the statement

`cin >> c;`

is used in place of

`cin.get(c);`

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

```
………..
char c;
c= cin.get(); // cin.get(c); replaced
………..
………..
```

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.**

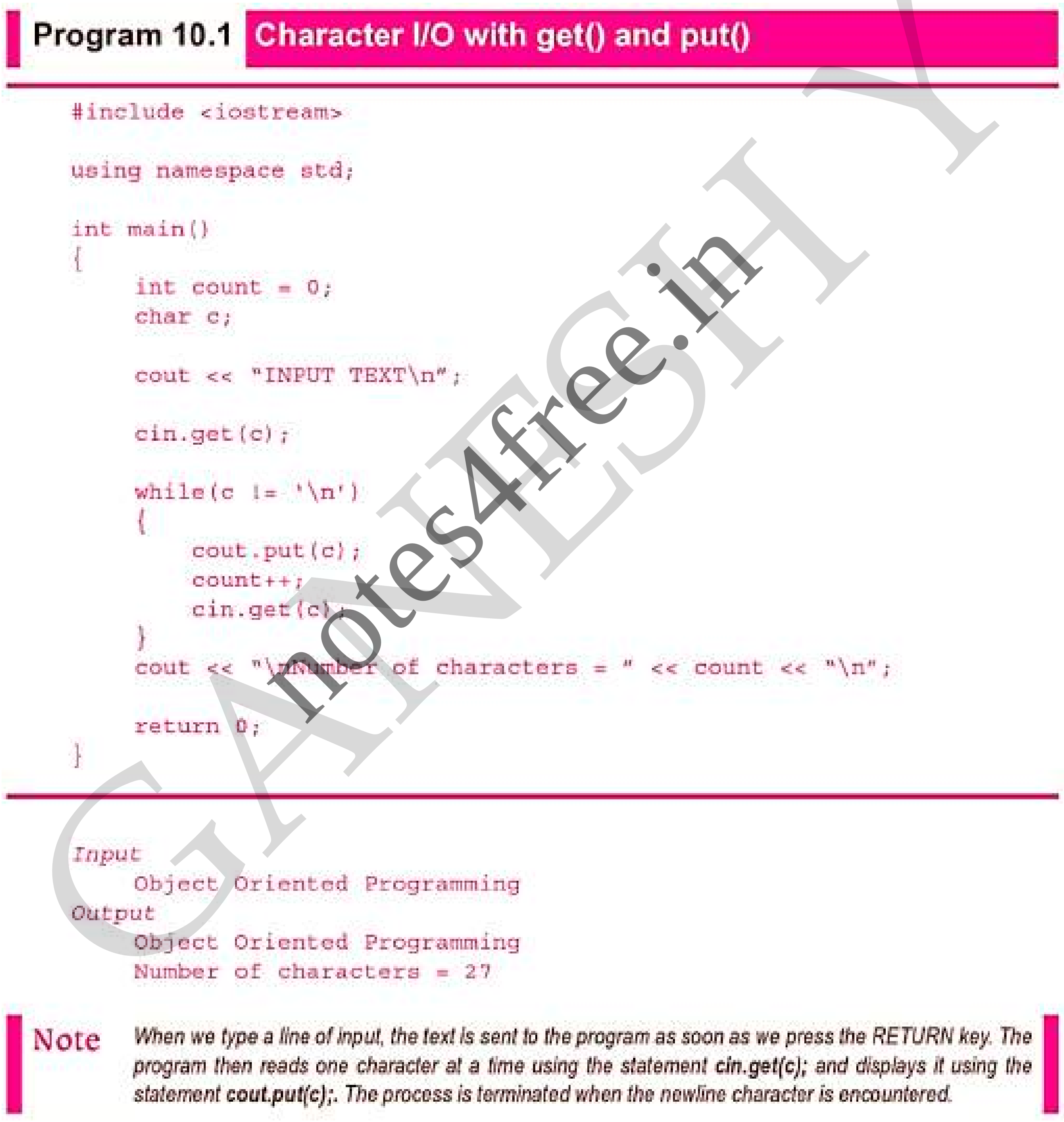
The variable **ch** must contain a character value. We can also use a number as an argument to the function **put().** For example,

`cout.put(68);`

displays the character D. This statement will convert the **int** value 68 to a **char** value and display the character whose ASCII value is 68.

The following segment of a program reads a line of text from the keyboard and displays it on the screen.

```
        char c;
        cin.get(c); // read a character
        while (c !='\n')
        {
             cout.put(c); // display the character on screen
             cin.get(c);
        }
```



Program 10.1 **Character I/O with get() and put()**

```
#include <iostream>

using namespace std;

int main()
{
     int count = 0;
     char c;

     cout << "INPUT TEXT\n";

     cin.get(c);

     while(c != '\n')
     {
          cout.put(c);
          count++;
          cin.get(c);
     }
     cout << "\nNumber of characters = " << count << "\n";

     return 0;
}
```

```
Input
     Object Oriented Programming
Output
     Object Oriented Programming
     Number of characters = 27
```

Note *When we type a line of input, the text is sent to the program as soon as we press the RETURN key. The program then reads one character at a time using the statement* ***cin.get(c);*** *and displays it using the statement* ***cout.put(c);****. The process is terminated when the newline character is encountered.*

getline() and write() Functions

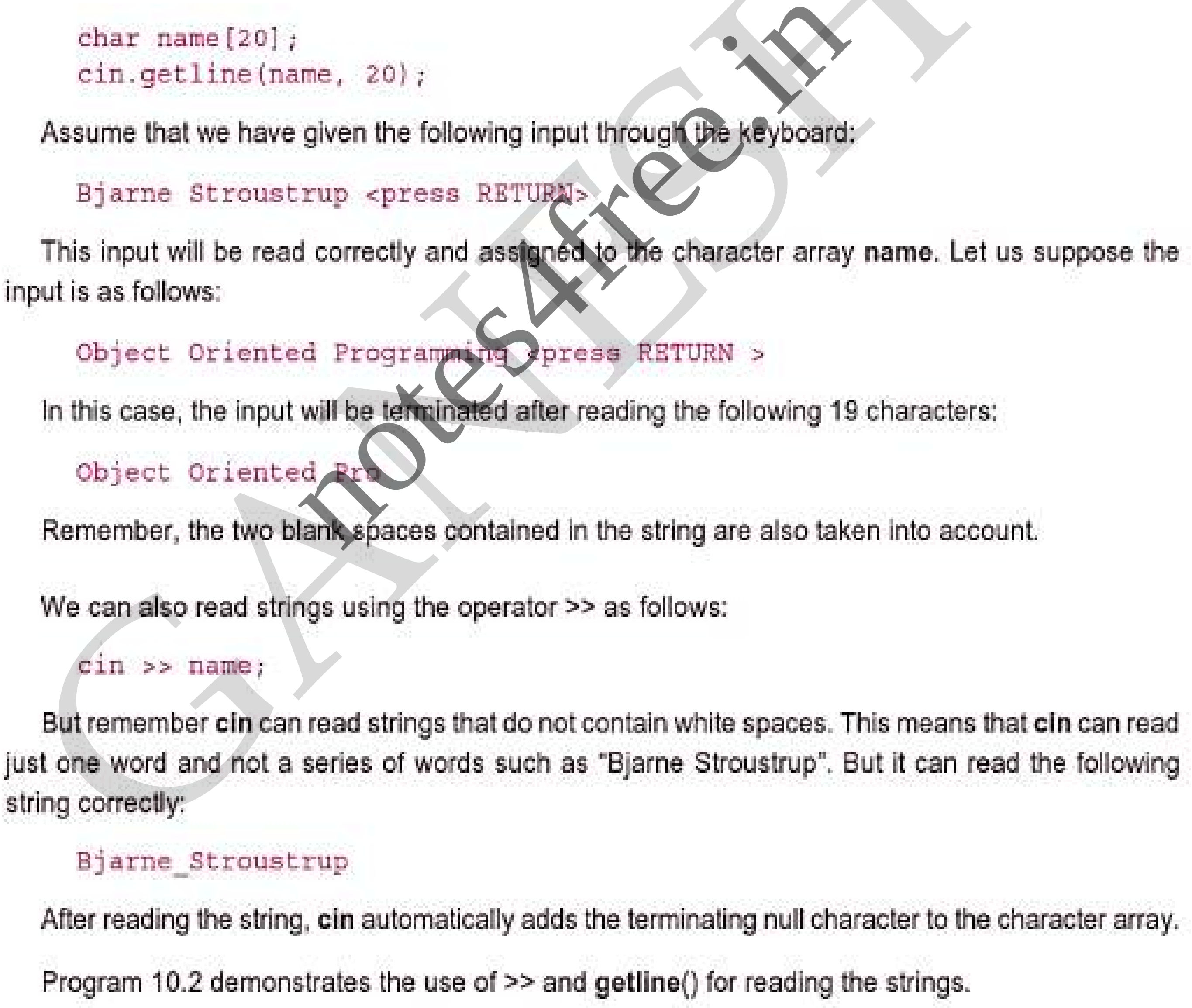
We can read and display a line of text more efficiently using the line-oriented input/output functions **getline()** and **write().** The **getline()** function reads a whole line of text that ends with a newline character (transmitted by the RETURN key).

This function can be invoked by using the object **cin** 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 (whichever occurs first).

The newline character is read but not saved. Instead, it is replaced by the null character. For example, 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
```

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

We can also read strings using the operator >> as follows:

```
cin >> name;
```

But remember **cin** can read strings that do not contain white spaces. This means that **cin** can read just one word and not a series of words such as "Bjarne Stroustrup". But it can read the following string correctly:

```
Bjarne_Stroustrup
```

After reading the string, **cin** automatically adds the terminating null character to the character array.

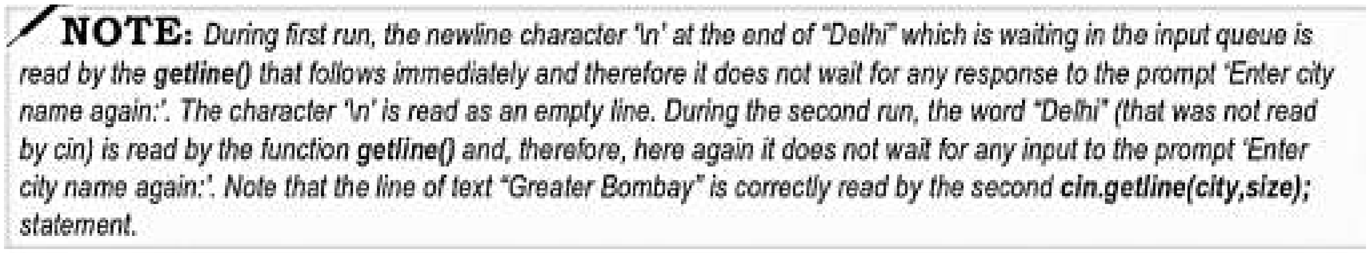
Program 10.2 demonstrates the use of >> and **getline()** for reading the strings.

Program 10.2 Reading Strings with getline()

```
#include <iostream>

using namespace std;

int main()
{
     int size = 20;
     char city[20];

     cout << "Enter city name: \n";
     cin >> city;
     cout << "City name:" << city << "\n\n";

     cout << "Enter city name again: \n";
     cin.getline(city, size);
     cout << "City name now: " << city << "\n\n";

     cout << "Enter another city name: \n";
     cin.getline(city, size);
     cout << "New city name: " << city << "\n\n";

     return 0;
}
```

The output of Program 10.2 would be:

```
First run
     Enter city name:
     Delhi
     City name: Delhi

     Enter city name again:
     City name now:
     Enter another city name:
     Chennai
     New city name: Chennai


Second run
     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
```



NOTE: *During first run, the newline character '\n' at the end of "Delhi" which is waiting in the input queue is read by the* ***getline()*** *that follows immediately and therefore it does not wait for any response to the prompt 'Enter city name again:'. The character '\n' is read as an empty line. During the second run, the word "Delhi" (that was not read by cin) is read by the function* ***getline()*** *and, therefore, here again it does not wait for any input to the prompt 'Enter city name again:'. Note that the line of text "Greater Bombay" is correctly read by the second* ***cin.getline(city,size);*** *statement.*

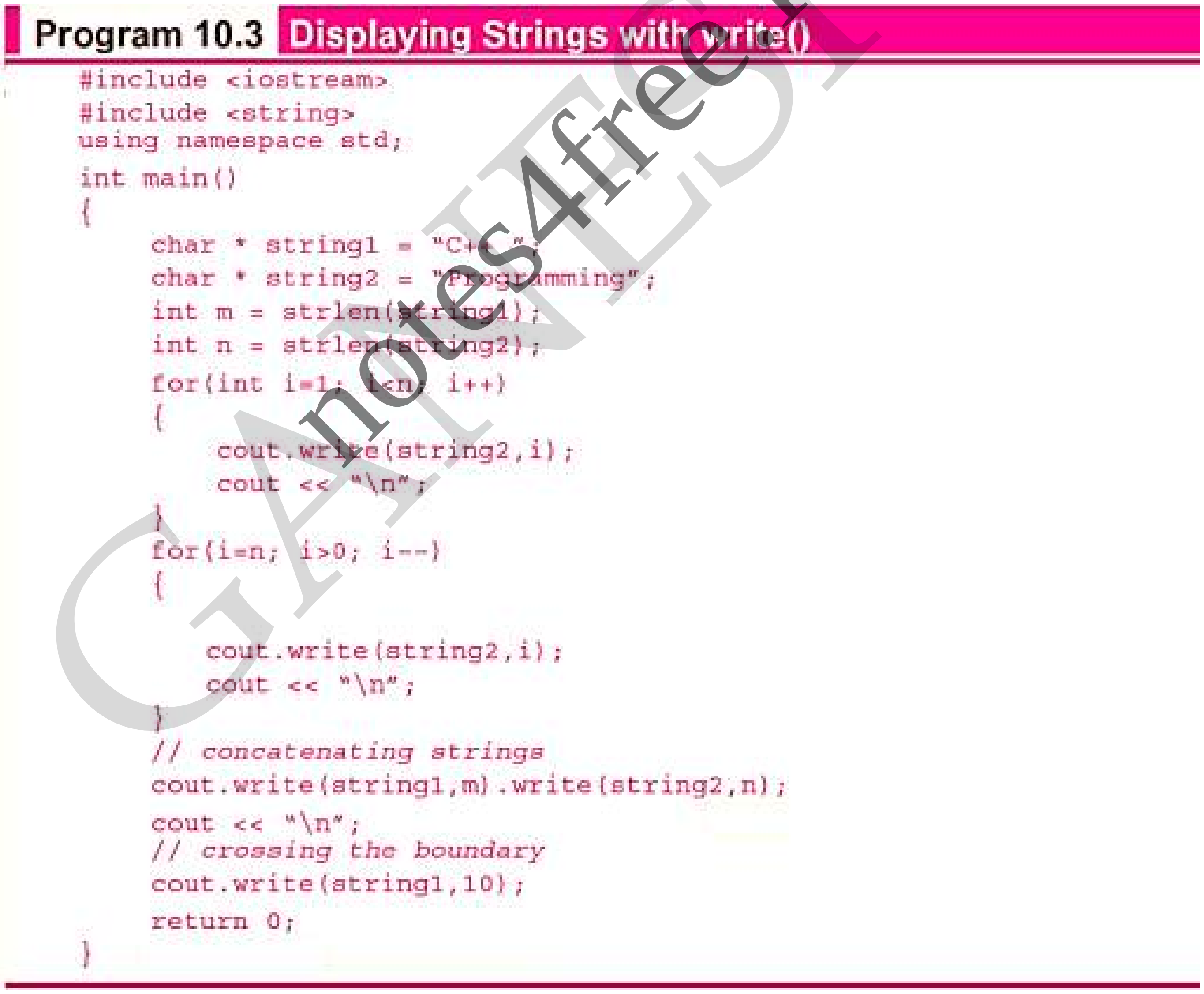
The **write()** function displays an entire line and has the following form:

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.

Note that it *does not stop displaying* the characters automatically when the null character is encountered. If the size is greater than the length of line, then it displays beyond the bounds of line.

Program 10.3 illustrates how **write()** method displays a string.



Program 10.3 Displaying Strings with write()

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    char * string1 = "C++ ";
    char * string2 = "Programming";
    int m = strlen(string1);
    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";
    // crossing the boundary
    cout.write(string1,10);
    return 0;
}
```

The output of Program 10.3 would be:

```
P
Pr
Pro
Prog
Progr
Progra
Program
Programm
Programmi
Programmin
Programming
Programmin
Programmi
Programm
Program
Progra
Progr
Prog
Pro
Pr
P
C++ Programming
C++ Progr
```

The last line of the output indicates that the statement

```
cout.write(string1, 10);
```

displays more characters than what is contained in **string1**.

It is possible to concatenate two strings using the **write()** function. The statement

```
cout.write(string1, m).write(string2, n);
```

is equivalent to the following two statements:

```
cout.write(string1, m);
cout.write(string2, n);
```

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. The most important ones among them are listed in Table 10.2.

Table 10.2 *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 10.3 shows some important manipulator functions that are frequently used. To access these manipulators, the file iomanip should be included in the program.

Table 10.3 *Manipulators*

Manipulators	*Equivalent ios function*
setw()	**width()**
setprecision()	**precision()**
setfill()	**fill()**
setiosflags()	**setf()**
resetiosflags()	**unsetf()**

In addition to these functions supported by the C++ library, we can create our own manipulator functions to provide any special output formats. The following sections will provide details of how to use the pre-defined formatting functions and how to create new ones.

Defining Field Width: width()

We can use the **width()** function to define the width of a field necessary for the output of an item. Since, it is a member function, we have to use an object 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 (the item that follows immediately). After printing one item (as per the specifications) it will revert back to the default. For example, the statements

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

will produce the following output:

```
  54312
```

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. This can be improved as follows:

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

This produces the following output:

```
  543   12
```

Remember that the field width should be specified for each item separately. 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. Program 10.4 demonstrates how the function **width()** works.

Program 10.4 **Specifying Field Size with width()**

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

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;
    }
```

The output of Program 10.4 would be:

```
    ITEMS       COST  TOTAL VALUE

       10         75          750
        8        100          800
       12         60          720
       15         99         1485

   Grand Total = 3755
```

NOTE: *A field of width two has been used for printing the value of sum and the result is not truncated. A good gesture of C++ !*

Setting Precision: precision()

By default, the floating numbers are printed with six digits after the decimal point. However, 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:

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

Not that, unlike the function **width(), precision()** retains the setting in effect until it is reset. That is why we have declared only one statement for the precision setting which is used by all the three outputs.

We can set different values to different precision as follows:

```
cout.precision(3);
cout << sqrt(2) << "\n";
cout.precision(5);                    // Reset the precision
cout << 3.14159 << "\n";
```

We can also combine the field specification with the precision setting. 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

Program 10.5 shows how the functions **width()** and **precision()** are jointly used to control the output format.

Program 10.5 Precision Setting with precision()

```
#include <iostream>
#include <cmath>

using namespace std;

int main()
{
        cout << "Precision set to 3 digits \n\n";
        cout.precision(3);

        cout.width(10);
        cout << "VALUE";
        cout.width(15);
        cout << "SQRT_OF_VALUE" << "\n";
```

```
            for(int n=1; n<=5; n++)
            {

               cout.width(8);
               cout << n;
               cout.width(13);
               cout << sqrt(n) << "\n";
            }
            cout << "\n Precision set to 5 digits \n\n";
            cout.precision(5);             // precision parameter changed
            cout << " sqrt(10) = " << sqrt(10) << "\n\n";

            cout.precision(0);             // precision set to default
            cout << " sqrt(10) = " << sqrt(10) << " (default
                                     setting)\n";
            return 0;
    }
```

The output of Program 10.5 would be:

```
  Precision set to 3 digits
       VALUE       SQRT_OF_VALUE
         1               1
         2            1.41
         3            1.73
         4               2
         5            2.24

  Precision set to 5 digits
  sqrt(10) = 3.1623
  sqrt(10) = 3.162278 (default setting)
```

Note *Observe the following from the output:*

1. *The output is rounded to the nearest cent (i.e., 1.6666 will be 1.67 for two digit precision but 1.3333 will be 1.33).*
2. *Trailing zeros are truncated.*
3. *Precision setting stays in effect until it is reset.*
4. *Default precision is 6 digits.*

Program 10.6 shows another program demonstrating the functionality of width and precision manipulators:

Program 10.6 Width and Precision Manipulators

```
#include <iostream>
void main()
{     float pi=22.0/7.0;
      int i;
      cout<<"Value of PI:\n";
      for(i=1;i<=10;i++)
      {
         cout.width(i+1);
         cout.precision(i);
         cout<<pi<<"\n";
      }
}
```

The output of Program 10.6 would be:

```
Value of PI:

3.1
3.14
3.143
3.1429
3.14286
3.142857
3.1428571
3.14285707
3.142857075
3.1428570747
```

Filling and Padding: fill()

We have been printing the values using much larger field widths than required by the values. The unused positions of the field are filled with white spaces, by default. However, 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. Like **precision()**, **fill()** stays in effect till we change it. See Program 10.7 and its output.

Program 10.7 Padding with fill()

```
#include <iostream>

using namespace std;

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 << 12.345678 << "\n";

return 0;
}
```

The output of Program 10.7 would be:

```
<<<<1<<<<<<<<<1
<<<<2<<<<<<<0.5
<<<<3<<<<<0.333
>>>>4>>>>>>0.25
>>>>5>>>>>>>0.2
>>>>6>>>>>0.167

Padding changed

#########12.346
```

Formatting Flags, Bit-fields and setf()

We have seen that when the function **width()** is used, the value (whether text or number) is printed right-justified in the field width created. But, it is a usual practice to print the text left-justified. How do we get a value printed left-justified? Or, how do we get a floating-point number printed in the scientific notation?

The **setf()**, a member function of the **ios** class, can provide answers to these and many other formatting questions. The **setf()** (*setf* stands for 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.

Table 10.4 shows the bit fields, flags and their format actions. There are three bit fields and each has a group of format flags which are mutually exclusive. Examples:

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

Table 10.4 *Flags and bit fields for setf() function*

Format required	*Flag (arg1)*	*Bit-field (arg2)*
Left-justified output	ios :: left	ios :: adjustfield
Right-justified output	ios :: right	ios :: adjustfield
Padding after sign or base Indicator (like +##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

Note that the first argument should be one of the group members of the second argument.

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:

```
TABLE 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:

```
-*****1.235e+01
```

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

Program 10.8 demonstrates the manipulation of flag and bit fields for setf() function:

Program 10.8 Manipulation of Flag and Bit Fields

```
#include <iostream>
using namespace std;
void main()
{
    int num;
    cout<<"Enter an integer value: ";
    cin>>num;

    cout<<"The hexadecimal, octal and decimal representation of
    "<<num<<" is: ";

    cout.setf(ios::hex, ios::basefield);
    cout<<num<<", ";

    cout.setf(ios::oct, ios::basefield);
    cout<<num<<" and ";

    cout.setf(ios::dec, ios::basefield);
    cout<<num<<" respectively";
}
```

The output of Program 10.8 would be:

```
Enter an integer value: 92

The hexadecimal, octal and decimal representation of 92 is: 5c, 134
and 92 respectively
```

Displaying Trailing Zeros and Plus Sign

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

			1	0	.	7	5
						2	5
				1	5	.	5

Note that 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. Under default precision, the value 3.25 will be displayed as 3.250000. Remember, the default precision assumes a precision of six digits.

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

```
cout.setf(ios::showpos); // show +sign
```

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 bit fields and therefore are used as single arguments in **setf()**. This is possible because the **setf()** has been declared as an overloaded function in the class **ios**. Table 10.5 lists the flags that do not possess a named bit field. These flags are not mutually exclusive and therefore can be set or cleared independently.

Table 10.5 *Flags that do not have bit fields*

Flag	*Meaning*
ios :: showbase	Use base indicator on output
ios :: showpos	Print + before positive numbers
ios :: showpoint	Show trailing decimal point and zeroes
ios :: uppercase	Use uppercase letters for hex output
ios :: skipus	Skip white space on input
ios :: unitbuf	Flush all streams after insertion
ios :: stdio	Flush **stdout** and **stderr** after insertion

Program 10.9 demonstrates the setting of various formatting flags using the overloaded **setf()** function.

Program 10.9 Formatting with Flags in setf()

```
#include <iostream>
#include <cmath>

using namespace std;

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";
            }

            // floatfield changed
                cout.setf(ios::scientific, ios::floatfield);
                cout << "\nSQRT(100) = " << sqrt(100) << "\n";

                return 0;
}
```

The output of Program 10.9 would be:

```
VALUE*********SQRT OF VALUE
+...1................+1.0000
+...2................+1.4142
+...3................+1.7321
+...4................+2.0000
+...5................+2.2361
+...6................+2.4495
+...7................+2.6458
+...8................+2.8284
+...9................+3.0000
+..10................+3.1623

SQRT(100) = +1.0000e+001
```

NOTE:

1. *The flags set by **setf()** remain effective until they are reset or unset.*
2. *A format flag can be reset any number of times in a program.*
3. *We can apply more than one format controls jointly on an output value.*
4. *The setf() sets the specified flags and leaves others unchanged.*

10.6 Managing Output with Manipulators

The header file *iomanip* provides a set of functions called *manipulators* which can be used to manipulate the output formats. They provide the same features as that of the **ios** member functions and flags. Some manipulators are more convenient to use than their counterparts in the class **ios**. For example, two or more manipulators can be used as a chain in one statement as shown below:

```
cout << manip1 << manip2 << manip3 << item;
cout << manip1 << item1 << manip2 << item2;
```

This kind of concatenation is useful when we want to display several columns of output.

The most commonly used manipulators are shown in Table 10.6. The table also gives their meaning and equivalents. To access these manipulators, we must include the file *iomanip* in the program.

Table 10.6 *Manipulators and their meanings*

Manipulator	*Meaning*	*Equivalent*
setw (int *w*)	Set the field width to w.	width( )
setprecision(int *d*)	Set the floating point precision to *d*.	precision( )
setfill(int *c*)	Set the fill character to c.	fill( )
setiosflags(long *f*)	Set the format flag *f*.	setf( )
resetiosflags(long *f*)	Clear the flag specified by *f*.	unsetf( )
endl	Insert new line and flush stream.	"\n"

Some examples of manipulators are given below:

```
cout << setw(10) << 12345;
```

This statement prints the value 12345 right-justified in a field width of 10 characters. The output can be made left-justified by modifying the statement as follows:

```
cout << setw(10) << setiosflags(ios::left) << 12345;
```

One statement can be used to format output for two or more values. For example, the statement

```
cout    << setw(5)  << setprecision(2) << 1.2345
        << setw(10) << setprecision(4) << sqrt(2)
        << setw(15) << setiosflags(ios::scientific) << sqrt(3);
        << endl;
```

will print all the three values in one line with the field sizes of 5, 10, and 15 respectively. Note that each output is controlled by different sets of format specifications.

We can jointly use the manipulators and the **ios** functions in a program. The following segment of code is valid:

```
cout.setf(ios::showpoint);
cout.setf(ios::showpos);
cout << setprecision(4);
cout << setiosflags(ios::scientific);
cout << setw(10) << 123.45678;
```

NOTE: *There is a major difference in the way the manipulators are implemented as compared to the* ***ios*** *member functions. The* ***ios*** *member function return the previous format state which can be used later, if necessary. But the manipulator does not return the previous format state. In case, we need to save the old format states, we must use the* ***ios*** *member functions rather than the manipulators. Example:.*

```
cout.precision(2);                  // previous state
int p = cout.precision(4);          // current state;
```

When these statements are executed, **p** will hold the value of 2 (previous state) and the new format state will be 4. We can restore the previous format state as follows:

```
cout.precision(p);                  // p = 2
```

Program 10.10 illustrates the formatting of the output values using both manipulators and **ios** functions.

Program 10.10 Formatting with Manipulators

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

using namespace std;

int main()
{
        cout.setf(ios::showpoint);

        cout << setw(5) << "n"
             << setw(15) << "Inverse_of_n"
             << setw(15) << "Sum_of_terms\n\n";

        double term, sum = 0;

        for(int n=1; n<=10; n++)
        {
             term = 1.0 / float(n);
             sum = sum + term;

             cout << setw(5) << n
               << setw(14) << setprecision(4)
               << setiosflags(ios::scientific) << term
               << setw(13) << resetiosflags(ios::scientific)
```

```
                    << sum << endl;

          }
          return 0;
  }
```

The output of Program 10.10 would be:

```
n    Inverse_of_n  Sum_of_terms

1    1.0000e+000   1.0000
2    5.0000e-001   1.5000
3    3.3333e-001   1.8333
4    2.5000e-001   2.0833
5    2.0000e-001   2.2833
6    1.6667e-001   2.4500
7    1.4286e-001   2.5929
8    1.2500e-001   2.7179
9    1.1111e-001   2.8290
10   1.0000e-001   2.9290
```

Designing Our Own Manipulators

We can design our own manipulators for certain special purposes. The general form for creating a manipulator without any arguments is:

```
ostream & manipulator (ostream & output)
{
     .....
     ..... (code)
     .....
     return output;
}
```

Here, the *manipulator* is the name of the manipulator under creation. The following function defines a manipulator called **unit** that displays "inches":

```
ostream & unit(ostream & output)
{
          output << " inches";
          return output;
}
```

The statement

```
cout << 36 << unit;
```

will produce the following output

```
36 inches
```

We can also create manipulators that could represent a sequence of operations. Example:

```
ostream & show(ostream & output)
{
      output.setf(ios::showpoint);
      output.setf(ios::showpos);
      output<< setw(10);
      return output;
}
```

This function defines a manipulator called **show** that turns on the flags **showpoint** and **showpos** declared in the class **ios** and sets the field width to 10.

Program 10.11 illustrates the creation and use of the user-defined manipulators. The program creates two manipulators called **currency** and **form** which are used in the **main** program.

Program 10.11 **User-Defined Manipulators**

```
#include <iostream>
#include <iomanip>
using namespace std;
// user-defined manipulators
ostream & currency(ostream & output)
{
      output << "Rs";
      return output;
}
ostream & form(ostream & output)
{
      output.setf(ios::showpos);
      output. setf ( ios: ·: showpoint) ;
      output .fill ( '*');
      output.precision(2);
      output<<setiosflags(ios::£ixed)
            << setw(10);
      return output;
}
int main()
{
      cout <<currency<< form<< 7864.5;
      return 0;
}
```

The output of Program 10.11 would be:

```
Rs**+7864.50
```

Note that **form** represents a complex set of format functions and manipulators.

Working with files

Many real-life problems handle large volumes of data and, in such situations, we need to use some devices such as floppy disk or hard disk to store the data. The data is stored in these devices 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.

This is illustrated in Fig. 11.1.

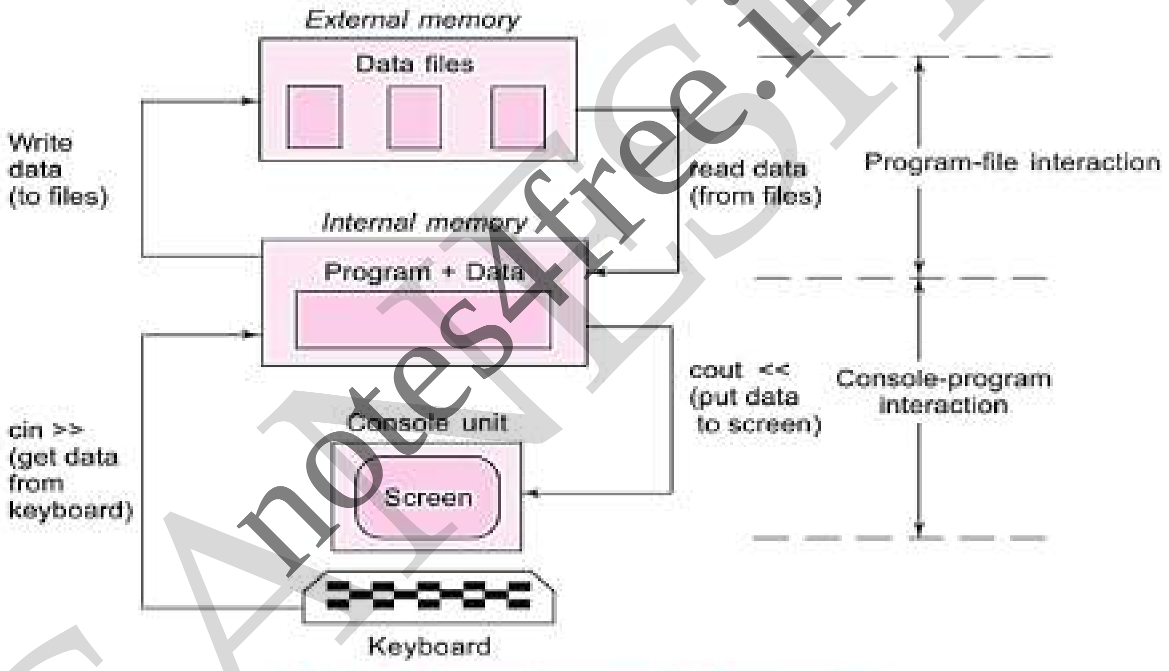


Fig. 11.1 *Consol-program-file interaction*

In this section we will discuss various methods available for storing and retrieving the data from files.

The I/0 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.* In other words, the input stream extracts (or reads) data from the file and the output stream inserts (or writes) data to the file. This is illustrated in Fig. 11.2.

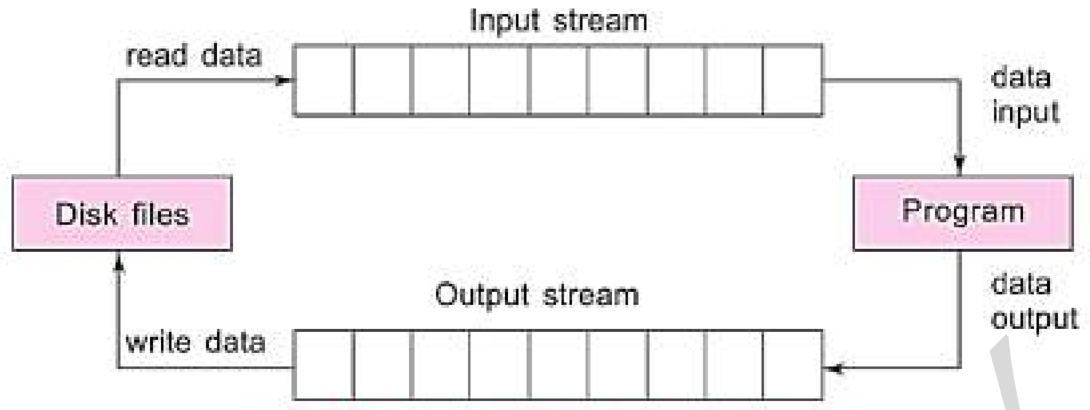


Fig. 11.2 *File input and output streams*

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 system 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 *iostream* class as shown in Fig. 11.3.

These classes, designed to manage the disk files, are declared in *fstream* and therefore, we must include this file in any program that uses files. Table 11.1 shows the details of file stream classes. Note that these classes contain many more features.

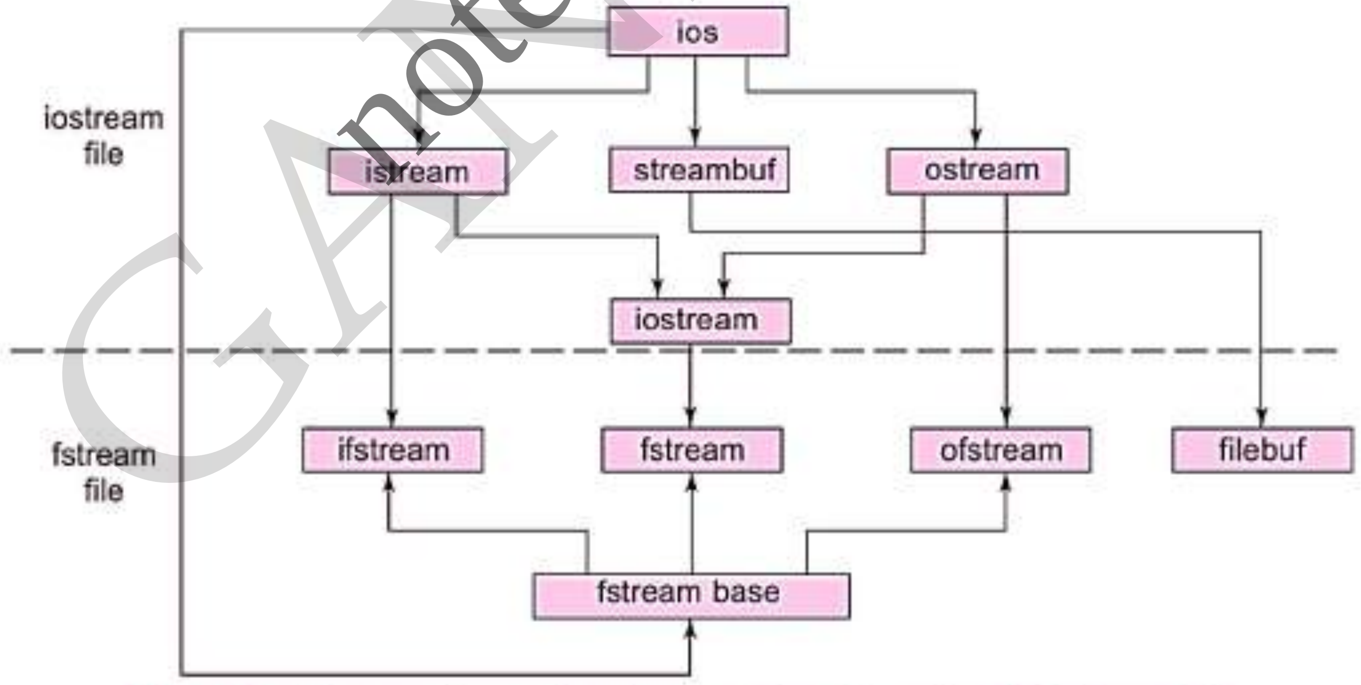
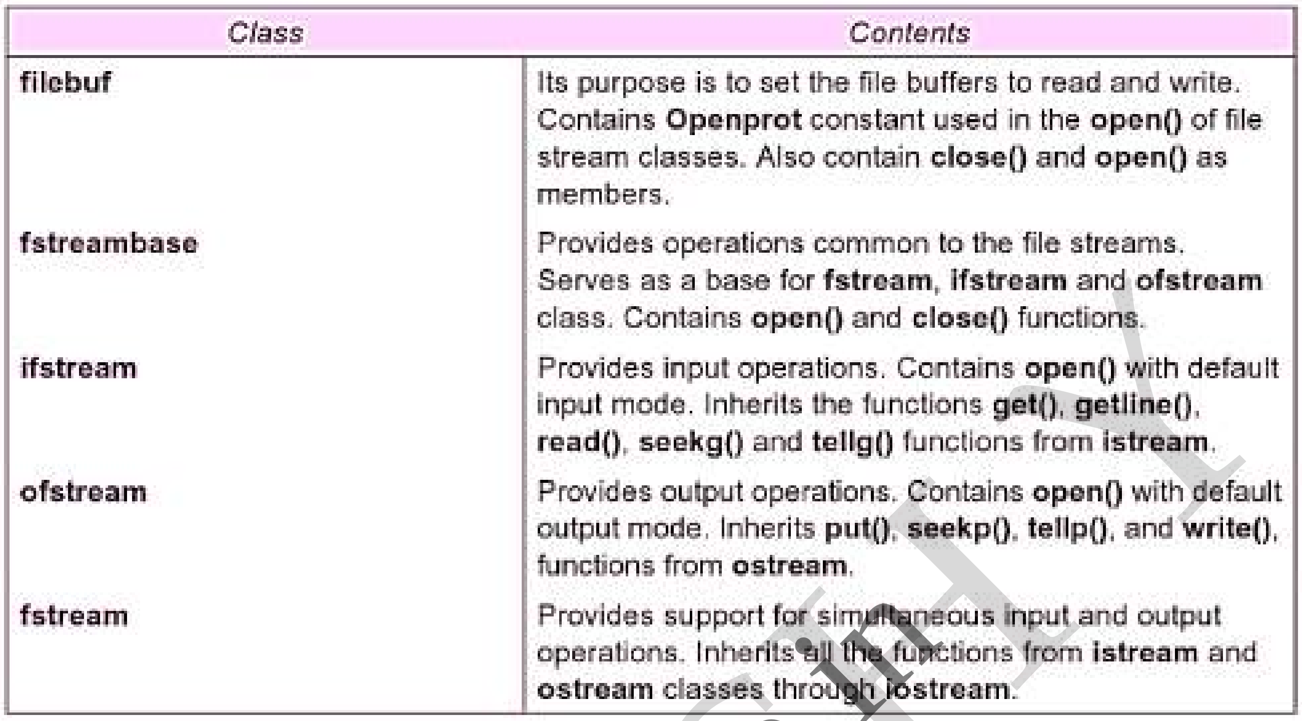


Fig. 11.3 *Stream classes for file operations (contained in fstream file)*

Table 11.1 Details of file stream classes



Class	Contents
filebuf	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes. Also contain **close()** and **open()** as members.
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 **put(), seekp(), tellp(),** and **write(),** functions from **ostream**.
fstream	Provides support for simultaneous input and output operations. Inherits all the functions from **istream** and **ostream** classes through **iostream**.

OPENING AND CLOSING A FILE

If we want to use a disk file, we need to decide the following things about the file and its intended use:

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

The filename 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. Examples:

Input.data
Test.doc
INVENT.ORY
student
salary
OUTPUT

As stated earlier, 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

We know that 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 to say, 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. 11.4.

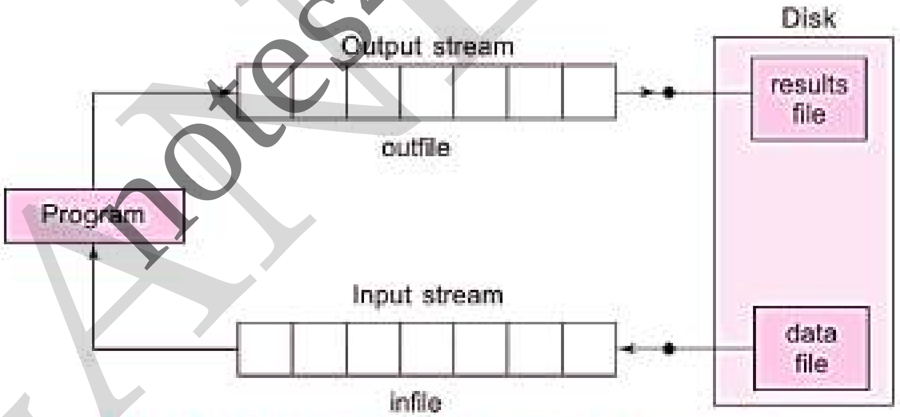


Fig. 11.4 *Two file streams working on separate files*

Similarly, the following statement declares **infile** as an **ifstream** object and attaches it to the file **data** for reading (input).

```
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. 11.5. The programs would contain the following statements:

```
Program1
        .....
        .....
        ofstream outfile("salary");      // creates outfile and
                                            connects
                                         // "salary" to it
        .....
        .....
Program2
        .....
        .....
        ifstream infile("salary");     // creates infile and connects
                                       // "salary" to it
        .....
        .....
```

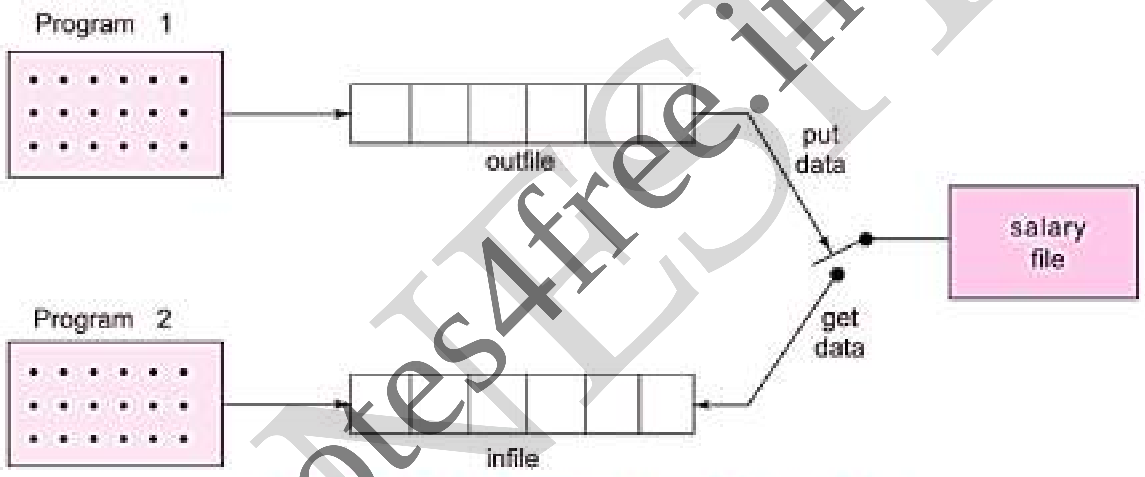


Fig. 11.5 *Two file streams working on one file*

The connection with a file is closed automatically when the stream object expires (when the program terminates). In the above statement, when the *program1* 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.

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

Although we have used a single program, we created two file stream objects, **outfile** (to put data into the file) and **infile** (to get data from the file). Note that the use of a statement like

```
outfile.close();
```

disconnects the file salary from the output stream **outfile**. Remember, the object **outfile** still exists and the **salary** file may again be connected to **outfile** later or to any other stream. In this example, we are connecting the **salary** file to **infile** stream to read data.

Program 11.1 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.

Program 11.1 Working with Single File

```
// 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 key board and

        outf << name << "\n";       // write to file ITEM

        cout << "Enter item cost:";
        float cost;
        cin >> cost;                // get cost from key board and

        outf << cost << "\n";       // write to file ITEM

        outf.close();               // Disconnect ITEM file from outf

        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;
}
```

The output of Program 11.1 would be:

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

Item name:CD-ROM
Item cost:250
```

CAUTION: *When a file is opened for writing only, a new file is created if there is no file of that name. If a file by that name exists already, then its contents are deleted and the file is presented as a clean file. We shall discuss later how to open an existing file for updating it without losing its original contents.*

Opening Files Using open()

As stated earlier, 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. This is done as follows:

```
file-stream-class stream-object;
stream-object.open ("filename");
```

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 previous program segment opens two files in sequence for writing the data. Note that the first file is closed before opening the second one. This is necessary because a stream can be connected to only one file at a time. See Program 11.2 and Fig. 11.6.

Program 11.2 Working with Multiple Files

```
// 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
     char line[N];

     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;
    }
```

The output of Program 11.2 would be:

```
Contents of country file
United States of America
United Kingdom
South Korea

Contents of capital file
Washington
London
Seoul
```

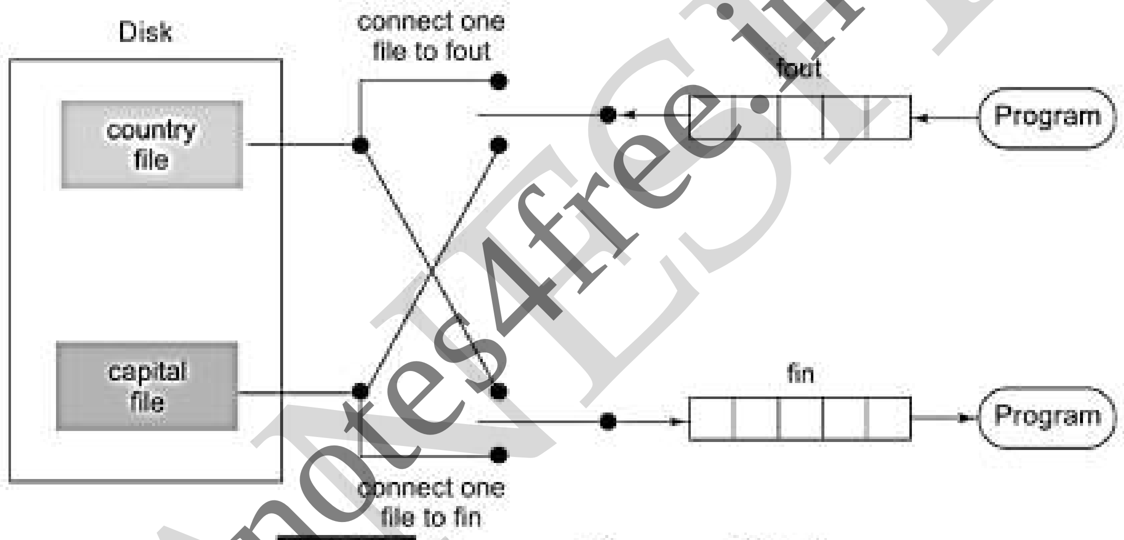


Fig. 11.6 *Streams working on multiple files*

At times we may require to use two or more files simultaneously. For example, we may require to merge two sorted files into a third sorted file. This means, both the sorted files have to be kept open for reading and the third one kept open for writing. In such cases, we need to create two separate input streams for handling the two input files and one output stream for handling the output file. See Program 11.3.

Program 11.3 Reading from Two Files Simultaneously

```
// Reads the files created in Program 11.2

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>                     // for exit() function

int main()
```

```
    {
            const int SIZE = 80;
            char line[SIZE];

            ifstream fin1, fin2;            // create two input streams
            fin1.open("country");
            fin2.open("capital");

            for(int i=1; i<=10; i++)
            {
               if(fin1.eof() != 0)
               {
                 cout << "Exit from country \n";
                 exit(1);
               }

               fin1.getline(line, SIZE);
               cout << "Capital of "<< line ;

               if(fin2.eof() != 0)
               {
                 cout << "Exit from capital\n";
                 exit(1);
               }

               fin2.getline(line,SIZE);
               cout << line << "\n";
            }
        return 0;
    }
```

The output of Program 11.3 would be:

```
Capital of United States of America
Washington
Capital of United Kingdom
London
Capital of South Korea
Seoul
```

11.4 Detecting end-of-file

Detection of the end-of-file condition is necessary for preventing any further attempt to read data from the file. This was illustrated in Program 11.2 by using the statement

```
while(fin)
```

An **ifstream** object, such as **fin**, returns a value of 0 if any error occurs in the file operation including the end-of-file condition. Thus, the **while** loop terminates when **fin** returns a value of zero on reaching the end-of-file condition. Remember, this loop may terminate due to other failures as well. (We will discuss other error conditions later.)

There is another approach to detect the end-of-file condition. Note that we have used the following statement in Program 11.3:

```
    if(fin1.eof() != 0) {exit(1);}
```

eof() is a member function of **ios** class. It returns a non-zero value if the end-of-file(EOF) condition is encountered, and a zero, otherwise. Therefore, the above statement terminates the program on reaching the end of the file.