மனோன்மணியம் சுந்தரனார் பல்கலைக்கழகம்

MANONMANIAM SUNDARANAR UNIVERSITY
TIRUNELVELI-627 012

தொலைநிலை தொடர் கல்வி இயக்ககம்

DIRECTORATE OF DISTANCE AND CONTINUING EDUCATION

B.Sc. CHEMISTRY

II YEAR

Programming Language C++

Sub. Code: JECS41

Prepared by

Dr. S. KALAISELVI

Assistant Professor

Department of Mathematics

Sarah Tucker College(Autonomous), Tirunelvei-7.

B.Sc. CHEMISTRY–II YEAR

JECS41: Programming Language C++

SYLLABUS

Unit I

Introduction, Tokens, Key words, Identifiers and constants, Basic data types, User defined data types, storage classes, Derived data types, Symbolic constants.

Chapter 1: Sections 1.1.-1.6.

Unit II

Introduction, The main function, function prototyping, Call by reference, Return by references, Inline functions, Default arguments, constant Arguments, Recursion, Function overloading, Friend and virtual functions, Math library functions, C structures Revisited, Specifying a class, Defining member functions, A C++ program with class, Making an outside functions inline, Nesting member functions, Private member functions, Arrays within a class, Memory allocation for objects, Static member functions, Array of objects, objects as function arguments, Friend functions, Returning objects.

Chapter 2: Sections 2.1.-2.22.

Unit III

Introduction, Constructors, Parameterized constructors, Multiple constructors in a class, Constructors with default arguments, Dynamic initialization of objects, Copy constructor, Constructing Two-Dimensional arrays, constant objects, Destructors.

Chapter 3: Sections 3.1.-3.7.

Unit IV

Introduction, defining operator over loading, over loading unary operator, Overloading Binary operator, Overloading Binary operators using Friends, Manipulation of strings using operators, Some other Operator over loading examples, Rules for Over loading Operators

Chapter 4: Sections 4.1.-4.7.

Unit V

Introduction, Defining Derived classes, Single inheritance, Making a private member in heritable, multi-level in heritance, Multiple inheritance, Hierarchical inheritance, Hybrid inheritance.

Chapter 5: Sections 5.1-5.8.

Reference Books

1.ReemaThareja, Object Oriented Programming with C++, Oxford University Press (January 2018).

JECS41: Programming Language C++

CONTENTS

Unit I

Introduction, Tokens, Key words, Identifiers and constants, Basic data types, User defined data types, storage classes, Derived data types, Symbolic constants.

Chapter 1: Sections 1.1-1.6.

1.1. Introduction to C++

C++ is a general purpose programming language developed by Bjarne Stroustrup in 1979 at Bell Labs. Similar to C programming, C++ is also considered as an intermediate level language because it includes the features of both high-level and low-level languages. C++ is a very popular programming language that can be implemented on a wide variety of hardware and operating system platforms. It is a powerful language for high-performance applications, including writing operating systems, system software, application software, device drivers, embedded software, high-performance client and server applications, software engineering, graphics, games, and animation software.

C++ is a superset of the C language; it supports all features of C language and adds other new features such as classes, objects, polymorphism, inheritance, data abstraction, encapsulation, single-line comments using two forward slashes, strong type checking, and so on. C++ is an object-oriented programming (OOP) language and facilitates design, reuse, and maintenance for complex software. It has an extensive library to enable programmers to reuse the existing code. Generally, the number of instructions required to perform a task in C++ is comparatively less than those required to be written in other high-level languages. The code written in C++ is easy to write, debug, and modify.

1.1.1. History of C++

In 1979, Bjarne Stroustrup, while working for his Ph D thesis, used a language called Simula, which was specifically designed for simulations. It was the first language to support the features of OOP. Though the language was very powerful, it was too slow for practical use. Therefore, Stroustrup started working on 'C with Classes', with an aim to integrate OOP features such as classes, inheritance, inline functions, and default arguments with the C language. This new language was easily portable and fast, provided low-level functionality, and included OOP concepts. In the same period, C front (the first C with classes compiler) was developed to translate C with classes code to ordinary C. Most of the code for C front was written in C with classes, making it a self-hosting compiler. A self-hosting compiler compiles itself. However, the use of this compiler was abandoned in 1993 after it became difficult to integrate new features such as exceptions into it.

In 1983, the name of the language was changed to C++, where ++ refers to adding new features in the C language. At this time, some more features such as virtual functions, function overloading, references with the & symbol, const keyword, and single-line comments were added. In 1985, though C++ was not officially standardized, it was implemented as a commercial product. In 1989, C++ was again updated to include protected members, static members, and multiple inheritance. In 1990, Turbo C++ compiler was released and added additional libraries. In 1998, the first international standard for C++ that included the standard template library, called the C++98, was published. However, this version had some problems which were addressed in the version released in 2003. It was called C++03.

The latest version of C++ known as C++11 was released in 2011. It added new features such as support for regular expression, a new C++ time library, atomics support, a standard threading library, a new for loop syntax (similar to for each loop), the auto keyword, new container classes, and better support for unions and array-initialization.

Structure of C++ Program

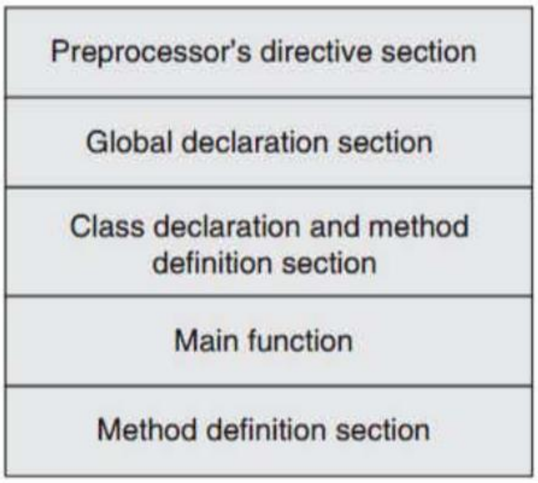
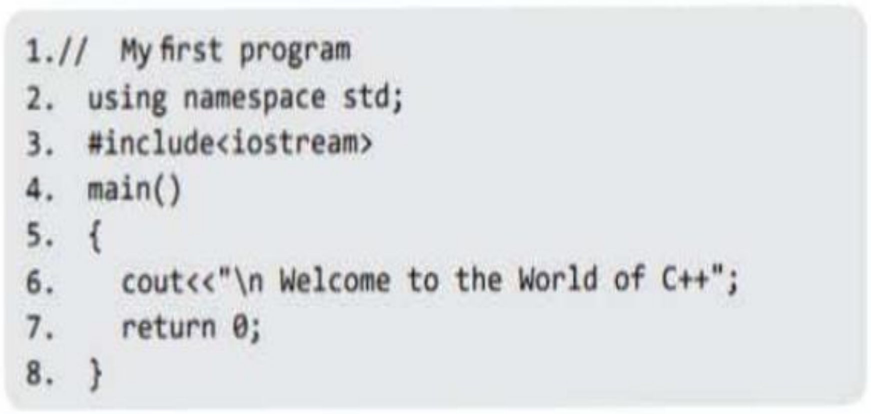

Preprocessor's directive section
Global declaration section
Class declaration and method definition section
Main function
Method definition section

Figure 1.1. Structure of a C++ program

A C++ program is composed of a pre-processor directives section, global declaration section, class declaration and method definition section, and a main function as shown in Fig. 1.1. The pre-processor directives contain special instructions that indicate how to prepare the program for compilation. One of the most important and commonly used preprocessor commands is 'include', which informs the compiler that some information is needed from the specified header file to execute the program. In this section, we will omit the global declaration part and revisit it in the chapter on 'Functions'. Usually, it is used to declare data (variables), functions, and structures that have global scope. The class declaration and method definition section can be considered as a part of global declaration section that is used to declare classes. A class consists of data and methods. The methods or functions of the class are also defined in this section. A C++ program may contain any number of functions. A function is defined as a group of C++ statements that are executed together and written in a logical sequence to perform a specific task. The 'main()' function is the most important function and is a part of every C++ program. Rather, the execution of a C++ program begins at this function as the operating system automatically calls it main(). This means that main() is the entry point for all the functions. All other functions, if present in the program, are called from main(). The method definition section is optional and required only if there are functions other than class methods in the program. This section was also added to support C++ programs.

1.1.2. Writing the first C++ program:

To write a C++ program, we need to write the code. We need to open a text editor. If you are a Windows user, you may use Notepad and if you prefer working on UNIX or Linux, you can use emacs or vi. Once the text editor is opened on your screen, type the following statements.



```
// My first program
using namespace std;
#include<iostream>
main()
{
   cout<<"\n Welcome to the World of C++";
   return 0;
}
```

Note:

The cout and return statements have been indented or moved away from the left side. This is done to make the code more readable.

After writing this code, save the file with any name with a.cpp extension. For example, let us save this file with the name, 'first.cpp'. Once Turbo C++ is installed in your computer and if you are using Windows, open the command prompt by clicking 'Start->Run', type 'command', and click Ok.

Using the command prompt, change to the directory in which you had saved your file and then type the following:

C:\>tcc first.cpp

This command is used to compile your C++ program. If there are any mistakes in the program, the compiler point out the mistake and the line on which it was done. In case of errors, you need to reopen your.cpp file and correct those mistakes. However, if everything is correct, no errors will be reported and the compiler will create an exe file for your program. This.exe file can be directly run by typing the following:

C:\>first.exe or simply C:\>first

Linux users should write$ g++ first.cpp to compile the program, and then type $./a.out to execute it. This will give you the following output on the screen

Welcome to the World of C++

Let us now try to understand the meaning of each line of the program.

Line 1. // My first program

The statement begins with two slash signs to indicate that the rest of the line is a comment. A comment is specifically inserted by the programmer to include short explanations concerning the code. This makes the programs more readable and understandable by the users. These comments have no effect on execution of the program. In the program mentioned here, comment has been simply included to give a brief introductory description of the program.

Line 2. using namespace std;

Let us take an analogy if a team has four members, then you can either refer to each person individually or just say the entire team. When you use the word 'team' it means every individual member. Same is the case with namespaces. We will read about namespaces in the latter chapters but for now just understand that namespace is a collection of important functions and classes. There can be many namespaces. So, every namespace is identified by a specific name. If you are using any particular thing from a namespace, then you have to write the statement using namespace name;

The statement using namespace std; makes all the names from the namespace std available in the current program. std is an abbreviation of standard. It is the standard namespace in C++. cout, cin and a lot of other things are defined in it. If you don't write the using namespace std;

then the compiler will not be able to understand which classes or functions you are trying to access and will thus, generate a compiler error.

Line 3. #include

This is a preprocessor command that comes as the first statement in our code. All preprocessor commands start with a hash symbol (#). The preprocessor commands are not executed by the compiler. Rather, they are read and interpreted by the preprocessor. It is a special line interpreted before the commencement of the compilation of the program. The #include statement tells the compiler to include the standard input or output library or header file (iostream) in the program. This file has some in-built functions. By including this file in our code, we can use these functions directly. The standard input or output header file contains functions for input and output of data such as reading values from the keyboard and printing the results on the screen.

Line 4. int main()

int is the return value ofthe main function. After all the statements in the program have been written, the last statement of the program will return an integer value (zero for successful execution and a non-zero value for any abnormal termination) to the operating system. The statement int main() is basically the function declaration. We will read more on function declaration in chapter 4.

Note: By default the return type of main() function is int. So, writing main() or int main() is equivalent.

Lines 5&8. { }

The two curly brackets are used to group all related statements of the function main. All statements between the braces form the function body. The function body contains a set of instructions to perform the given task.

Line 6. cout<<"\n Welcome to the World of C++";

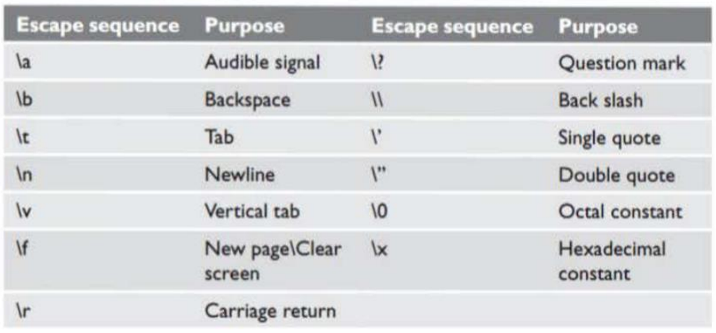
The cout function is defined in the iostream.h file and is used to print text on the screen. The message that has to be displayed on the screen is enclosed within double quotes. The '\n' is an escape sequence and represents a newline character. It is used to print the message on a new line on the screen. Programming Tip: Placing a semi-colon after the parenthesis of main will generate a compiler error. An escape sequence is a combination of characters that is translated into another character or a sequence of characters that may be difficult or impossible to represent directly. Similar to the newline character, the other escape sequences supported by C++ are shown in Table 2.1. The end of the statement is marked with semicolon (;).

Note

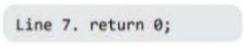
1. Escape sequences are actually non-printing control characters that begin with a backslash (1).

2. statement in the main function ends with a semi-colon (;).

Table 1.1 Escape sequences



Escape sequence	Purpose	Escape sequence	Purpose
\a	Audible signal	\?	Question mark
\b	Backspace	\\	Back slash
\t	Tab	\'	Single quote
\n	Newline	\"	Double quote
\v	Vertical tab	\0	Octal constant
\f	New page\Clear screen	\x	Hexadecimal constant
\r	Carriage return		



```
Line 7. return 0;
```

1.2.Tokens in C++ :

Tokens are the basic buildings blocks in C++ language. You may think of a token as the smallest individual unit in a C++ program. This means that a program is constructed using a combination these tokens. There are six main types of tokens in C++. They are shown in the Fig. 1.1.

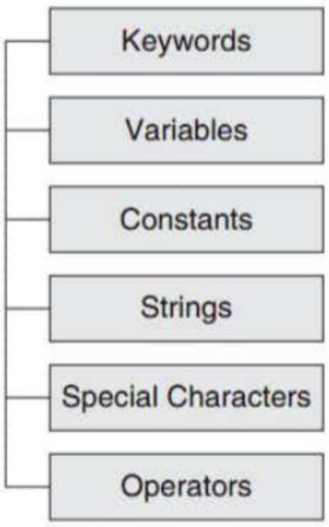
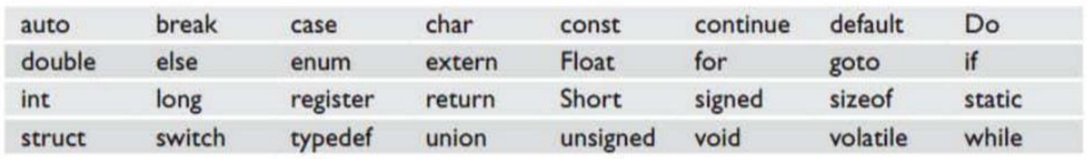


Figure1.1. Tokens is C++

1.3. Keywords:

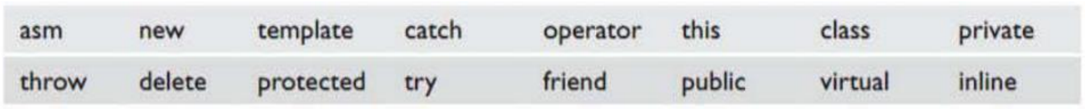
Similar to every computer language, C++ has a set of reserved words, often known as keywords that cannot be used as an identifier. All keywords are basically a sequence of characters that have a fixed meaning. The meaning of a keyword cannot be changed by the programmer. Conventionally, all keywords must be written in lowercase. Table 2.3 contains a list of keywords that are common to C and C++. Table 2.4 lists some C++-specific keywords.

Table.1.2. Keywords common to C and C++ languages



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

Table 1.3. Keywords specific to C++ language



asm	new	template	catch	operator	this	class	private
throw	delete	protected	try	friend	public	virtual	inline

As you read this book, the meaning and utility of each keyword will become clearer to you.

1.4. Identifier:

Identifiers, as the name suggests, helps us to identify data and other objects in the program. Identifiers are basically the names given to program elements such as variables, arrays, and functions. An identifier may consist of an alphabet, digit, or an underscore.

The name cannot include any special characters or punctuation marks, except the underscore"_". There cannot be two successive underscores. Keywords cannot be used as identifiers. The case of alphabetic characters that form the identifier name is significant. For example, "FIRST" is different from "first" and "First". The identifier name must begin with an alphabet or an underscore. However, use of underscore as the first character must be avoided because several complier-defined identifiers in the standard C++ library have underscore as their first character. Therefore, inadvertently duplicated names may cause definition conflicts. Identifiers can be of any reasonable length. They should not contain more than 31 characters. They can be longer than 31; however, the compiler looks at only the first 31 characters of the name.

Although it is not compulsory, it is a good practice to use meaningful identifier names. Good identifiers are descriptive but short. To cut short the identifier, you may use abbreviations. C++

allows identifiers (names) to be up to 63 characters long. If a name is longer than 63 characters, then only the first 63 characters are used. As a general practice, if the identifier is a little long (more than 4-5 characters), then you may use an underscore to separate the parts of the name or you may use capital letters for each part.

Examples of valid identifiers include the following:

roll_number, marks, name, emp_number, basic_pay, HRA, DA, dept_code, DeptCode, RollNo, EMP NO

Examples of invalid identifiers include the following:

23_student, %marks, @name, #emp_number, basic.pay, -HRA, (DA), &dept_code, auto

Note:

C++ is a case-sensitive language. Therefore, rno, Rno, RNo, and RNO are different identifiers.

1.5.Constants:

Constants are identifiers whose value does not change. While variables can change their value at any time, constants can never change their value. Constants are used to define fixed values such as Pi or the charge on an electron so that their value does not get changed in the program even by mistake. A constant is an explicit data value specified by the programmer. The value of the constant is known to the compiler at the compile time. C++ allows the programmer to specify constants of integer type, floating point type, character type, and string type (refer to Fig. 2.2).

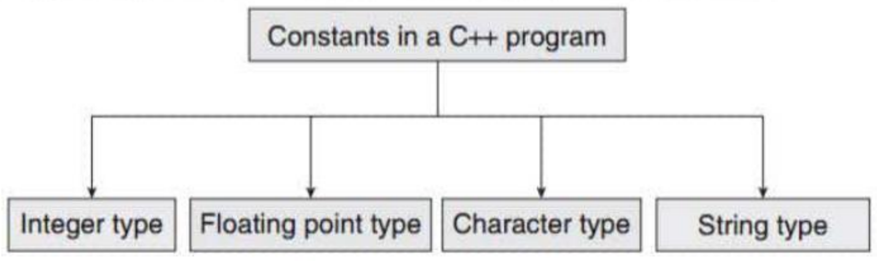


Figure 2.2. Constants in C++

1.5.1. Integer Constant:

A constant of integer type consists of a sequence of digits. For example, 1, 34, 567, and 8907 are valid integer constants. A literal (constant) integer such as 1234 is of type int by default. For a long integer constant, the literal is succeeded with either 'L' or '1' (such as 1234567L). Similarly, an unsigned int literal is written with a 'U' or 'u' suffix (ex, 120). Therefore, 1234L

12341 1234U 1234u 1234LU 1234ul are all valid integer constants. Literal integers can be expressed in decimal, octal, or hexadecimal notation. By default, an integer is expressed in decimal notation. Decimal integers consist of a set of digits from e through 9, preceded by an optional - or + sign. Examples of decimal integer constants include 123 -123 +123 e While writing integer constants, embedded spaces, commas and non-digit characters are not allowed. Therefore, integer constants given below are totally invalid in C++. 123 456 12, 34, 567 $123 123.456 An integer constant preceded by a zero (0) is an octal number. Octal integers consist of a set of digits, e through 7. Example of octal integers include

012 Θ 01234

Similarly, an integer constant is expressed in hexadecimal notation if it is preceded by 0x or ex. Hexadecimal numbers contain digits from 0-9 and alphabets from A through F. The alphabets A through F represent numbers 10 through 15. For example, decimal 72 is equivalent to 0110 in octal notation and 0x48 in hexadecimal notation. Example of octal integers include

0X12 0x7F 0xABCD 0X1A3B

Note:

A decimal integer constant is treated as an unsigned long if its magnitude exceeds that of signed long. An octal or hexadecimal integer that exceeds the limit of int is taken to be unsigned. If even this limit is exceeded, it is taken as long; and in case this limit is exceeded, it is treated as unsigned long.

1.5.2. Floating Point Constant:

Integer numbers are inadequate to express numbers that have a fractional part. A floating point constant, therefore, consists of an integer part, a decimal point, a fractional part, and an exponent field, containing an e or E (e means exponent) followed by an integer where the fraction part and integer part are sequence of digits. However, it is not necessary that every floating point constant must contain all these parts. Some floating point numbers may have certain parts missing. Some valid examples of floating point numbers are as follows: 0.02 -0.23 123.456 +0.34 123. .9 -.7 +.8

A literal such as 0.07 is treated as of type double by default. To make it a float-type literal, you must specify it using suffix F or f. Consider some valid floating point literals given below. (Note that suffix L is for long double)

0.02F 0.34f 3.141592654L 0.002146 2.146E-3

A floating-point number may also be expressed in scientific notation. In this notation, the mantissa is either a floating point number or an integer and exponent is an integer with an optional plus or minus sign. Therefore, the numbers given below are valid floating point numbers.

0.5e2 14E-2 1.2e+3 2.1E-3 -5.6e-2

Hence, we see that scientific notation is used to express numbers that are either very small or very large. For example,

120000000 = 1.2E8 and -0.000000025 = -2.5E-8:

1.5.3. Character Constant:

A character constant consists of a single character enclosed in single quotes. For example, 'a', @ are character constants. In computers, characters are stored using machine's character set for example using ASCII codes. Note that all escape sequences mentioned in Table 2.6 are also character constants.

1.5.4. String Constant:

A string constant is a sequence of characters enclosed in double quotes. Hence, 'a' is not the same as "a". The characters comprising the string constant are stored in successive memory locations.

When a string constant is encountered in a C++program, the compiler records the address of the first character and appends a null character ('\0') to the string to mark the end of the string. Therefore, the length of a string constant is equal to number of characters in the string plus 1 (for the null character). In the same manner, the length of string literal "he11o" is 6.

1.5.5. Declaring Constants:

To declare a constant, precede the normal variable declaration with const keyword and assign it a value. For example,

const data_type const_name = value;

The const keyword specifies that the value of pi cannot change.

const float PI = 3.14;

However, another way to designate a constant is to use the pre-processor command define. Similar to other pre-processor commands, define is preceded with a # symbol. Although #define statements can be placed anywhere in a C program, it is always recommended that these statements be placed at the beginning of the program to make them easy to find and modify at a later stage. Look at the example given here which defines the value of PI using define.

```
#define PI 3.14159
#define service_tax 0.12
```

In these examples, though the value of PI will never change, service tax may change. Whenever the value of the service tax is altered, it can be corrected only in the define statement. When the pre-processor reformats the program to be compiled by the compiler, it replaces each defined name (such as PI and service_tax) with its corresponding value, wherever it is found in the source program. Hence, it just works similar to the Find and Replace command available in a text editor. Let us have a look at some rules that needs to be applied to a #define statement which defines a constant.

Rule 1 Constant names are usually written in capital letters to visually distinguish them from other variable names which are normally written in lower case characters. Note that this is just a convention and not a rule.

Rule 2 No blank spaces are permitted in between the # symbol and define keyword.

Rule 3 Blank space must be used between #define and constant name and between constant name and constant value.

Rule 4 #define is a pre-processor compiler directive and not a statement. Therefore, it does not end with a semi-colon.

1.6.Data Types in C++:

Data are used to represent information. They can be classified into different categories or types shown in Fig.2.3.

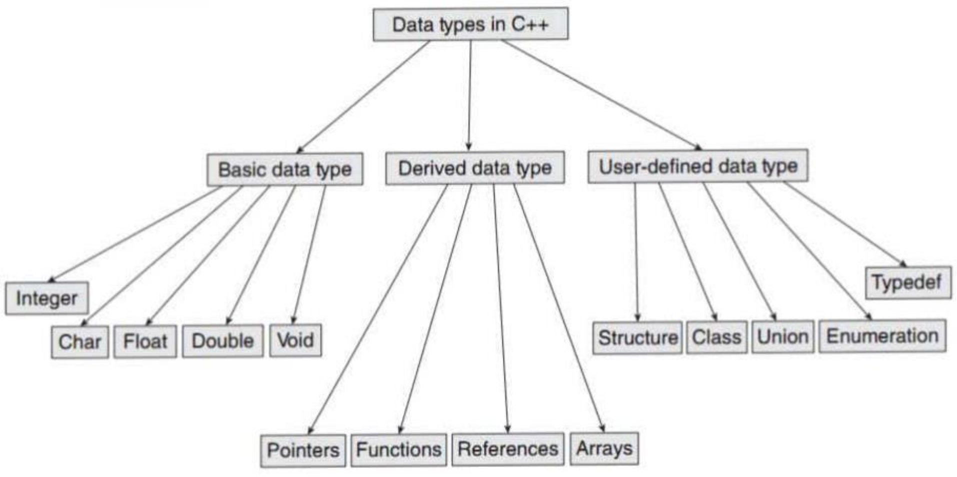


Figure 2.3. Data types in C++

In this section, we will read only about the basic data types in C++. The complex data types will be taken up in subsequent chapters.

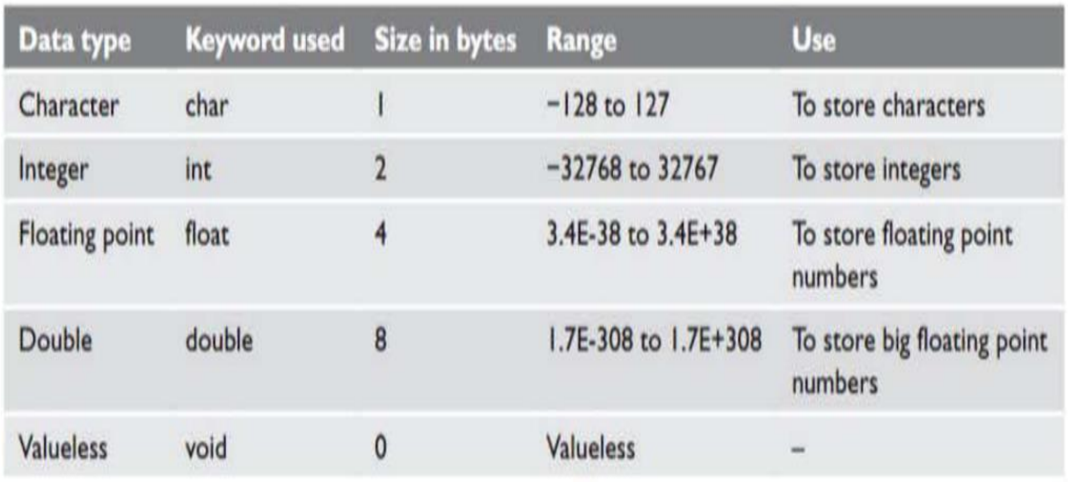
C++ language provides very few basic data types. Table 1.4 lists the data types, their keywords, size, range, and usage for a C++ programmer on a 16-bit computer.

In addition to this, we also have variants of char, int, and double data types.

The char data type is one byte and is used to store single characters. Note that C++ does not provide any data type for storing text. This is because the text is made up of individual characters.

Generally, char is supposed to store characters not numbers; however, the range of char is given as -128 to 127 here. The reason for this change is that in memory, characters are stored in their

Table 1.4. Basic data types in C++

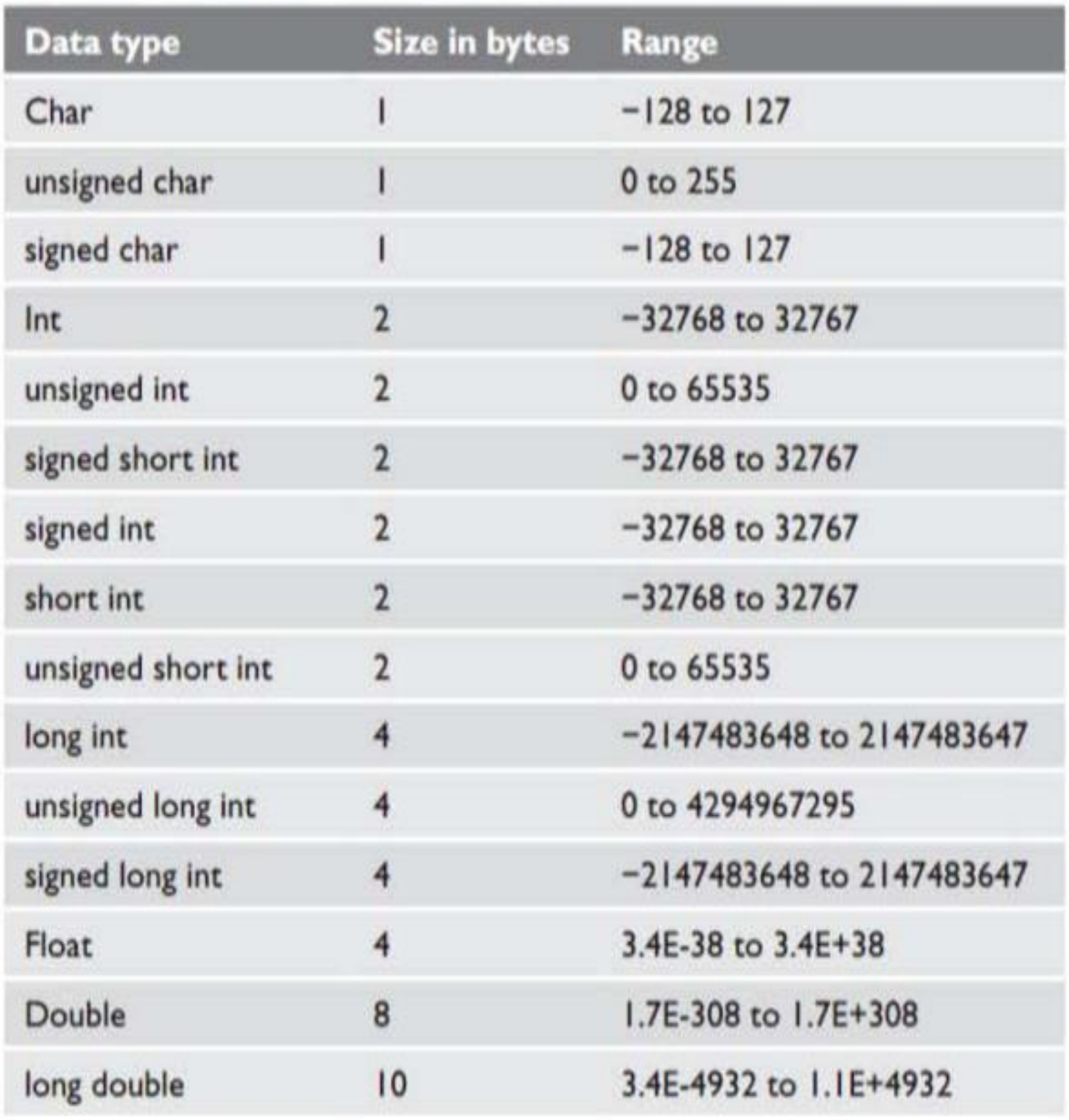


Data type	Keyword used	Size in bytes	Range	Use
Character	char	1	−128 to 127	To store characters
Integer	int	2	−32768 to 32767	To store integers
Floating point	float	4	3.4E-38 to 3.4E+38	To store floating point numbers
Double	double	8	1.7E-308 to 1.7E+308	To store big floating point numbers
Valueless	void	0	Valueless	–

ASCII codes. For example, the character 'A' has the ASCII code 65. In memory, we will not store 'A' but 65 in binary format.

Table 1.5 shows the variants of basic data types in detail.

Table 1.5. Detailed list data types



Data type	Size in 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 short int	2	−32768 to 32767
signed int	2	−32768 to 32767
short int	2	−32768 to 32767
unsigned short int	2	0 to 65535
long int	4	−2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
signed long int	4	−2147483648 to 2147483647
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

In Table 1.5, we have unsigned char and signed char, but do not have negative characters. The reason for having negative data types is because we use signed and unsigned char to ensure portability of programs that store non-character data as char.

While the smaller data types occupy less memory, the larger types incur a performance penalty. Although the data type we use for our variables does not have a big impact on the speed or memory usage of the application, we should always try to use int, unless there is a special need to use any other data type.

Last but not the least, the void type holds no value. It is primarily used in three cases as follows:

• To specify the return type of a function (when the function returns no value)

• To specify the parameters of the function (when the function accepts no arguments from the caller)

• To create generic pointer

Note:

Unsigned int/char liberates the sign bit and makes the entire word available for storage of the non-negative numbers.

Review Questions

1. What are variables?
2. What does the data type of a variable signify?
3. Give the structure of a C++ program.
4. What do you understand by identifiers and keywords?
5. Write a short note on basic data types that the C++ language supports.
6. How can we get formatted output in C++ programs?
7. Why do we include <iostream.h> in our programs?
8. Explain the utility of #define and #include statements**.**
9. Explain the types of data types.
10. Explain the user defined data types.

Unit II

Introduction, The main function, function prototyping, Call by reference, Return by references, Inline functions, Default arguments, constant Arguments, Recursion, Function overloading, Friend and virtual functions, Math library functions, C structures Revisited, Specifying a class, Defining member functions, A C++ program with class, Making an outside functions inline, Nesting member functions, Private member functions, Arrays within a class, Memory allocation for objects, Static member functions, Array of objects, objects as function arguments, Friend functions, Returning objects.

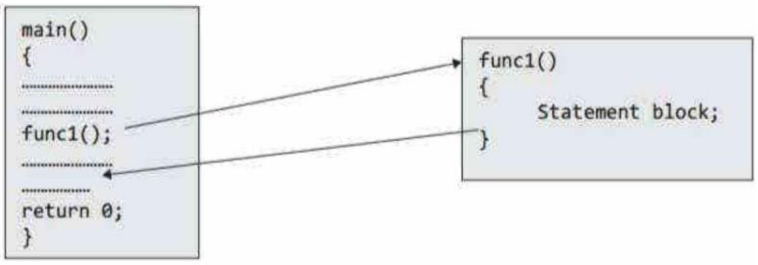
Chapter 2: Sections 2.1.-.

2.1. Introduction:

C＋＋ enables programmers to break up a program into segments commonly known as functions. Each function can be written more or less independently of the others. Every function in the program is supposed to perform a well-defined task. Therefore, the program code of one function is completely insulated from that of other functions.

Every function interfaces to the outside world in terms of how information is transferred to it and how results generated by the function are transmitted back from it. This interface is basically specified by the function name. For example, look at Fig. 2.1 which explains how the main() calls another function to perform a well-defined task.

In Fig. 2.1, we see that main() calls function named func1(). Therefore, main() is known as the calling function and func1() is known as the called function. The moment the compiler encounters a function call, instead of executing the next statement in the calling function, the control jumps to the statements that are a part of the called function. After the called function is executed, the control is returned back to the calling program.



```
main()
{
………………
………………
func1();
………………
………………
return 0;
}
```

```
func1()
{
    Statement block;
}
```

Figure 2.1 main() and func1()

It is not necessary that the main () can call only one function. It can call as many functions as it wants and as many times as it wants. For example, a function call placed within a for loop, while loop, or do-while loop may call the same function multiple times until the condition holds true.

Another point to note here is that it is not that only the main() can call another functions. Any function can call any other function. For example, look at Fig. 2.2 which shows one function calling another, and the other function, in turn, calling some other function.

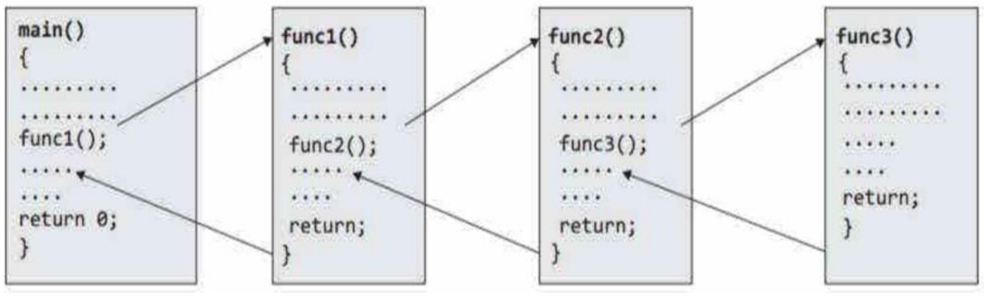


Figure 2.2 Function calling another function

Therefore, we see that every function encapsulates a set of operations and when called, it returns information to the calling program.

2.2. Need for Functions:

Analysing the reasons for segmenting a program into manageable chunks is an important aspect of programming.

- Dividing the program into separate well-defined functions facilitates each function to be written and tested separately. This simplifies the process of getting the total program to work. Figure 2.3 shows that the main() calls other functions for dividing the entire code into smaller sections (or functions).
- Understanding, coding, and testing multiple separate functions are far easier than doing the same for one huge function.
- If a big program has to be developed without using any function other than main(), there will be many lines in the main(). Maintaining the program will be a big mess. A large program is a serious issue in micro-computers where memory space is limited.
- All libraries in C + + contain a set of functions that programmers are free to use in their programs. These functions have been prewritten and pretested. Therefore,

programmers use them without worrying about their code details. This speeds up program development, by allowing the programmer to concentrate only on the code that

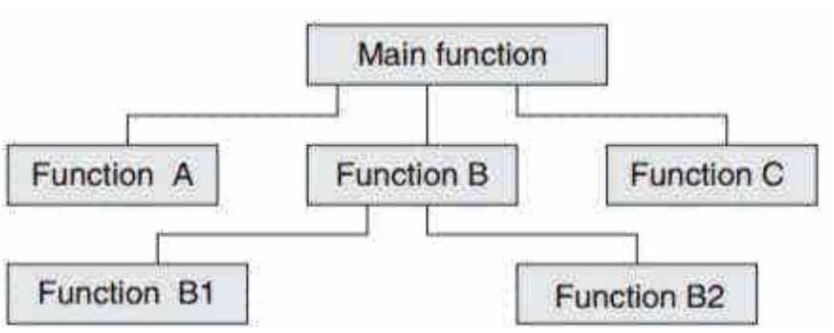
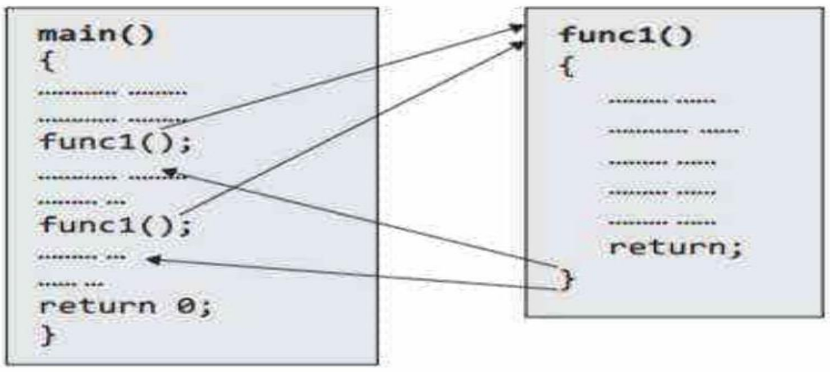


Figure 2.3 Top-down approach of solving a problem he has to write.

- When a big program is broken into comparatively smaller functions, different programmers working on that project can divide the workload by writing different functions.
- Like C + + libraries, programmers can also make their functions and use them from different points in the main program or any other program that needs its functionalities.



```
main()
{
........... ........
........... ........
func1();
........... ........
........ ...
func1();
........ ...
...... ...
return 0;
}
```

```
func1()
{
    ........ .....
    .......... .....
    ........ .....
    ........ .....
    ........ .....
    return;
}
```

Figure 2.4 Function func1() called twice from the main()

Consider a program that executes a set of instructions repeatedly n times, though not continuously. In case the instructions had to be repeated continuously for n times, they can better be placed within a loop. However, if these instructions have to be executed abruptly from anywhere within the program code, instead of writing these instructions in all areas where they are required, it is better to place these instructions in a function and call that function wherever required. Figure 2.4 explains this concept.

2.3 Using Functions:

In the first chapter, we have mentioned that while executing a C + + program, the operating system calls the main() function which marks the entry point for execution of the program. When the program is executed, the main() returns some value to the operating system.

Any function, including main, can be compared to a black box that takes in input, processes it, and spits out the result. However, we may also have a function that does not take any inputs at all or that does not return anything at all.

While using functions, we will use the terminology as follows.

- A function, f, that calls another function g, is known as the calling function and g is known as the called function.
- The inputs that the function takes are known as arguments or parameters.
- When a called function returns some result back to the calling function, it is said to return that result.
- The calling function may or may not pass parameters to the called function. If the called function accepts the arguments, the calling function will pass parameters, otherwise, it will not do so.
- Function declaration is a declaration statement that identifies a function with its name, a list of arguments that it accepts, and the type of data it returns.
- Function definition consists of a function header that identifies the function, followed by the body of the function containing the executable code for that function

2.4. Function Declaration or Function Prototype:

Before using the function, the compiler must know about the number and types of parameters that the function expects to receive and the data type of value that it will return to the calling program. For this, we need to declare a function (or give the function prototype). Placing the function declaration statement prior to its use enables the compiler to make a check on the arguments used while calling that function.

The general format for declaring a function that accepts some arguments and returns some value as result can be given as follows:

```
return_data_type function_name (data_type variable1, data_type variable2,..);
```

We must note the following here.

- function_name is a valid name for the function. Naming a function follows the same rules that naming variables follows. A function should have a meaningful name that must clarify the task

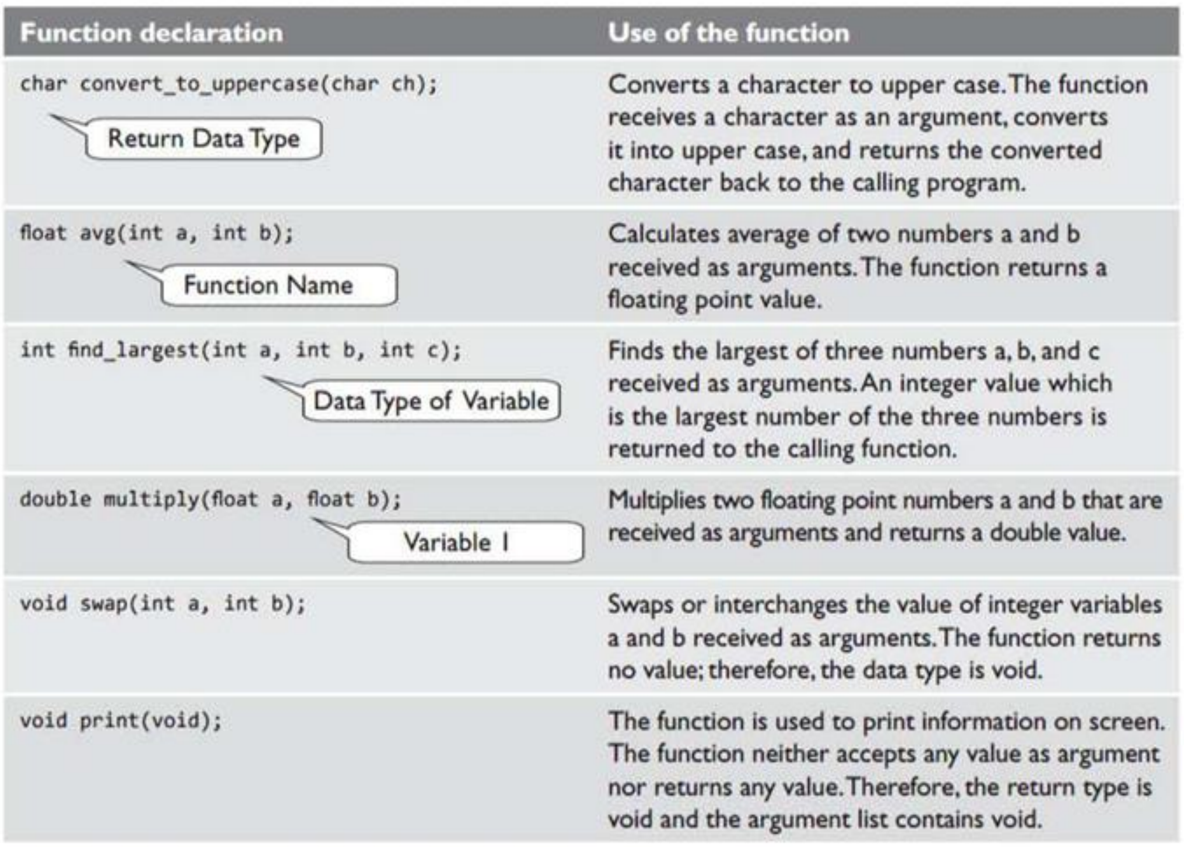
Programming

Tip: Not placing a semicolon after the function declaration is an error.

that it will perform. The function name is used to call it for execution in a program. Every function must have a different name that indicates the particular job that the function does.

- return_data_type specifies the data type of the value that will be returned to the calling function as a result of the processing performed by the called function.
- data_type variable1, data_type variable2, ... is a list of variables of specified data types. These variables are passed from the calling function to the called function. They are also known as arguments or parameters that the called function accept to perform its task. Table 2.1 shows examples of valid function declarations in C + +.

Table 2.1 Valid function declarations



Function declaration	Use of the function
`char convert_to_uppercase(char ch);` Return Data Type	Converts a character to upper case. The function receives a character as an argument, converts it into upper case, and returns the converted character back to the calling program.
`float avg(int a, int b);` Function Name	Calculates average of two numbers a and b received as arguments. The function returns a floating point value.
`int find_largest(int a, int b, int c);` Data Type of Variable	Finds the largest of three numbers a, b, and c received as arguments. An integer value which is the largest number of the three numbers is returned to the calling function.
`double multiply(float a, float b);` Variable I	Multiplies two floating point numbers a and b that are received as arguments and returns a double value.
`void swap(int a, int b);`	Swaps or interchanges the value of integer variables a and b received as arguments. The function returns no value; therefore, the data type is void.
`void print(void);`	The function is used to print information on screen. The function neither accepts any value as argument nor returns any value. Therefore, the return type is void and the argument list contains void.

The following are the key points to remember about function declaration:

- After the declaration of every function, there is a semicolon. If the semicolon is missing, the compiler will generate an error message.

Programming

Tip: Though optional, use argument names in the function declaration.

- The name of the function is global. Therefore, the declared function can be called from any point in the program.
- Use of argument names in the function declaration statement is optional. Both declaration statements are valid in C + +.

```
int func(int, char, float); or
int func(int num, char ch, float fnum);
```

- No function can be declared within the body of another function.
- A function having void as its return type cannot return any value.
- A function having void as its parameter list cannot accept any value. A function is declared as

```
void print(void); OR void print()
```

does not accept any input/arguments from the calling function.

- If the function declaration does not specify any return type, by default, the function returns an integer value. Therefore, when a function is declared as

```
sum(int a, int b);
```

the function sum accepts two integer values from the calling function and, in turn, returns an integer value to the caller.

- Some compilers make it compulsory to declare the function before its usage while other compilers make it optional. However, it is always good to declare the function before its usage as it allows error checking on the arguments in the function call.

2.5 Function Definition:

When a function is defined, space is allocated for that function in the memory. A function definition comprises two parts as follows:

Programming

Tip: It is an error to place a semicolon after the function header in the function definition.

Function header

- Function body

The syntax of a function definition can be given as

```
return_data_type function_name(data_type variable1, data_type variable2,..)
{ ..........
   statements
   -m.......
   return( variable);
}
```

Note The number of arguments and the order of arguments in the function header must be same as that given in function declaration statement.

Programming

Tip: The parameter list in the function definition as well as function declaration must match with each other.

Programming

Tip: A function can be defined either before or after the main().

While return_data_type function_name(data_type variable1, data_type variable2,..) are known as the function header, the rest of the portion comprising program statements within { } is the function body which contains the code to perform the specific task.

The function header is same as that of function declaration. The only difference between the two terms is that a function header is not followed by a semicolon. The list of variables in the function header is known as the formal parameter list. The parameter list may have zero or more parameters of any data type. The function body contains instructions to perform the desired computation in a function.

The function definition itself can act as an implicit function declaration. Therefore, the programmer may skip the function declaration statement if the function is defined before being used.

Note The argument names in the function declaration and function definition need not be the same. However, the data types of the arguments must match with that specified in function declaration and function definition.

2.6. Call-by-Reference:

When the calling function passes arguments to the called function using call-by-value method, the only way to return the modified value of the argument to the caller is by using the return statement explicitly. A better option when a function can modify the value of the argument is to pass arguments using call-by-reference technique. In call-by-reference, we declare the function parameters as references rather than normal variables. When this is done, any changes made by the function to the arguments it received are visible by the calling program.

To indicate that an argument is passed using call-by-reference, an ampersand sign (8) is placed after the type in the parameter list. In this way, changes made to that parameter in the called function body will be reflected in its value in the calling program.

Hence, in a call-by-reference method, a function receives an implicit reference to the argument, rather than a copy of its value. Therefore, the function can modify the value of the variable and that change will be reflected in the calling function as well. The following program explains this concept.

Note

The reference type parameters are accessed in the same way as other normal variables are accessed.

Example 2.1: Program that uses call-by-reference method to pass arguments to the called function

```
using namespace std;
#include<iostream>
void add(int &n);
int main()
{ int num = 2;
   cout<<"\n The value of num before calling the function = "<<num;
   add(num);
   cout<<"\n The value of num after calling the function = "<<num;
}
void add(int &n)
{ n=n+10;
   cout<<"\n The value of num in the called function = "<<n;
}
```

OUTPUT

```
The value of num before calling the function =2
The value of num in the called function = 12
The value of num after calling the function = 12
```

Pros and Cons of Call-by-Reference

The advantages of using call-by-reference technique of passing arguments include the following:

- Since arguments are not copied into new variables, it provides greater time and space efficiency
- The function can change the value of the argument and the change is reflected in the caller.
- A function can return only one value. In case, we need to return multiple values, pass those arguments by reference, so that modified values are visible in the calling function.

Note We will make a comparison between call-by-reference and call-by-address in Chapter 7, where we will read about pointer variables and using address in detail.

Example 2.2:

Program to swap the values of two variables using call-by-value and call-by-reference mechanisms. Note the value of integers in the calling function and the called function

```
using namespace std;
#include<iostream>
void swap_call_by_val(int, int);
void swap_call_by_add(int *, int *);
void swap_call_by_ref(int &, int &);
int main()
{ int a=1,b=2,c=3,d=4;
   cout<<"\n In main(), a = "<<a<<" and b = "<<<b;
   swap_call_by_val(a, b);
   cout<<"\n In main(), a = "<<a<<" and b = "<<b;
   cout<<"\n\n In main(), c= "<<c<<" and d = "<<d;
   swap_call_by_add(&c, &d);
   cout<<"\n In main(), c="<<c<<<" and d= "<<d;
   swap_call_by_ref(c, d);
   cout<<"\n In main(), c= "<<c<<" and d= "<<d;
}
void swap_call_by_val(int a, int b)
{ int temp;
   temp = a;
   a = b;
   b = temp;
   cout<<"\n In function (Call-By-Value Method) - a = "<<a<<" and b = "<<b;
}
void swap_call_by_add(int *c, int *d)
{ int temp;
   temp = *c; // *operator used to refer to the value
   *c=*d;
   *d = temp;
   cout<<"\n In function (Call-By-Address Method) - c="<<*C<<" and d= "<<*d;
}
void swap_call_by_ref(int &c, int &d)
{
   int temp;
   temp = c;
   c=d;
   d = temp;
   cout<<"\n In function (Call-By-Reference Method) - c = "<<c<<" and d = "<<d;
}
```

OUTPUT

```
In main(), a = 1 and b = 2
In function (Call-By-Value Method) −a = 2 and b = 1
In main(), a = 1 and b = 2
In main(), c=3 and d=4
In function (Call-By-Reference Method) - c=4 and d=3
In main(), c=4 and d=3
In main(), c=3 and d=4
In function (Call-By-Address Method) - c=4 and d=3
In main(), c=4 and d=3
```

Program 2.1: Write a program to find the biggest of three integers using functions.

```
using namespace std;
#include<iostream>
int larger(int a, int b, intc);
int main()
{ int num1, num2, num3, large;
   cout<<"\n Enter the three numbers : ";
   cin>>num1>>num2>>num3;
   large = larger(num1, num2, num3);
   cout<<"\n Largest number = "<<large;
}
int larger(int a, int b, int c)
{ if(a>b && a>c)
   return a;
   if(b>a && b>c)
      return b;
   else
      return c;
}
OUTPUT
Enter the three numbers : 45 23 34
Largest number = 45
```

Program 2.2: Write a program to calculate the area of a circle using functions.

```
using namespace std;
#include<iostream>
float cal_area(float r );
int main()
{ float area, radius;
cout<<" $$ n Enter the radius of the circle : ";
cin>>radius;
area = cal_area(radius);
cout. precision(2);
cout<<" $$ n Area of the circle with radius "<<radius<<" = "<< area;
}
float cal_area(float radius)
{ return ( 3.14 * radius * radius);
}
```

OUTPUT

```
Enter the radius of the circle : 7
Area of the circle with radius 7 = 153.83
```

Program 2.3:

Write a program to find whether a number is even or odd using functions.

```
using namespace std;
#include<iostream>
int evenodd(int);
int main()
{ int num, flag;
   cout<<"\n Enter the number : ";
   cin>>num;
   flag = evenodd(num);
   if (flag == 1)
   cout<<"\n"<<num<<" is EVEN";
   else
      cout<<"\n"<<num<<" is ODD";
}
int evenodd(int a)
{ if(a%2 == 0)
      return 1;
   else
      return 0;
}
```

OUTPUT

```
Enter the number : 42
EVEN
```

Program 2.4:

Write a program to convert time into minutes.

```
using namespace std;
#include<iostream>
int convert_time_in_mins(int hrs, int minutes);
main()
f int hrs, minutes, total_mins;
cout<<" ln Enter the hours and minutes : ";
cin>>hrs>>minutes;
total_mins = convert_time_in_mins(hrs, minutes);
cout<<"\n Total minutes = "<<total_mins;
}
int convert_time_in_mins(int hrs, int minutes)
{ int mins;
mins = hrs*60 + minutes;
return mins;
}
```

OUTPUT

```
Enter the hours and minutes : 430
Total minutes = 270
```

Program 2.5:

Write a program to calculate $P(n/r)$.

```
using namespace std;
#include<iostream>
int Fact(int num);
main()
{ int n,r;
float result;
cout <<" \ n Enter the value of n and r: ";
cin>>n>>r;
result = (float)Fact (n)/ Fact (n − r);
cout<<" \ nP(n/r) − P(" ≪ n ≪ ")/(" ≪ r ≪ ") = " ≪ result;
}
int Fact(int num)
{ int f = 1, i;
for ( i = num; i >= 1; i − − )
f = f ∗ i;
   return f;
}
```

OUTPUT

```
Enter the value of }n\mathrm{ and }r:4
P(n/r)-P(4)/(2)=12.00
```

Program 2.6:

Write a program to calculate C(n/r).

```
using namespace std;
#include<iostream>
int Fact(int num)
main()
{ int n, r;
   float result;
   cout<<"\n Enter the value of n and r : ";
   cin>>n>>r;
   result =(float)Fact(n)/(Fact(r)*Fact (n-r));
   cout.precision(2);
   cout<<"|n C(n/r) - C("<<n<<"/"<<r<<") = "<<result;
}
int Fact(int num)
{ int f=1,i;
   for(i=num;i>= 1;i--)
      f=f*i;
   return f;
}
```

OUTPUT

```
Enter the value of }n\mathrm{ and }r:4
C(n/r)-C(4)/(4)=6.00
```

Program 2.7:

Write a program to sum the series 1/1! +1/2!+1/3!++1/n!}\mathrm{ .

```
using namespace std;
#include<iostream>
int Fact(int);
main()
{ int n, f, i;
   float result = 0.0;
   cout<<"\n Enter the value of n : ";
   cin>>n;
   for(i=1;i<< n;i++)
   { f=Fact(i);
      result += 1/(float)f;
   }
   cout<<"\n The sum of the series 1/1!+1/2!+1/3!.. = "<< result;
}
int Fact(int num)
{ int f=1,i;
   for(i=num;i>= 1;i--)
      f=f*i;
   return f;
}
```

OUTPUT

```
Enter the value of $n: 5$
The sum of the series $1 / 1!+1 / 2!+1 / 3!\ldots=1.716667$
```

Program 2.8:

Write a program to sum the series 1/1!+4/2!+27/3! +

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

```
#include<math.h>
int Fact(int);
main()
{ int n, i, NUM, DENO;
  float sum = 0.0;
  cout<<"\n Enter the value of n : ";
  cin>>n;
  for(i=1;i<= n;i++)
  { NUM = pow(i,i);
    DENO = Fact(i);
    sum t= (float)NUM/DENO;
  }
  cout<<"\n 1/1! + 4/2! + 27/3! + == "<<sum;
}
int Fact(int n)
{ int f=1, i;
  for(i=n;i>=1;i--)
    f=f*i;
  return f;
}
```

OUTPUT

```
Enter the value of n : 5
1/1!+4/2!+27/3!+=44.208332
```

2.7. Return By Reference:

Similar to call-by-reference, a C++ program can also return a value by reference. This allows a function to be used on the left side of an assignment statement.

The following are the key points to remember while returning by reference:

- The function should not return a local variable by reference as it will go out of scope immediately as soon as the function ends. The code given below illustrates this concept. Though the compiler will not generate any error but will definitely issue a warning stating "reference to local variable ' x ' returned"

```
int &my_func(int num)
{ int x = num*2;
return x; //x is a local variable, you should not return it as reference
}
```

The function may return a reference to a static variable. Therefore, the code given here is permissible in C + +.

```
#include<iostream>
#include<cstring>
using namespace std;
int &my_func()
{ static int x=10;
return x;
}
main()
{
   int x;
   x = my_func();
   cout<<x;
}
```

OUTPUT

```
10
```

- Variables passed by reference can be returned by reference.
- Return by reference is extensively used to return structure variables and objects of classes.

Note

Values returned by reference must be variables. Returning a reference to a literal or an expression is not allowed.

Example 2.3.

To return value using reference

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

```
int &larger(int &x, int &y)
{ if(x>y)
   return x;
   else
      return y;
}
main()
{ int num1, num2, large;
   cout<<"\n Enter two numbers : ";
   cin>>num1>>num2;
   cout<<"\n Two numbers are : "<<num1<<" "<<num2;
   large = larger(num1, num2);
   cout<<"\n Large = "<<large;
   larger(num1, num2) = -1;
   cout<<"\n Two numbers are : "<<num1<<" "<<num2;
}
```

OUTPUT

```
Enter two numbers : 52
Two numbers are : 52
Large = 5
Two numbers are : -12
```

Explanation: In the given here, variables num 1 and num2 are passed by reference to greater(). The function will make a reference to variable x or y whichever is greater and return it to the caller. When we write

```
greater(num1, num2) = - 1;
```

the statement will assign -1 to the variable having greater value. Therefore, the output is obtained.

2.8. Inline Functions:

C++ provides a very useful feature of inline functions that is commonly used with classes. The major difference between an ordinary function and an inline function is that when an inline function is called, the compiler places a copy of its code at each point of call. As in case of an ordinary function, the compiler does not have to jump to the called function. This saves the function call overhead and results in faster execution of the code. We can make a function inline by taking caring of two aspects as follows.

- First, write the keyword inline before the function name.
- Second, define that function before any calls are made to it.

However, the programmer must not forget that keyword inline just makes a request to the compiler to make the function inline (and place its code at each point of call), the compiler may ignore the request if the function has too many lines.

Note Inline functions work best with short functions that are executed frequently.

Let us see two programs that use the concept of inline functions.

Example 2.4: To find the larger number using an inline function

Programming
Tip: main()
cannot be used
as an inline
function.

```
using namespace std;
#include<iostream>
inline int larger(int }x\mathrm{ , int }y\mathrm{ ) // Function definition
{ return (x>y)? x : y; // Function returning larger of the two nos
}
int main( )
{ int num1, num2;
   cout << "\n Enter two numbers: ";
   cin>>num1>>num2;
      cout<<"\n The larger number is : "<<larger(num1,num2); // Function call
   }
```

```
OUTPUT
Enter two numbers : 59
The larger number is : 9
```

Example 2.5: To find square of a number using inline function

```
using namespace std;
#include<iostream>
inline int sqr(int x)
{return (x*x);
}
int main()
{ int num;
   cout<<"\n Enter a number : ";
   cin>>num;
   cout<<"\n Square of "<<num<<" = "<<sqr(num);
}
```

```
OUTPUT
Enter a number : }
Square of 9=81
```

2.8.1 Advantages and Disadvantages of Inline Functions

The advantages of using inline function are as follows:

- An inline function generates faster code as it saves the time required to execute function calls.
- Small inline functions (three lines or less) create less code than the equivalent function call as the compiler does not have to generate code to handle function arguments and a return value.
- Inline functions are subject to code optimizations that are usually not available to normal functions as the compiler does not perform inter-procedural optimizations.
- Inline function avoids function call overhead. As a result, we need to save variables and other program parameters on the system stack.
- Since there are no function calls, the overhead of returning from a function is also avoided.

- It allows the compiler to apply intra-procedural optimization

The disadvantages of using inline function are as follows:

- Size of the program increases.
- Large programs may take longer time to be executed.
- At times, the program may not fit in the cache memory.
- There may be a problem in making efficient use of CPU registers if the inline function has many register variables.
- If the code of the inline function is modified, the entire program needs to be re-compiled.
- Inline functions should not be used for designing embedded systems due to memory size constraints.

2.8.2 Comparison of Inline Functions with Macros:

The concept of inline functions is similar to that of macros (refer Annexure 5). However, inline functions have a good blend of flexibility and power offered by macros as well as an ordinary function. Therefore, inline functions should be preferred over macros that were extensively used in C language due to the following reasons:

- Macro invocations skip the job of type checking; this is a must-to-do work in function calls.
- Macros cannot return a value while a function can return a value.
- Macros use mere textual substitution which can give unintended results due to inaccurate reevaluation of arguments and order of operations.
- Debugging of compiler errors in case of macros is more difficult than debugging functions.
- All constructs cannot be expressed using macros; however, with functions, they can be expressed with ease.
- Macros have a slightly difficult syntax while the syntax of writing function is similar to that of a normal function.

Example 2.6: To show why the use of inline function is better than macros

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

```
#define SQUARE (x) x * x
inline int sqr(int }x\mathrm{ )
{return (x*x);
}
int main()
{ int num, n;
   cout<<"\n Enter a number : ";
   cín>>num;
   n= num;
   cout<<"\n square of "<<num<<" using Inline function = "<<sqr(++num);
   cout<<"\n Square of "<<n<<" using Macros = "<<SQUARE(++n);
}
```

```
OUTPUT
Enter a number : 6
Square of }7\mathrm{ using Inline function = 49
Square of 8 using Macros = 64
```

Explanation:

In this program, you will find that due to wrong evaluation of arguments, macros give wrong results.

Inline functions cannot be used with the following:

- Recursive functions
- Functions having static variables
- Functions that return a value and also have go to statements, switch statements, or iterative statements
- Functions that do not return a value but have a return statement

2.9. Default Arguments:

Till now we have studied that when a function is called, all its arguments must be passed to it in totality. However, C++ gives little freedom to programmers by providing default arguments. When a function is specified with default arguments, the function can be called with missing arguments. When a function is called with a missing argument, the function assigns a default value to the parameter. This default value is specified by the programmer in the function declaration statement.

While specifying the default values during function declaration, the programmer must keep the following in mind:

- Only trailing arguments can have default values; therefore, specify them from right to left.
- No argument specified in the middle can have default values.

Let us analyse some function declarations with default arguments to understand which declaration is correct.

```
int my_func(int a, int b, int c=10); //Correct
int my_func(int }a\mathrm{ , int }b=5\mathrm{ , int }c=10); //Correc
int my_func(int a = 1, int b = 5, int c = 10); //Correct
int my_func(int }a\mathrm{ , int }b=5\mathrm{ , int c); //Wrong
```

Note

Default arguments are used when arguments usually have the same value (with or without some exceptional cases).

Some example codes which demonstrate the usage of default arguments are as follows.

Programming Tip: Arguments explicitly specified in function call override the default values given during function declaration.

Program 2.9:

Write a program to calculate the volume of a cuboid using default arguments.

```
using namespace std;
#include<iostream>
int Volume ( int length, int width = 3, int height = 4 );
int main()
{cout <<" | n Volume = " ≪ Volume (4, 6, 2);
cout ≪ " \ n \ n Volume = " ≪ Volume (4,6);
cout ≪ " \ n \ n Volume = " ≪ Volume (4);
}
int Volume(int length, int width, int height)
```

```
{ cout ≪ " ln Length = " ≪ length ≪ " Width = " ≪ width ≪ " and Height = "<<height;
return length * width * height;
}
```

OUTPUT

```
Length = 4 Width = 6 and Height = 2
Volume = 48
Length = 4 Width = 6 and Height = 4
Volume = 96
Length = 4 Width = 3 and Height = 4
Volume = 48
```

Program 4.10 Write a program to calculate simple interest. Suppose the customer is a senior citizen. He is being offered 12 per cent rate of interest; for all other customers, the ROI is 10 per cent.

```
using namespace std;
#include<iostream>
float interest(float principle, int years, int r = 10);
main()
{ float p;
   int n;
   char senior_citizen;
   cout<<"\n Enter the principal and number of years : ";
   cin>>p>>n;
   cout<<"\n Is the customer a senior citizen? (y/n) ";
   cin>>senior_citizen;
   if(senior_citizen == ' }\mp@subsup{y}{}{\prime}\mathrm{ ')
      cout<<"\n Interest = "<<interest( }p,n,12)
   else
      cout<<"\n Interest = "<<interest(p,n);
}
float interest(float principle, int years, int r)
{ return(principle*years*r/100);
```

}

OUTPUT

```
Enter the principal and number of years : 100005
Is the customer a senior citizen? (y/n)y
Interest = 6000
```

Program 4.11 Write a program to print the following pattern using default arguments.

```
%%%%%%
^^^^^^^^
^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^^^
```

```
using namespace std;
#include<iostream>
void print(char c='%', int n=6, int r=1);
main()
{ char c;
   int num_rows, num_cols;
   cout<<"\n Enter the character, number of rows and columns : ";
   cin>>c>>num_rows>>num_cols;
   print();
   print(c);
   print(c,10);
   print(c,15,2);
}
void print(char c, int n, int r)
{ cout<<"\n";
   for(int i = 0;i< r ; i++)
   { cout<<"\n";
      for(int j=0;j<n;j++)
      cout<<c;
   }}
```

2.10 Passing Constants as Arguments:

When parameters are passed by reference parameter, the called function may intentionally or inadvertently modify the actual parameters. However, at times, the programmer may strictly want the actual parameters not to be modified by the called function. In such cases, a constant parameter must be passed.

We have seen that pass by reference is a preferred technique of passing arguments to a function due to performance reasons. Therefore, using the const keyword allows programmers to achieve performance benefits while ensuring that the actual parameter is not modified. The following program explains this concept.

Example 2.7:

To demonstrates the use of a constant argument

```
using namespace std;
#include<iostream>
void add_2(int const &x)
{ x = x + 2; // ERROR, cannot modify a constant
   cout<<"\n The numbers is now: "<<x;
}
main ()
{int num1;
   cout<<"\n Enter a number: ";
   cin>>num1;
   add_2(num1);
}
OUTPUT
Error
```

Explanation:

The program given here will not execute and give a compiler error as the function cannot modify a constant. The function can use it but cannot alter it.

2.11. Recursion versus Iteration:

Recursion is more of a top-down approach to problem solving in which the original problem is divided into smaller sub-problems. On the contrary, iteration follows a bottom-up approach that begins with what is known and then constructing the solution step by step.

Recursion is an excellent way of solving complex problems especially when the problem can be defined in recursive terms. For such problems, a recursive code can be written and modified in a much simpler and clearer manner.

However, recursive solutions are not always the best solutions. In some cases, recursive programs may require substantial amount of run-time overhead. Therefore, when implementing a recursive solution, there is a trade-off involved between the time spent in constructing and maintaining the program and the cost incurred in running time and memory space required for the execution of the program.

Whenever, a recursive function requires some amount of overhead in the form of a run-time stack is always involved. Before jumping to the function with a smaller parameter, the original parameters, the local variables, and the return address of the calling function are all stored on the system stack. Therefore, while using recursion, a lot of time is needed to first push all the information on the stack when the function is called and then, time is again involved in retrieving the information stored on the stack once the control passes back to the calling function.

To conclude, one must use recursion only to find solution to a problem for which no obvious iterative solution is known. To summarize the concept of recursion, let us briefly discuss the pros and cons of recursion.

Pros and Cons of using a Recursive Program

The benefits of using a recursive program are as follows:

- Recursive solutions often tend to be shorter and simpler than non-recursive ones.
- Code is clearer and easier to use.
- Recursion represents like the original formula to solve a problem.

- Follows a divide and conquer technique to solve problems.
- In some (limited) instances, recursion may be more efficient.

The drawbacks of using a recursive program are as follows:

- For some programmers and readers, recursion is a difficult concept.
- Recursion is implemented using system stack. If the stack space on the system is limited, recursion to a deeper level will be difficult to implement.
- Aborting a recursive process in midstream is slow and sometimes nasty.
- Using a recursive function takes more memory and time to execute as compared to its non-recursive counterpart.
- It is difficult to find bugs, particularly, when using global variables

Therefore, the benefits of recursion pay-off for the extra overhead involved in terms of time and space required.

2.12 Function Overloading:

Function overloading, also known as method overloading, is a feature in C + + that allows creation of several methods with the same name but with different parameters. For example, print(), print(int), and print("He110") are overloaded methods. While calling print(), no arguments are passed to the function; however, when calling print (int) and print("Hello"), an integer and a string arguments are passed to the called function.

Note Function overloading allows one function to perform different tasks.

Function overloading is a type of polymorphism. Basically, there are two types of polymor-phism-compile time (or static) polymorphism and run-time (or dynamic) polymorphism. Function

overloading falls in the category of static polymorphism which calls a function using the best match technique or overload resolution.

According to this technique, the compiler compares the arguments in terms of number and type used in function call with the parameters specified in function definition to determine the most appropriate definition to use.

2.12.1 Matching Function Calls with Overloaded Functions

When an overloaded function is called, one of the following cases occurs:
Case 1: A direct match is found, and there is no confusion in calling the appropriate overloaded function.
Case 2: If a match is not found, a linker error will be generated. However, if a direct match is not found, then, at first, the compiler will try to find a match through the type conversion or type casting. For example, char, unsigned char, and short are promoted to an int; unsigned short can be promoted to int or unsigned int; float is promoted to double; and enum is promoted to int. Hence, the following function call will be matched with the given declaration.

```
void print(int); // Function declaration
print('R'); // Function call
```

Note

To find a suitable match, C++ compiler also tries user-defined conversions, which we will discuss later in this book. If nothing is possible, then a no match, linker error is generated.

Case 3: If an ambiguous match is found, that is, when the arguments match more than one overloaded function, a compiler error will be generated. This usually happens because all standard conversions are treated equal. For example, consider the function declarations and function call given here.

Example 2.8: To demonstrate the ambiguity in function call

```
using namespace std;
#include<iostream>
void print(int n){ cout<<n;}
void print(char c){ cout<<c; }
void print(float f) { cout<<f;}
main()
{ print(5); //Function call
   print((float)98.7);
   print('R');
}
OUTPUT
Error
```

Explanation: When you execute this code, you will get a compile time error as there two function declarations are valid for print(5); the function call can invoke either int or float version because all standard conversions are treated equal. Therefore, 5 can be treated as an int as well as a float.

Note All floating literals are treated as double unless they have the ' *f* ' suffix.

2.12.2 Key Points about Function Overloading

- In function overloading, there are multiple definitions for the same function name in the same scope.
- The definitions of these functions vary according to the types and/or the number of arguments in the argument list.
- Data type of the return value is not considered while writing overloaded functions because the appropriate function is called at the compile time while the return value will be obtained only when the function is called and executed.

Example 2.9: To compute the volume of different shapes using function overloading concept

```
using namespace std;
#include<iostream>
int volume(int side)
{ return side*side*side; // cube
}
float volume(float radius, float height)
{ return 3.14+radius*radius*height; //cylinder
}
long int volume(int length, int breadth, int height)
{ return length*breadth*height; //cuboid
}
main()
```

```
{ int s,l,b,height;
   float r, h;
   cout<<"\n Enter the side of the cube : ";
   cin>>s;
   cout<<"\n volume of cube with side "<<s<<" = "<<volume(s);
   cout<<"\n \n\n Enter the radius and height of the cylinder : ";
   cin>>r>>h;
   cout<<"\n Volume of cylinder with radius "<<r<<" and height "<<h<<" = "<<volume(r,h);
   cout<<"\n Enter the length, breadth and height of the cuboid : ";
   cin>>l>>b>>height;
   cout<<"\n Volume of cuboid with length "<<l<<" breadth "<<b<<" and height
   "<<height<<" = "<<volume(l,b,height);
}
```

OUTPUT

```
Enter the side of the cube : }
Volume of cube with side 3=27
Enter the radius and height of the cylinder : 3 4
Volume of cylinder with radius 3 and height 4=39.13
Enter the length, breadth and height of the cuboid: : 3 4";
volume of cuboid with length 2 breadth 3 and height 4=24
```

Note Function overloading is a form of compile time or static polymorphism (to be discussed in detail in Inheritance).

Program 2.10: Write a program to display values of different data types using compile time polymorphism.

```
using namespace std;
#include<iostream>
void print(float num)
{ cout<<"\n The number is: "<<num;
}
void print(char str[])
```

```
{ cout<<"\n The string is : "<<str;
}
main()
{
   print(9);
   print(3.78);
   print("Hello");
}
OUTPUT
The number is : }
The number is : 3.78
The string is : Hello
```

Program 2.11: Write a program to add two values of different data types using static polymorphism.

```
#includesstring.h>
using namespace std;
#include<iostream>
int add(int a, int b )
{ return a + b;
}
double add(double a, float b )
{ return a + b;
}
void add(string str 1 , string str2)
{
cout <<" \ n The concatenated string is : "<<(str1 + str2);
}
main()
{
cout<<" \ n5 + 7 = " << add(5,7);
cout<<"\n 123.678 + 25.97 = "<<add(123.678, 25.97);
add("Oxford", "University");
}
```

```
OUTPUT
5 + 7 = 12
123.678 + 25.97 = 149.647
The concatenated string is: OxfordUniversity
```

Program 2.12: Write a program that finds the absolute value of a number.

```
#includesstring.h>
using namespace std;
int abs(int n )
{ return n > θ ? n: −n;
}
double abs(double n )
{ return n > 0 ? n : −n;
}
main()
§ cout<<"\n Absolute value of 1234 = "<<abs(-1234);
cout<<" \ n Absolute value of 1234.5678 = "<<abs(1234.5678);
}
```

```
OUTPUT
Absolute value of 1234=1234
Absolute value of 1234.5678=1234.5678
```

2.12.3 Functions that Cannot be Overloaded

- Functions that differ only in the return type. For example, the program given here will give compile time error.

```
using namespace std;
#include<iostream>
int my_func() {return 1; }
char my_func() { return 'E'; }
main()
```

```
{ int num = my_func();
   char c = my_func();
}
```

- Parameter declarations that differ only in a pointer * versus an array [] are equivalent. A program with the following declarations will give a compilation error as both declarations are equivalent. The reason why they are equivalent will be clear in the chapter on Pointers.

```
int my_func(int *ptr);
int my_func(int ptr[]);
```

- If parameters in two functions differ only in the presence or absence of const and/or volatile, then those functions are considered to be equivalent. For example, the program having following declarations will not compile.

```
int my_func(int n);
int my_func(const int n;
```

- Using typedef does not introduce a new type, therefore, the following two function declarations are equivalent and this cannot be overloaded. (Use of typedef will be studied in chapter on Structures).

```
typedef int integer;
int my_func(int n);
int my_func(integer n;
```

- Function declarations that differ only in their default arguments are equivalent. Therefore, the program having function declarations given below will not compile.

```
int my_func(int n);
int my_func(int n=10);
```

- Function declarations that differ only in a reference parameter and a normal parameter cannot be overloaded. Therefore, the program having function declarations given here will not compile.

```
int my_func(int n);
int my_func(int &n);
```

2.13. Specifying a Class:

A class is the basic mechanism to provide data encapsulation. Data encapsulation is an important feature of object-oriented programming paradigm. It binds data and member functions in a single entity in such a way that data can be manipulated only through the functions defined in that class. When defining a class, we are actually creating a new user-defined data type which will be treated in the same way as other built-in data types.

The process of specifying a class consists of two steps-class declaration and function definitions (Fig. 2.5). While class declaration specifies the type and scope of its

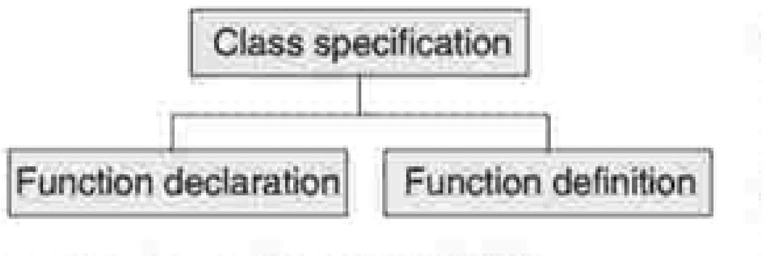
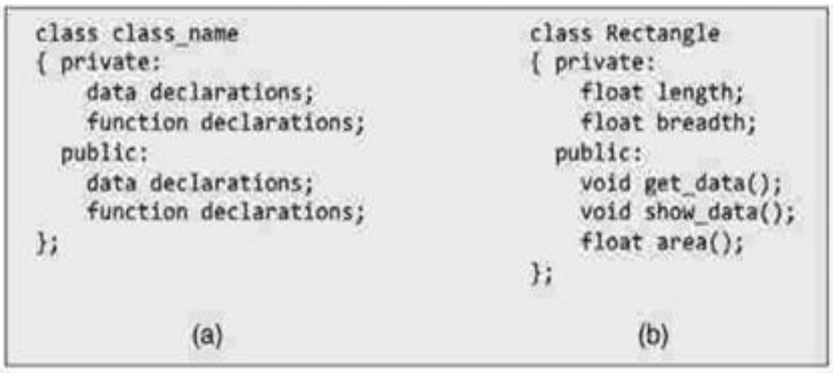


Figure 2.5. Class specification

members-private, public, or protected, function definition specifies how functions perform their intended task.

2.13.1 Class Declaration:

The syntax for class declaration is given in Fig. 2.6(a) and a sample class declaration is shown in Fig. 2.6(b).



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

(a)

```
class Rectangle
{ private:
    float length;
    float breadth;
  public:
    void get_data();
    void show_data();
    float area();
};
```

(b)

Figure 2.6 (a) Syntax of class declaration (b) Sample class

The keyword class denotes that the class name that follows is user-defined data type. The body of the class is enclosed within curly braces and terminated with a semicolon, as in structures. Data members and functions are declared within the body of the class. These data and functions form the members of the class.

Data and functions are grouped under two sections-private and public. They are also called visibility labels or access specifiers.

Private

All data members and member functions declared private can be accessed only from within the class. They are strictly not accessible by any entity-function or class-outside the class in which they have been declared. In C++, data hiding is implemented through the private visibility label.

Public

Data and functions that are public can be accessed from outside the class.

By default, members of the class, both data and function, are private. If any visibility label is missing, they are automatically treated as private members.

Note

A class having all its members as private is completely hidden from the outside world and does not serve any purpose**.**

Figure 2.7. shows that private members of a class can be accessed from outside the class only through the public member functions. Moreover, public data and public member functions can be accessed from outside the class.

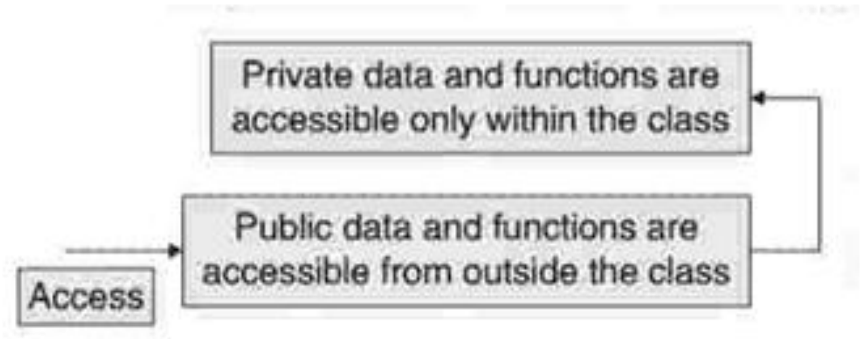

Private data and functions are accessible only within the class
Public data and functions are accessible from outside the class
Access

Figure 2.7 Accessing members of a class

2.14. Function Definition:

Member functions can be defined either inside the class or outside the class as shown in Fig. 2.4. However, wherever they may be defined, they perform the same task.

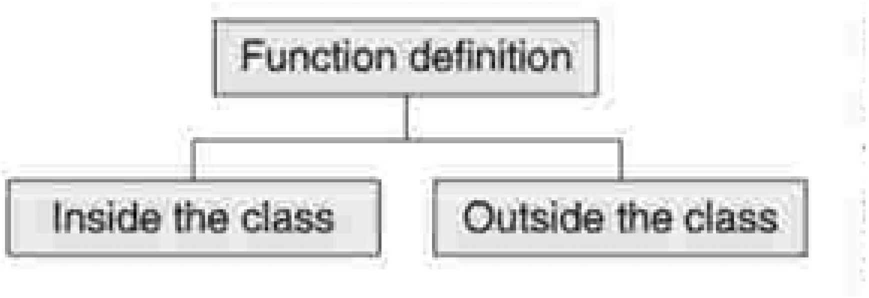


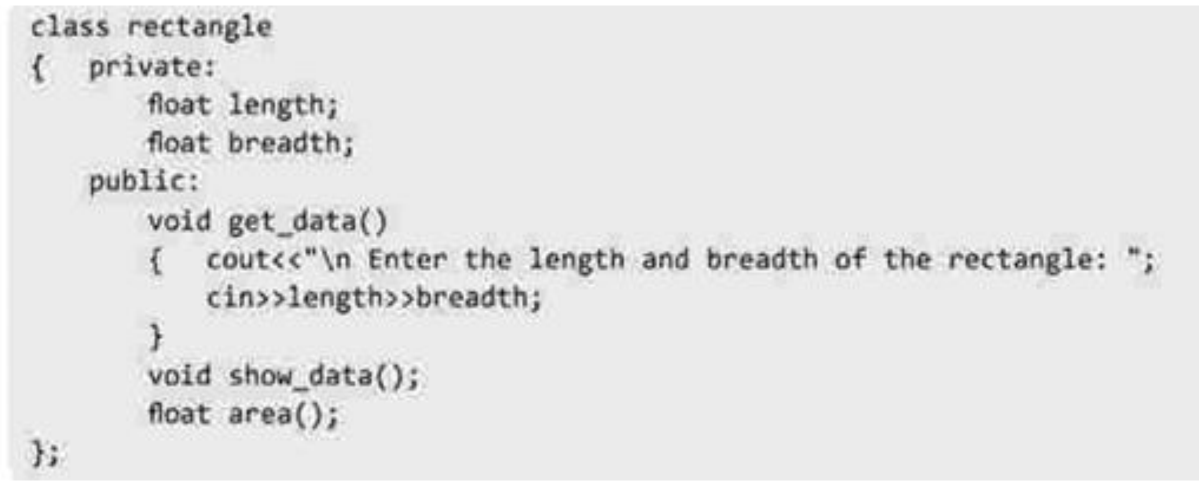
Figure 2.8. Function definition

2.14.1.Defining a Function Inside the Class

In this method, function declaration or prototype is replaced with function definition inside the class. Though it increases the execution speed of the program, it consumes more space. A function defined inside the class is treated as an inline function by default, provided they do not fall into the restricted cate- gory of inline functions.

Note: Functions that cannot be inline include functions having loop, switch, or go to statements, static variables, or recursive code.

As a good practice, you must define only small functions inside the class. Let us try to define the get_data() inside the class.



```
class rectangle
{   private:
        float length;
        float breadth;
    public:
        void get_data()
        {   cout<<"\n Enter the length and breadth of the rectangle: ";
            cin>>length>>breadth;
        }
        void show_data();
        float area();
};
```

In the code, get_data() is an inline member function of the class rectangle. This function is used to read the values of private data members of the class from the users.

2.14.2.Defining a Function Outside the Class

Member functions defined outside the class are defined in the same way as other normal functions. However, the only difference between a member function and a normal function is that a member function is specified using a membership identity label in the function header. The identity label informs the compiler about the class to which the function belongs to.

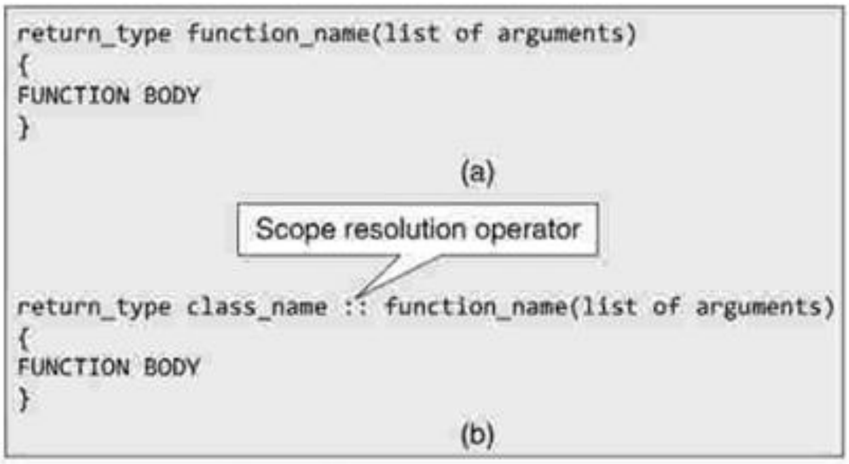
Figure 2.9.shows the general form of a normal function and that of a class member function defined outside the class.

class_name:: and function_name tell the compiler that scope of the function is restricted to the class_name. Hence, the name of the :: operator is scope resolution operator. The scope resolution operator identifies and specifies the context to which an identifier refers. The importance of the :: is even more prominent in the following cases:

• When different classes in the same program have functions with the same name. In this case, the :: operator will tell the compiler which function belongs to which class.

• To restrict the non-member functions of the class to use its private members.

• Allows a member function of the class to call another member function directly without using the dot operator.



```
return_type function_name(list of arguments)
{
FUNCTION BODY
}
```

```
return_type class_name :: function_name(list of arguments)
{
FUNCTION BODY
}
```

Figure 2.9 (a) Normal function definition

(b) Member function definition outside the class

Following is an example for defining the function (area()) outside the class:

float Rectangle :: area(void**)**

{ return length breadth;

}

Defining a member function outside the class reduces the execution speed but takes more space.

2.15. Making a Member Function Inline:

We have seen that member functions can be either defined inside the class or outside the class. While defining a function inside the class makes the code shorter, as a good programming practice, member functions must be defined outside the class. This will not only enhance clarity but will also result in separating the details of implementation from the class definition which is a basic objective concept of object-oriented programming.

However, programmers do not want to compromise speed for code clarity. We have seen that all functions defined inside the class are treated as inline by default. Therefore, the overhead of function call is completely avoided and the code executes faster. However, to gain the same benefits while separating the implementation details, C++ enables programmers to make a member function defined outside the class an inline. This can be done by using the keyword inline in the header line of function definition. For example, consider the code given here which makes show_data(), a func- tion defined outside the class as an inline function.

```
class Rectangle
{   private:
    float length;
    float breadth;
    public:
        void get_data()     {      cin>>length>>breadth;      }
        void show_data();
};
inline void Rectangle :: show_data(void)
{       cout<<"Length = "<<length<<" Breadth = "<<breadth;
}
```

2. 16. Nested Member Functions:

We have learnt that only an object of a class can call any member function of that class using the dot operator. However, when we use nested functions-a function inside another function-we do not need the object name. A member function can directly call another function of that class. For example,

```
class Rectangle
{   private:
    float length;
    float breadth;
    public:
    void get_data()     {      cin>>length>>breadth;      }
    void show_data()
    {   cout<<"Length = "<<length<<" Breadth = "<<breadth<<" and Area = "<<area();
    }
        int area()     {      return length * breadth;      }
};
```

In this example, data members of the class are directly accessible by the member functions. Member functions do not need the dot operator to use the data members. Similarly, when we call a member function from within another member function, we do not need the name of the object or the dot operator. The member function can be directly called by using its name along with arguments, if any.

2.17. Memory Allocation for Class and Objects:

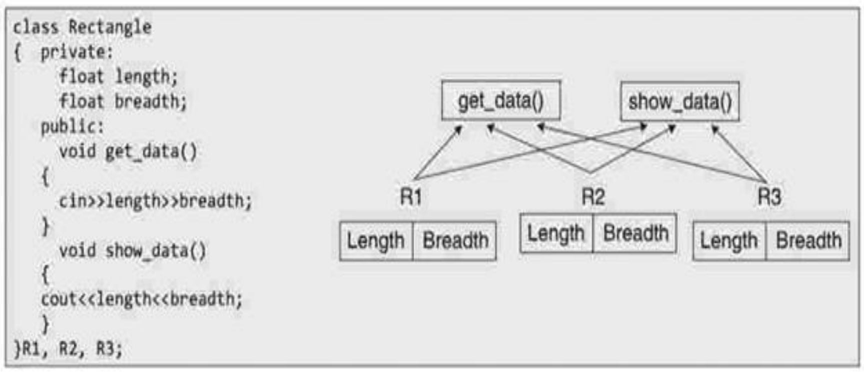
Before learning how memory is allocated to objects, let us recapitulate some important points about objects.

- An object is an instance of a class.
- Every object is identifiable.
- Objects communicate with classes through passing messages known as function calls.
- Every object is defined by a state or value of variables.
- The state of the object can be changed by one or more operation performed by member function(s) of that class.
- Memory for a class is allocated only when one or more objects of that class is created.

The last point is only partly true because when a class is specified, memory for member functions of class is allocated. When one or more objects are created, separate chunks of memory for storing data members are allocated to each object as shown in Fig. 9.6. This type of memory allocation is quite justified because while each object has a different set of values for its data members, they execute the same piece of function code. Hence, it is better to store function code in one place and let all the objects of that class share the function code.

Note

Memory for member functions is allocated only once when the class is defined. Therefore, a single copy of member function is shared among all objects.



```
class Rectangle
{  private:
      float length;
      float breadth;
   public:
      void get_data()
   {
      cin>>length>>breadth;
   }
      void show_data()
   {
   cout<<length<<breadth;
   }
}R1, R2, R3;
```

Figure 2.10. Objects sharing functions but having their own data

2.18. Static Member Functions:

C++ not only allows data members of a class to be static but also supports static member functions. Any static member-whether data or function is allocated to memory only once and is not a part of any object but is shared by all the objects of that class. Additionally, a static member function has the following features:

- It can access only the static members-data and/or functions-declared in the same class.

- It cannot access non-static members because they belong to an object but static functions have no object to work with.
- Since it is not a part of any object, it is called using the class name and the scope resolution operator.
- As static member functions are not attached to an object, the this pointer does not work on them.
- A static member function cannot be declared as virtual function.
- A static member function cannot be declared with const, volatile type qualifiers.

Don't worry if last three points are not clear. You will understand these concepts when we will deal with them in detail later in this book. In this chapter, we take a small example that makes use of static member function and is usually implemented in solutions designed for real-world problems. The function is used to automatically generate the next ID.

Note Static and non-static member functions in the same class with same names and same number and types of arguments are not allowed in C++. Therefore, static void print (float); and void print(float) is illegal.

Example 2.10. Static member function

```
using namespace std;
#include<iostream>
class ID_Generator
{   private:
        static int next_ID;          // next_ID is a static data member
    public:
```

```
        static int GenNextID()       // GenNextID is a static member function
    {       cout<<"\n NEXT ID = "<<next_ID++;      }
};
int ID_Generator :: next_ID = 1;
main()
{   for(int i=0;i<5;i++)
    ID_Generator :: GenNextID();
}
```

```
OUTPUT
NEXT ID = 1
NEXT ID = 2
NEXT ID = 3
NEXT ID = 4
NEXT ID = 5
```

Explanation:

The program is used to automatically generate the next, unique identification number. Every time the static function is called, it increments the value of static variable by 1. In this program, we have not made any object of the class. This is because our class has only one member function and that is static. Static member functions are not invoked in conjunction with objects. They are called using the class name and the scope resolution operator.

2.19. Array of Objects:

Just as we have arrays of basic data types, C++ allows programmers to create arrays of user-defined data types as well. Therefore, it is possible to make an array of class data type or array of objects. They are usually defined to handle a group of objects that reside in a contiguous memory location. For example, there is not one student in a class, rather a class has n number of students. Therefore, we can declare an array of class student by simple writing

student s[20]; // assuming there are 20 students in a class.

When this statement gets executed, the compiler will set aside memory for storing details of 20 stu- dents. This means that in addition to the space required by member functions, 20* sizeof(student) bytes of consecutive memory locations will be reserved for objects at the compile time.

An array of objects uses the same concepts as that of arrays of other data types. An individual object of the array of objects is referenced by using an index, and the particular member is accessed using the dot operator. Therefore, if we write,

s[i].get_data()

then the statement when executed will take the details of the ith student.

2.20. Objects as Function Arguments

Similar to variables, objects can also be passed as arguments to functions. Similar to basic data type arguments, objects can also be passed to functions in the following three ways:

Pass-by-value In this technique, a copy of the actual object is created and passed to the called function. Therefore, the actual (object) and the formal (copy of object) arguments are stored at dif- ferent memory locations. This means that any changes made in formal object will not be reflected in the actual object.

Pass-by-reference In this method, the address of the object is implicitly passed to the called

function

Pass-by-address In this technique, the address of the object is explicitly passed to the called function.

Since both pass by reference and pass-by-address techniques send the address of the actual object to the called function, any changes made in the object are reflected to actual object. Moreover, these two techniques must be preferred as they prevent duplication of object and hence, reduce memory space requirements of a program. Example 9.5 demonstrates how an object is passed as an argument using the three techniques discussed.

Example 2.11: Passing object as an argument

```
using namespace std;
#include<iostream>
class myClass
{   private:
        int num;
    public:
        void get_data()
        {   cout<<"\n Enter a number : ";
            cin>>num;
        }
        void set_data(myClass o2)     // function accepts object
```

```
        {   num = o2.num;
        }
        // function accepts address of object
        void set_data(int a, myClass &o2)
        {   num = o2.num;
        }
        void set_data(myClass *o2) // function accepts pointer
        {   num = o2->num;
        }
        void show_data()
        {   cout<<"\n Num = "<<num;
        }
};
main ()
{   myClass o1, o2, o3, o4;
    int a=0;
    o2.get_data();
    o1.set_data(o2);                //call by value
    o3.set_data(&o2);           //call by pointer
    o4.set_data(a, o2);             //call by reference
    o1.show_data();
    o2.show_data();
    o3.show_data();
}

OUTPUT
Enter a number: 7
NUM = 7
NUM = 7
NUM = 7
```

Explanation:

In this program, only the value of object o2 has been entered by the user. Values of other objects-01, 03, and 04-have been set through the o2 object using call-by-value, call- by-pointer, and call-by-reference techniques, respectively. When we called the function through call-by-reference, we had sent an extra argument that was absolutely not required. However, the argument was just passed to have some small difference in the function header so that the compiler does not generate any error. As per the rules of function overloading, reference variables are not over- loaded. In simple terms, void set_data(myClass obj); is exactly same as void set_data(myClass. &obj).

Note that as two functions with the same header are not allowed, just a small difference was made in the function header to avoid any error.

2.21.Friend Function:

Till now, we have studied that since data hiding is a fundamental concept of object-oriented programming, a non-member function cannot access an object's private or protected data. However, at times, this restriction makes the code too long and complex. So, to overcome this problem, C++ supports the use of friend functions and friend classes.

A friend function of a class is a non-member function of the class that can access its private and protected members (keyword protected will be discussed in Chapter 12 on Inheritance). Just as we allow our friends to use our personal belongings, a friend function of a class is allowed to access the members of that class. Friend functions help implement data encapsulation in C++. They keep data of a class private from all external functions and classes except from those that are explicitly declared as friend and thus permitted to access.

To declare an external function as a friend of the class, you must include function prototype in the class definition. The prototype must be preceded with keyword friend.

The template to use a friend function can be given as follows:

```
class class_name
friend return_type
function_name(list of
arguments);

return_type function_name(list of arguments)

    f

    }
```

Following are some important points related to friend function:

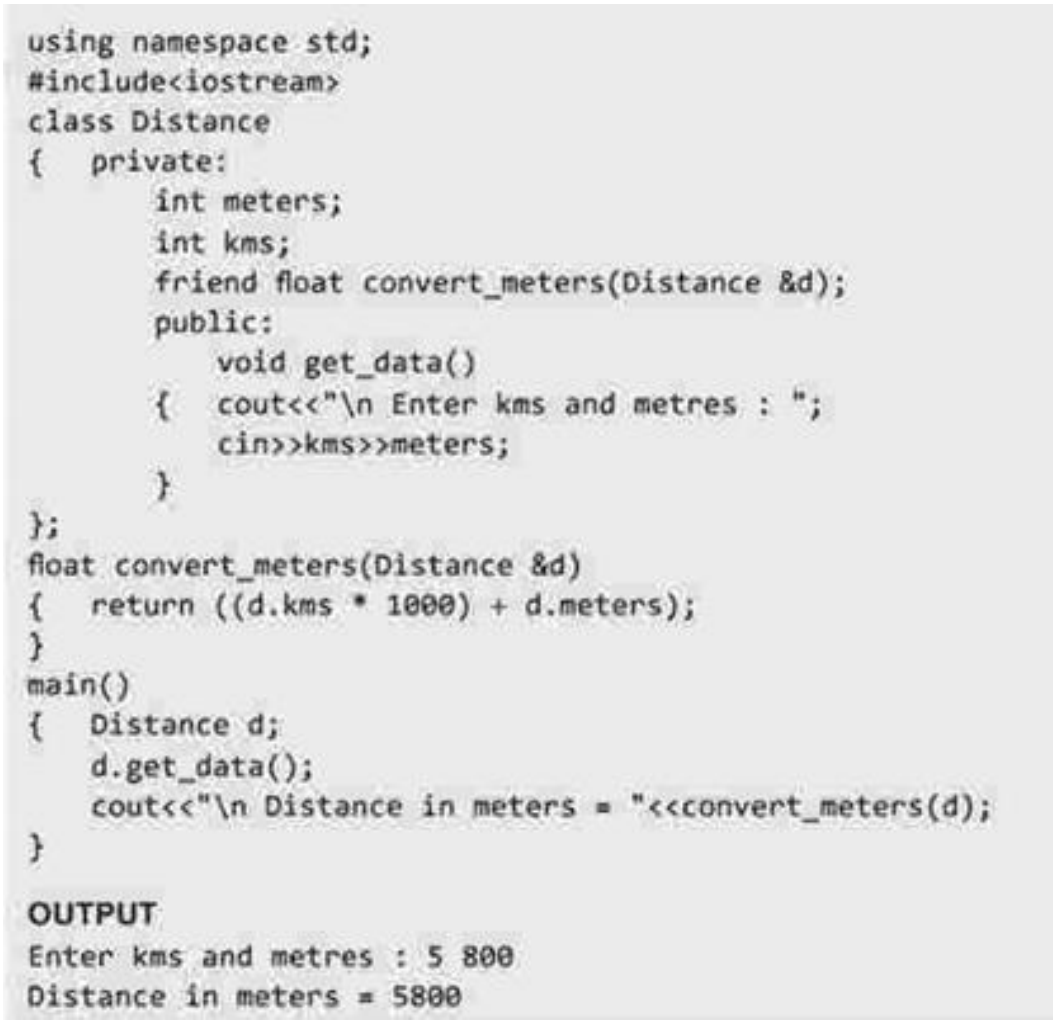
- Friend function is a normal external function that is given special access privileges.
- A friend function is defined outside that class' scope. Therefore, they cannot be called using theor->' operator. These operators are used only when they belong to some class.
- While the prototype for friend function is included in the class definition, it is not considered to be a member function of that class.
- The friend declaration can be placed either in the private or in the public section.
- A friend function of the class can be member and friend of some other class.
- A function friend of one class can be friend of another class.

- Since friend functions are non-members of the class, they do not require this pointer.
- The keyword friend is placed only in the function declaration and not in the function definition.
- A function can be declared as friend in any number of classes.
- A friend function can access the class's members directly using the object name and dot operator followed by the specific member.

Friend functions are frequently used to perform operations that are conducted on two different classes and thus require access to private or protected members of both classes.

Note While any non-member function of a class can **access** its public members, only the friend functions can access its private and protected members.

Example 2.12: Friend function



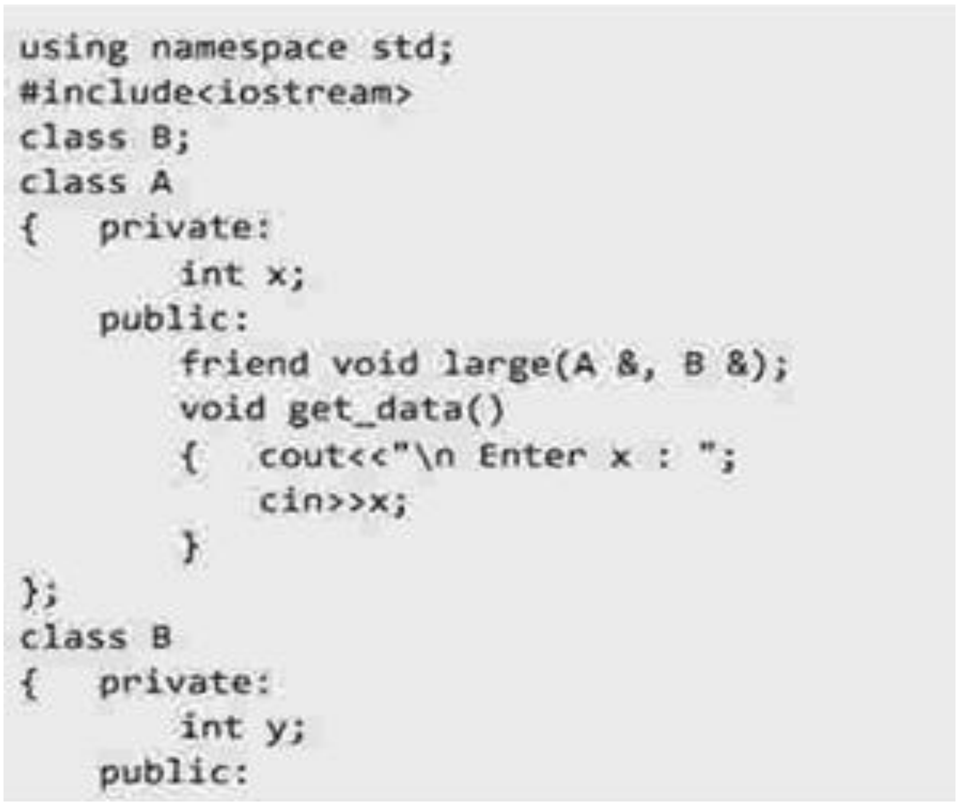
```
using namespace std;
#include<iostream>
class Distance
{   private:
        int meters;
        int kms;
        friend float convert_meters(Distance &d);
        public:
            void get_data()
        {   cout<<"\n Enter kms and metres : ";
            cin>>kms>>meters;
        }
};
float convert_meters(Distance &d)
{   return ((d.kms * 1000) + d.meters);
}
main()
{   Distance d;
    d.get_data();
    cout<<"\n Distance in meters = "<<convert_meters(d);
}
```

OUTPUT

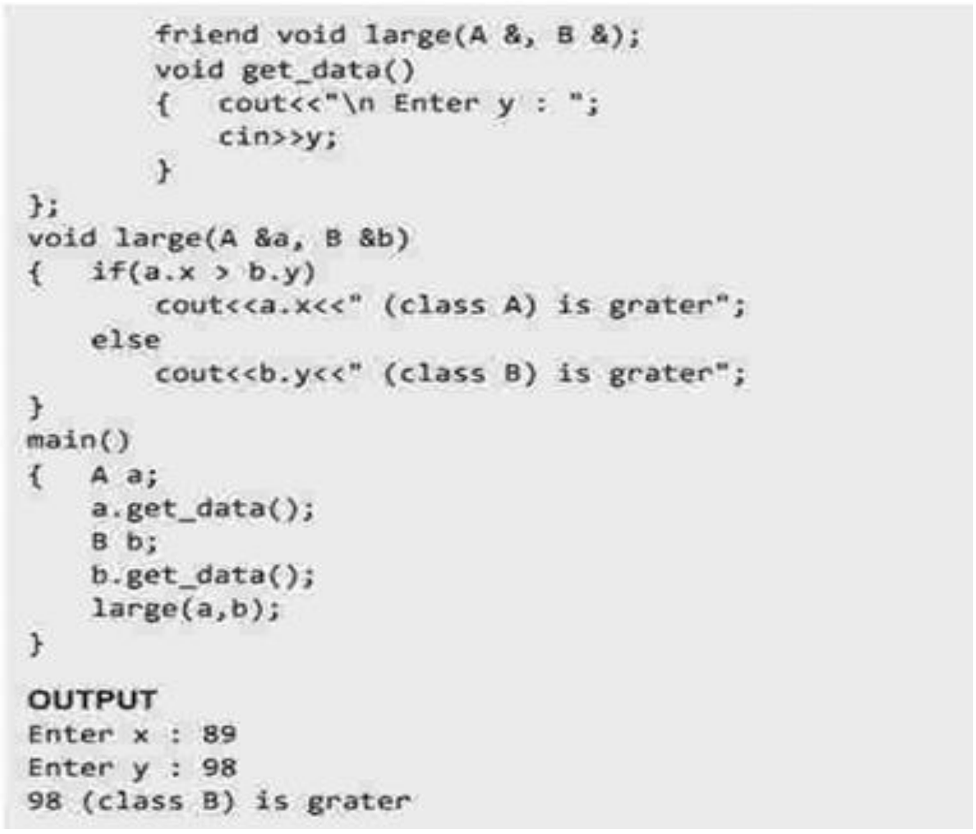
```
Enter kms and metres : 5 800
Distance in meters = 5800
```

Explanation:

In the program, the function convert meters () is an external function friend to class Distance. Now, consider another program in which a function is friend to two different classes. The program accepts objects of two classes and finds which one of them is larger.



```
using namespace std;
#include<iostream>
class B;
class A
{   private:
        int x;
    public:
        friend void large(A &, B &);
        void get_data()
        {   cout<<"\n Enter x : ";
            cin>>x;
        }
};
class B
{   private:
        int y;
    public:
```



```
        friend void large(A &, B &);
        void get_data()
        {   cout<<"\n Enter y : ";
            cin>>y;
        }
};
void large(A &a, B &b)
{   if(a.x > b.y)
        cout<<a.x<<" (class A) is grater";
    else
        cout<<b.y<<" (class B) is grater";
}
main()
{   A a;
    a.get_data();
    B b;
    b.get_data();
    large(a,b);
}
```

OUTPUT

```
Enter x : 89
Enter y : 98
98 (class B) is grater
```

Explanation:

In the program, note that we have written class 8; and then defined class B later in the program. The statement class B; is called forward declaration and is used to tell the com- piler that such a class will be defined later in the program. This is important because in class A, we have a friend function which accepts objects of another class that has not yet been defined. For the compiler to know B, we have to instruct the compiler that B is a class and declare it before it is defined.

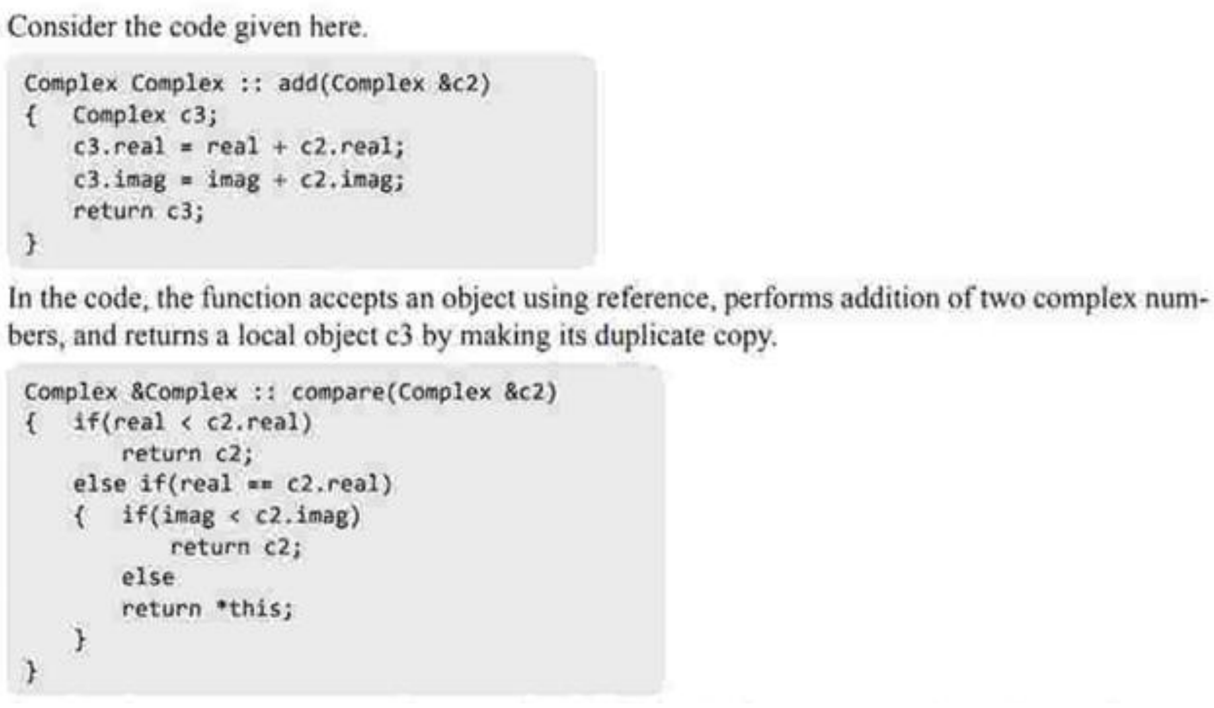
2.22.Returning Objects:

C++ not only allows users to pass objects to functions as arguments but also allows functions to return objects. The syntax is the same as that for a normal variable. You can return an object using the following three methods:

Returning by value which makes a duplicate copy of the local object.

Returning by using a reference to the object. In this way, the address of the object is passed implicitly

Returning by using the 'this pointer which explicitly sends the address of the object to the calling function,


Consider the code given here.

```
Complex Complex :: add(Complex &c2)
{   Complex c3;
    c3.real = real + c2.real;
    c3.imag = imag + c2.imag;
    return c3;
}
```

In the code, the function accepts an object using reference, performs addition of two complex numbers, and returns a local object c3 by making its duplicate copy.

```
Complex &Complex :: compare(Complex &c2)
{   if(real < c2.real)
        return c2;
    else if(real == c2.real)
    {   if(imag < c2.imag)
            return c2;
        else
        return *this;
    }
}
```

The function compares two complex numbers. In this code, there are two points to be noted:

• if(real < c2.real), then c2 is returned using reference. Check the function header;

the func- tion returns an object using an address either implicitly or explicitly. Since we are writing return c2, it means that object c2 is being returned using a reference.

• If object cz is greater than c1 (the object that invoked the compare function), then c2 is returned to the caller using reference in which the address of the object was implicitly passed. However, if c1 is greater, then it is being returned using a special pointer called this pointer.

Exercises:

1.What are nested member functions? Explain with the help of an example.

2.Illustrate with an example how a function defined outside the class can be made. inline.

3. When memory is allocated for class data members and member functions?

4. What are static data members? Why are they needed?

5. Write a short note on static member functions.

6. What are static objects? Write a short code to explain their concept.

7. What are friend functions? Are they a threat to data hiding? How do they help in data encapsulation?

8. Illustrate the significance of bit fields with an example.

9. Explain the concept of dynamic memory allocation for array of objects with an example.

10. Differentiate between a structure and a class in C++.

Unit III

Introduction, Constructors, Parameterized constructors, Multiple constructors in a class, Constructors with default arguments, Dynamic initialization of objects, Copy constructor, Constructing Two-Dimensional arrays, constant objects, Destructors.

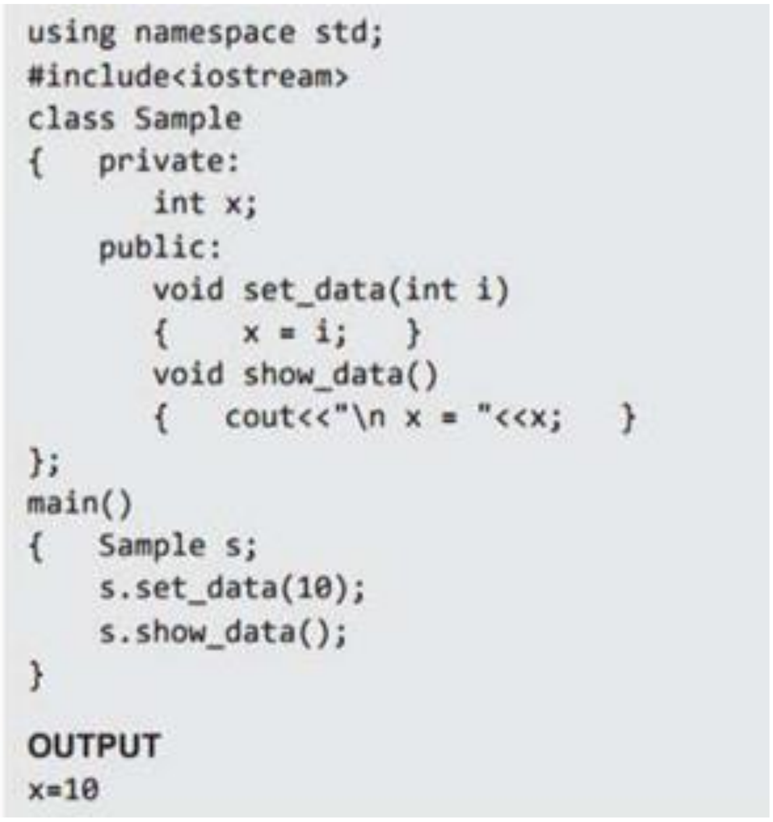
Chapter 3: Sections 3.1-3.7.

3.1. Introduction:

We have learnt that to initialize the private members of a class, we used a function known an set_data (). The values for private members were passed as arguments to the function as shown in the code given here.

Example 1:

Initializing private members of class



```
using namespace std;
#include<iostream>
class Sample
{   private:
        int x;
    public:
        void set_data(int i)
        {     x = i;    }
        void show_data()
        {    cout<<"\n x = "<<x;     }
};
main()
{   Sample s;
    s.set_data(10);
    s.show_data();
}

OUTPUT
x=10
```

Explanation:

In the program, observe carefully that we are first creating an object of the class. We are explicitly calling a class member function to initialize its data members. Though there is no problem in this code, a basic concept of object-oriented is not followed here. We know that C++ treats all its user-defined data types at par with its built-in data types. Therefore, when we write, int $x = 10$; to initialize an integer variable at the time of its declaration, then it is normal to question as to why cannot write a code that initializes the object's data members while it is being created or declared.

Therefore, to resolve this dissimilar behaviour, $\mathrm{C}++$ provides a member function called the constructor which enables an object to initialize itself when it is created. Like constructor, there is another member function called the destructor which de-initializes an object. We will read more about this function in the following sections.

3.2. Constructor:

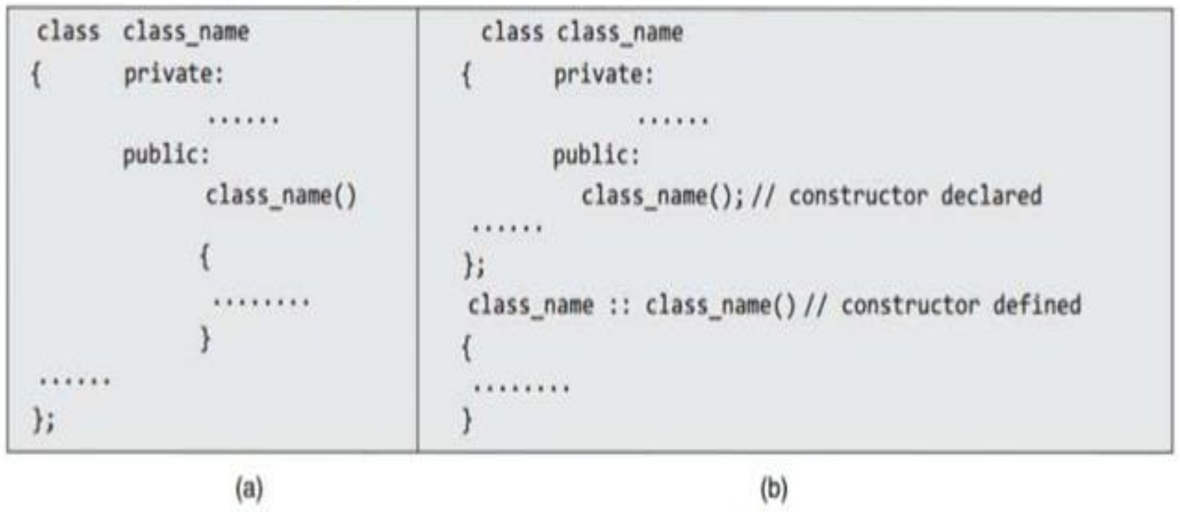
A constructor is a special member function of a class which is automatically invoked at the time of creation of an object to initialize or construct the values of data members of the object. However, some special points to note about a constructor function are as follows.

- The name of the constructor is the same as that of the class to which it belongs.
- A constructor must be declared in the public section.
- It should not be explicitly called because a constructor is automatically invoked when an object of a class is created.
- A constructor can never return any value; therefore, unlike a normal function, a constructor does not have any return value (not even void).
- A constructor cannot be inherited and virtual. We will read about this aspect in Chapter 12 on Inheritance.
- The address of a constructor cannot be referred to in programs. Therefore, pointers and references do not work with constructors.
- A constructor cannot be declared as static, volatile, or const.
- Like a normal function, a constructor function can also be overloaded.
- Like a normal function, a constructor function can also have default arguments.

Note:

The C++ run time mechanism ensures that constructor is the first member function to be executed when an object of that class is created.

Like a normal class member function, a constructor can be either defined inside the class or outside the class. The syntax for declaring and defining a constructor inside the class can be given as shown in Fig. 3.1(a). Similarly, the syntax for declaring and defining a constructor outside the class can be given as shown in Fig. 3.1(b).



```
class  class_name
{       private:
              ......
        public:
              class_name()
              {
               ........
              }
......
};
```

(a)

```
class class_name
{       private:
              ......
        public:
          class_name(); // constructor declared
......
};
class_name :: class_name() // constructor defined
{
........
}
```

(b)

Figure 3.1 (a) Declaring and defining a constructor within a class, (b) Defining a constructor outside the class

3.3. Types of Constructors:

A constructor function can be classified as follows:

- Dummy constructor
- Copy constructor
- Default constructor
- Dynamic constructor
- Parameterized constructor

Figure 3.2 gives the classification of constructors.

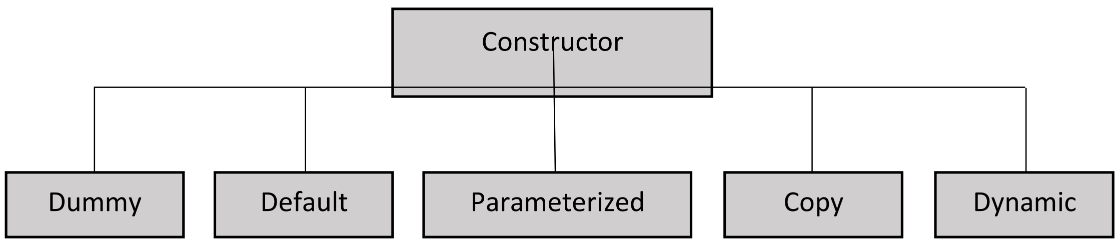
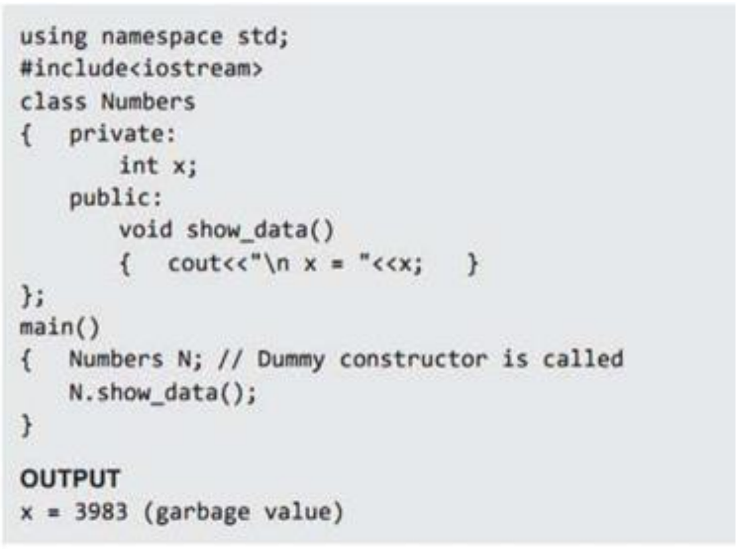


Figure 3.2 Types of constructor

3.3.1. Dummy Constructor (Do Nothing Constructor):

In Chapter 9, we had been writing programs without any constructor. In such cases, the C++ run time mechanism calls a dummy constructor which does not perform any action. Here, action means does not initialize any data member and thus, the variables acquire a garbage (irrelevant) value. Consider the program given here that makes use of a dummy constructor.

Example 3.2:



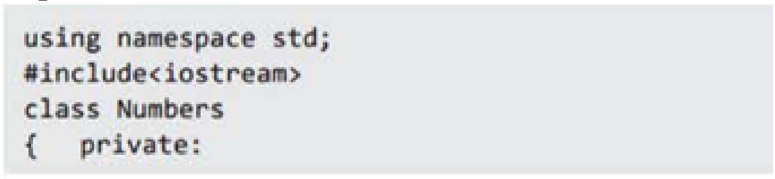
```
using namespace std;
#include<iostream>
class Numbers
{   private:
        int x;
    public:
        void show_data()
        {   cout<<"\n x = "<<x;    }
};
main()
{   Numbers N; // Dummy constructor is called
    N.show_data();
}
```

```
OUTPUT
x = 3983 (garbage value)
```

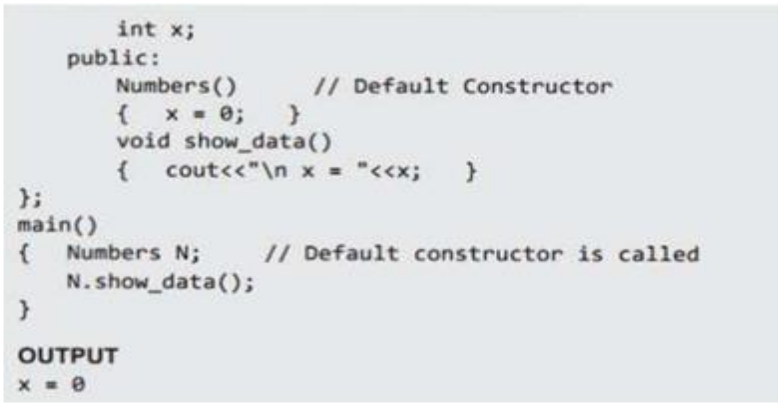
3.3.2. Default Constructor:

A constructor that does not take any argument is called a default constructor. The default constructor simply allocates storage for the data members of the object. It may even initialize the values of those data members. Let us rewrite the same program given earlier but with a default constructor.

Example 3.3:



```
using namespace std;
#include<iostream>
class Numbers
{   private:
```



```
        int x;
    public:
        Numbers()       // Default Constructor
        {   x = 0;    }
        void show_data()
        {   cout<<"\n x = "<<x;    }
};
main()
{   Numbers N;     // Default constructor is called
    N.show_data();
}
```

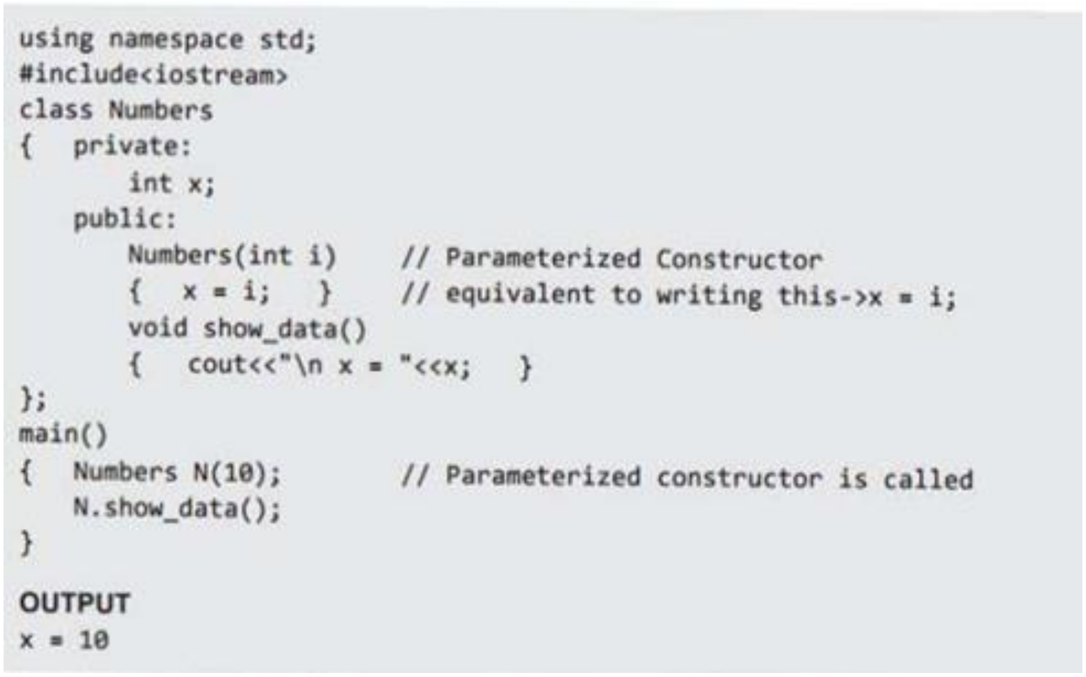
```
OUTPUT
x = 0
```

3.3.3. Parameterized Constructor

A constructor that accepts one or more parameters is called a parameterized constructor. The program code given here uses a parameterized constructor to initialize the data member of the class.

Example 3.4:

Initialize data member of the class through parameterized constructor



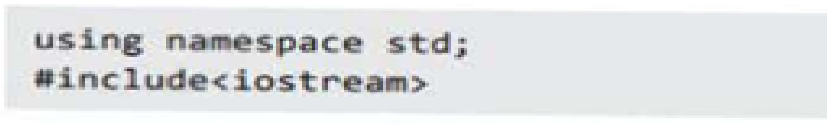
```
using namespace std;
#include<iostream>
class Numbers
{   private:
        int x;
    public:
        Numbers(int i)     // Parameterized Constructor
        {  x = i;   }      // equivalent to writing this->x = i;
        void show_data()
        {   cout<<"\n x = "<<x;   }
};
main()
{   Numbers N(10);         // Parameterized constructor is called
    N.show_data();
}

OUTPUT
x = 10
```

Programming Tip: You can call a constructor from main() since a constructor is like a member function declared in the public section.

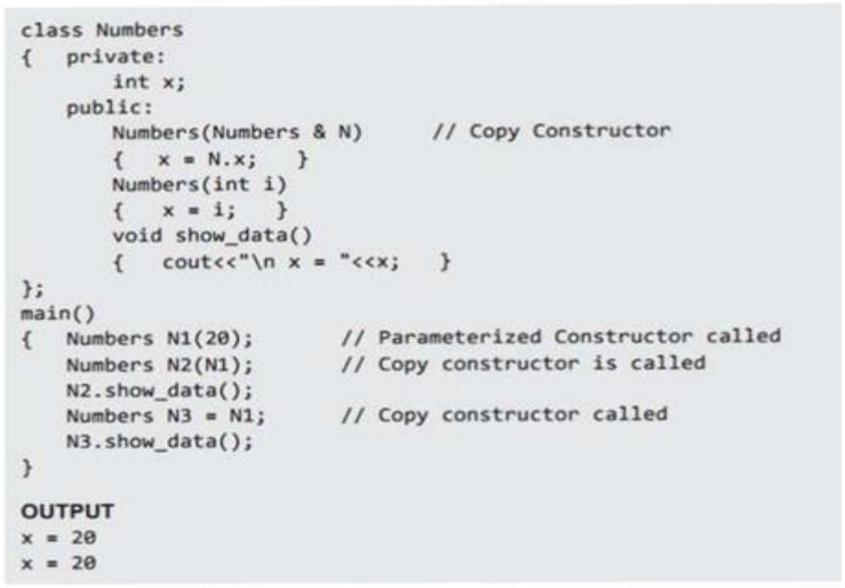
3.3.4. Copy Constructor:

A copy constructor takes an object of the class as an argument and copies data values of members of one object into the values of members of another object. Since it takes only one argument, it is also known as a one argument constructor. The primary use of a copy constructor is to create a new object from an existing one by initialization. For this, the copy constructor takes a reference to an object of the class as an argument.

Example 3.5:



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



```
class Numbers
{   private:
        int x;
    public:
        Numbers(Numbers & N)      // Copy Constructor
        {   x = N.x;    }
        Numbers(int i)
        {   x = i;    }
        void show_data()
        {   cout<<"\n x = "<<x;    }
};
main()
{   Numbers N1(20);        // Parameterized Constructor called
    Numbers N2(N1);        // Copy constructor is called
    N2.show_data();
    Numbers N3 = N1;       // Copy constructor called
    N3.show_data();
}

OUTPUT
x = 20
x = 20
```

Explanation:

In the program, the first object N1 invokes the parameterized constructor to initialize its data members. The objects N2 and N3 invoke the copy constructor. There are two ways of invoking the copy constructor. The first is invoked by writing Numbers N2 (**N1)**; and the second way is writing Numbers N3 = N1;. Both the statements perform the same task of initializing the data member of N2 and N3 with the value of data member of N1.

Besides these explicit calls to the copy constructor, the copy constructor is implicitly called when a function accepts an object as an argument using the call-by-value technique or returns an object by value. In both the cases, a copy of the object is made and passed to the destination function.

Note:

A copy constructor is called for **at the** time of invocation. If an assignment is made after the object has been created, then the assignment operator, and **not** the copy constructor, would work. In the program, if we write

Numbers N1(20), N2;

N1 N2**;** // here the assignment operator would **work**.

Why Do Copy **Constructors Take Objects by Reference and Not by Value?**

Copy constructors take objects by reference and not by value. As mentioned earlier, when an object is passed by value, the copy constructor is implicitly called to create a copy of the original argument. If the copy constructor had been designed to accept the object by value, then it would have resulted in an infinite recursion; the copy constructor would have called another copy constructor, which in turn, would have called another copy constructor, and so on, thereby consuming more time and space. To avoid such a situation, C++ mandates the copy constructor's parameter to be passed by reference.

Note

The parameter **to** the copy constructor **can be** declared **as** const if you want **the** original object's data members values should not be altered.

3.3.5 Dynamic Constructor

Dynamic constructors, as the name suggests, are those constructors in which memory for data members is allocated dynamically. Dynamic constructor enables the program to allocate the right amount of memory to data members of the object during execution. This is even more beneficial when the size of data members is not the same each time the program is executed. The memory allocated to the data members is released when the object is no longer required and when the object goes out of scope.

Example 3.6: Dynamic constructor

```
using namespace std;
#include<iostream>
class Array
{   private:
        int *arr;
        int n;
    public:
        Array()
        {   n = 0;      }
        Array(int);
        void show_data();
};
Array :: Array(int num)
{   n = num;
    arr = new int [n];   // memory allocated for array dynamically
    cout<<"\n Enter the elements : ";
    for(int i=0;i<n;i++)
        cin>>arr[i];
}
void Array :: show_data()
{   for(int i=0;i<n;i++)
        cout<<" "<<arr[i];
}
main()
{   int size;
    cout<<"\n Enter the size of the array : ";
    cin>>size;
    Array Arr(size);   //calls constructor and allocates memory
    Arr.show_data();
}
```

OUTPUT

```
Enter the size of the array : 5
Enter the elements : 1 2 3 4 5
1 2 3 4 5
```

3.4. Constructor With Default Arguments:

Like other functions, constructors can also have default arguments. When an object of a class is created, the C++ compiler calls the suitable constructor for initializing that object. Consider the program given here which makes use of a constructor with default arguments.

Programming Tip: A compiler error will be generated if you try to return any value from a constructor or a destructor. You can call a constructor from a destructor.

Example 1: Constructor with default arguments

```
using namespace std;
#include<iostream>
class Student
{private:
        int roll_no;
        int marks;
public:
        Student()
        {       roll_no=0;
                marks = 0;
        }
        student(int r, int m = 0)
        {       roll_no = r;
                marks = m;
}
        void show_data()
        {       cout<<"\n ROLL NO. = "<<roll_no;
                cout<<"\t MARKS = "<<marks;
        }
};
main()          // default constructor called
{       Student S1;
        S1.show_data();
        Student S2(3);
        S2.show_data();
        Student S3(05, 98);
        S3.show_data();
}
```

OUTPUT

```
ROLL NO. = 0            MARKS = 0
ROLL NO. = 3            MARKS = 0
ROLL NO. = 5            MARKS = 98
```

Explanation:

In the program, when the first object is created without any arguments, the default constructor would be called and data members will be initialized with zero. While creating the second object, one parameter is explicitly passed and the second argument is missing. Therefore, the default value will be used to initialize the missing data member. However, during the third object, both parameters are explicitly passed. Therefore, the default value will be overridden and the parameter's value will be used to initialize the data member.

Note: The missing argument(s) must always be the trailing ones.

[Programming Tip:

C++ allows users to place default arguments in the definition of a constructor rather than in the declaration.]

Now consider the same code again with a slight modification.

```
using namespace std;
#include<iostream>
class Student
{       private:
                int roll_no;
                int marks;
public:
        Student()               //constructor 1

        {       roll_no = 0;
                marks = 0;
        }
```

```
          Student(int r =0, int m = 0)    // constructor 2
          {         roll_no = r;
                    marks = m;
          }
          void show_data()
          {         cout<<"\n ROLL NO. = "<<roll_no;
cout<<"\t MARKS= "<<marks;
          }
};
main()
{         Student S1;              // which constructor to call? ERROR
          S1.show_data();
          Student S2(03);
          S2.show_data();
          Student S3(05, 98);
          S3.show_data();
}
```

Explanation:

If you compile this program, you will get an error. In the program, there are two constructors: one is the default constructor and the second is with both default arguments. When this program is compiled, the compiler gets confused which one of the two constructors to call as both of them satisfy the condition to be called during creation of object S1. Hence, an error will be shown. The moment you remove any one of the constructor the program will compile successfully.

Note:

A constructor that has all default arguments is similar to a default (no - argument) constructor.

3.5. Constructor Overloading:

Like normal functions, constructors can also be overloaded. When a class has multiple constructors, they are called overloaded constructors. Some important features of overloaded constructors are as follows:

- They have the same name; the names of all the constructors is the name of the class.

- Overloaded constructors differ in their signature with respect to the number and sequence of arguments passed.

When an object of the class is created, the specific constructor is called. Consider the program code given here which uses the concept of overloaded constructors.

Example 1:

Multiple constructors in a program

```
using namespace std;
#include<iostream>
#include<string.h>
class Person
{       private;
                int age;
                char first_name [10];
                char middle_name[10];
                char last_name[10];
        public:
                Person()                //Constructor 1 – default constructor
                {       age = -1;
                        strcpy(first_name, "\0");
                        strcpy(middle_name, "\0");
                        strcpy(last_name, "\0");
                }
                Person(int a)           //Constructor 2 - constructor with one argument

                {       age = a;
                        strcpy(first_name, "\e");
                        strcpy(middle_name, "\0");
                        strcpy(last_name, "\0");
                }
                Person(int a, char *fn) //Constructor 3 - constructor with two arguments
                {       age = a;
```

```
                    strcpy(first_name, fn);
                    strcpy(middle_name, "\0");
                    strcpy(last_name, "\0");
          }
          Person(int a, char *fn, char mn[])
//Constructor 4 constructor with three arguments
          {         age = a;
                    strcpy(first_name, fn);
                    strcpy(middle_name, mn);
                    strcpy(last_name, "\0");
          }
          Person(int a, char *fn, char mn[], char * In)
// Constructor 5 constructor with four arguments
          {         age = a;
                    strcpy(first_name, fn);
                    strcpy(middle_name, mn);
                    strcpy(last_name, In);
          }
          void show_data()
          {    cout<<"\n NAME: "<<first_namecc" "<<middle_name<<" "<<last_name;
               cout<<"\t\t AGE: "<<age;
          }
};
main()
{
Person P1;                //Constructor 1 called
P1.show_data();
Person P2(18);            //Constructor 2 called
P2.show_data();
Person P3(21, "Goransh");             //Constructor 3 called
P3.show_data();
Person P4(25, "Aditi", "Raj");                  //Constructor 4 called
```

```
P4.show_data();
Person P5(32, "Pallav", "Raj", "Thareja");    //Constructor 5 called
P5.show_data();
}
```

OUTPUT

```
NAME :         AGE = -1
NAME :         AGE = 18
NAME : Goransh       AGE = 21
NAME : Aditi Raj     AGE = 25
NAME : Pallav Raj Thareja            AGE = 32
```

Note:

When array of objects is created, the constructor is called for each object. Therefore, if we say class student s[5], then the constructor will be called five times not once.

Program 3.1:

Write a program that uses an overloaded constructor to dynamically allocate memory to an array and thus find the largest of its elements.

```
using namespace std;
#include<iostream>
class Array
{ private:
        int *arr;
        int n;
    public:
        Array()
        { n=0; }
        Array(int);
        void show_data();
        int largest();
}
Array ;: Array(int num)
```

```
{ n = num;
   arr = new int [n];
   cout<<"\n Enter the eleme
nts : ";
   for(int i=0;i<n;i++)
       cin>>arr[i];
}
void Array :: show_data()
{ for(int i=0;i<n;i++)
       cout<<" "<<arr[i];
}
int Array :: largest()
{ int largest = arr[0];
       for(int i=1;i<mn;i++)
       { if(arr[i]>largest)
           largest = arr[i];
       }
       return largest;
}
main()
{ int size;
   cout<<"\n Enter the size of the array : ";
   cin>>size;
   Array Arr(size);
   Arr.show_data();
   cout<<"\n Largest = "<<Arr.largest();
}
```

OUTPUT

```
Enter the size of the array : 5
Enter the elements : 10364
10364
```

```
Largest =6
```

Programming Tip: You can call a constructor from a destructor and vice versa

Program 3.2

Write a program that uses dynamic constructor to allocate memory to an array. Count the number of even and odd elements.

```
using namespace std;
#include<iostream>
class Array
{ private:
            int *arr;
         int n;
         int num_even;
         int num_odd;
   public:
         Array(int);
         void show_data();
};
Array :: Array(int num)
{ n = num;
   num_even = 0;
   num_odd = 0;
   arr = new int[n];
   cout<<"\n ENter the elements : ";
   for(int i=0;i<n;i++)
      cin>>arr[i];
}
void Array :: show_data()
{ int i;
   cout<<"\n Array is ";
   for(i=0;i<n;i++)
         cout<<" "<<arr[i];
   for(i=0;i<n;i++)
   { if(arr[i]%2 == 0)
         num_even++;
      else
         num_odd++;
   }
   cout<<"\n Number of evens = "<<num_even;
   cout<<"\n Number of odds = "<<num_odd;
}
main()
{ int size;
   cout<<"\n Enter the size of the array : ";
   cin>>size;
   Array Arr(size);
   Arr.show_data();
}
```

OUTPUT

```
Enter the size of the array : 7
Enter the elements : 1234567
Array is 12 2 3 5 6 7
Number of evens = 3
Number of odds = 4
```

Program 3.3:

Write a program that uses overloaded constructors and dynamically allocates memory to a string. Demonstrate the use of copy constructor

```
using namespace std;
#include<iostream>
#include<string.h>
class String
{        private:
                 int length;
                 char *str;
```

```
        public:
                String()
                {       length = 0;     }
                String(char *s)
                {       length = strlen(s);
                        str = new char[length + 1];
                        strcpy(str, s);
                }
                String(String &s)
                {       length=s.length;
                        str = new char[length + 1];
                        strcpy(str, s.str);
                }
                void show str()
                {       cout.write(str, length);}
};
main()
{       String s1("Welcome to the world of programming");
        String s2=s1;
        cout<<"\n String (s2)=";
        s2.show_str();
}
```

OUTPUT

```
String (s2) = welcome to the world of programming
```

3.6. Constant Objects:

Since C++, treats user defined variables same as built in data type variables, it permits users to declare constant objects as we have const int, const float, etc. The syntax for defining a constant object is as follows:

3.6.1. Key Features of Constant Object:

Constant object can be initialized only by a constructor. This means that a constant object is initialized during its creation.

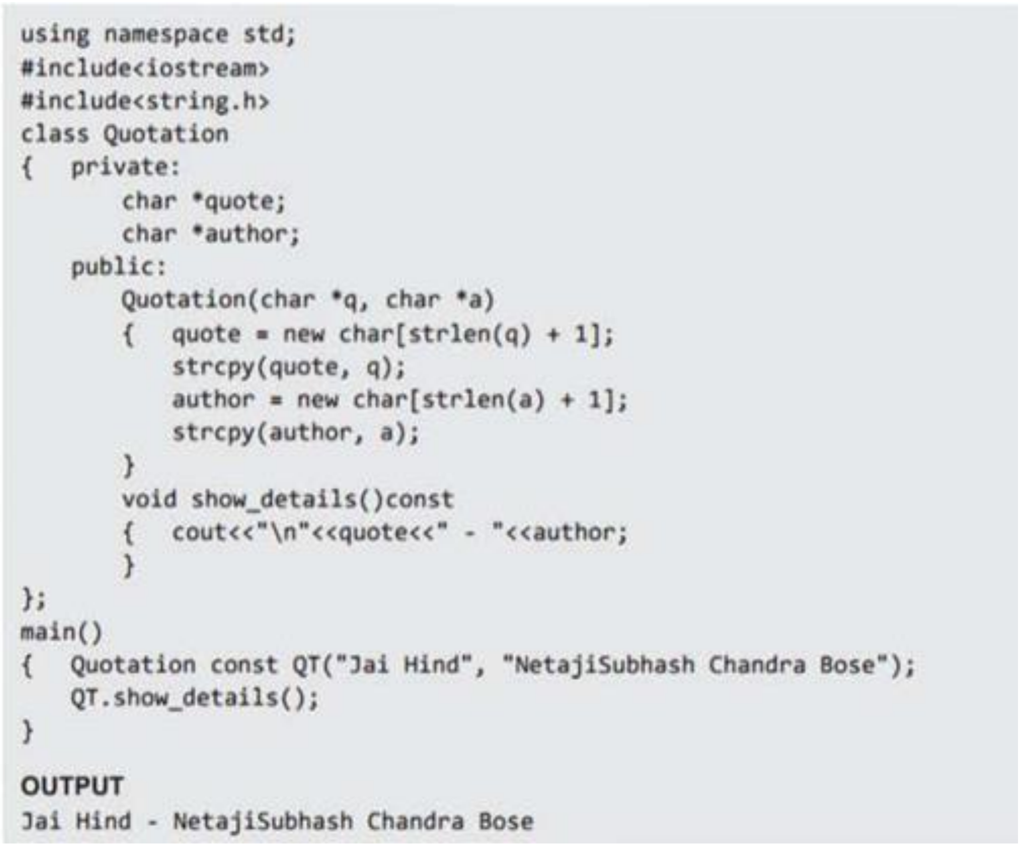
- Data members of the constant object cannot be modified by any member function. This means

that member function can only read the values of data members and not modify them.

- Constant objects behave as read only object. Their data members are read only members.

Let us consider a small program which declares a constant object.

Example 1: Constant object



```
using namespace std;
#include<iostream>
#include<string.h>
class Quotation
{   private:
        char *quote;
        char *author;
    public:
        Quotation(char *q, char *a)
        {   quote = new char[strlen(q) + 1];
            strcpy(quote, q);
            author = new char[strlen(a) + 1];
            strcpy(author, a);
        }
        void show_details()const
        {   cout<<"\n"<<quote<<" - "<<author;
        }
};
main()
{   Quotation const QT("Jai Hind", "NetajiSubhash Chandra Bose");
    QT.show_details();
}
```

OUTPUT

```
Jai Hind - NetajiSubhash Chandra Bose
```

Explanation:

In the program, QT is a constant object and show_details() is a constant member function. Now,if you make a member function that tries to change the quote or its author, will result in an error.

Note Constant objects should be **accessed** only by constant member functions.

3.7. Destructors:

Like a constructor, a destructor is also a member function that is automatically invoked. However, unlike the constructor which constructs the object, the job of destructor is to destroy the object. For this, it deallocates the memory dynamically allocated to the variable(s) or perform other cleanup operations.

3.7.1 Important Features:

- The name of the destructor is also the same as that of the class. However, the destructor's name is preceded by the tilde symbol ' ~ '.
- A destructor is called when an object goes out of scope.
- A destructor is also called when the programmer explicitly deletes an object using the delete operator.
- Like a constructor, a destructor is also declared in the public section.
- The order of invoking a destructor is the reverse of invoking a constructor.
- Destructors do not take any argument and hence cannot be overloaded.
- A destructor does not return any value.
- A destructor must be specifically defined to free (de-allocate) the resources such as memory and files opened that have been dynamically allocated in the program.
- The address of a destructor cannot be accessed in the program.
- An object with a constructor or a destructor cannot be used as a member of a union.
- Constructors and destructors cannot be inherited. We will discuss this topic in Chapter 12 on Inheritance.
- Unlike constructors, destructors can be virtual. We will read more on this in Chapter 12 on Inheritance.

Note: A class can have only one destructor.

Consider the program code and its output which makes it clear that the invocation timing of a destructor is just the reverse of a constructor.

Example 1: Calling a constructor, destructor

Programming Tip: Deleting an object more than once is a serious error.

```
using namespace std;
```

```
#include<iostream>
        class Sample
        { private:
                    int x ;
                    public:
                    Sample(int n )
                    { x=n;
                     cout<<"\n Constructor Called for object with value : "<<x;
    }
    ~Sample()
    { cout<<"\n Destructor Called for object with value : "<<x;
    }
};
main()
{ Sample S1(1);
Sample S2(2);
Sample S3(3);
}
```

OUTPUT

```
Constructor Called for object with value : 1
Constructor Called for object with value : 2
Constructor Called for object with value : 3
Destructor Called for object with value : 3
Destructor Called for object with value : 2
Destructor Called for object with value : 1
```

Note If the programmer has not defined any destructor, then the C++ compiler automatically declare a destructor as an inline public member function of its class.

Program 1:

Write a program that uses static variables to keep a track of number of objects created, number of objects destroyed, and number of active objects in a program.

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

```
class Sample
{ private:
   static int num_objects;
   static int num_objects_destroyed;
   public:
      Sample()
   { num_objects++; }
   ~Sample()
   { num_objects_destroyed++; }
   void show_data()
   { cout<<"\n\n\n Number of objects created : "<<num_objects;
         cout<<"\n Number of objects destroyed : "<<num_objects_destroyed;
         cout<<"\n Number of objects active : "<<num_objects –
                                                                                    num_objects_destroyed;
    }
};
int Sample :: num_objects = 0;
int Sample :: num_objects_destroyed = 0;
void my_func()
{ Sample s8, s9;
  s9.show_data();
}
main()
{ Sample s1, s2, s3;
   { s1.show_data();
      Sample s4;
      { Sample s5;
         s5.show_data();
      }
   }
   s3.show_data();
   my_func();
   s2.show_data();
}
```

OUTPUT

```
Number of objects created : 3
Number of objects destroyed : 0
Number of objects active : 3
Number of objects created : 5
Number of objects destroyed : 0
Number of objects active : 5
Number of objects created : 5
Number of objects destroyed : 2
Number of objects active : 3
Number of objects created : 7
Number of objects destroyed : 2
Number of objects active : 5
Number of objects created : 7
Number of objects destroyed : 4
Number of objects active : 3
```

Program 2:

Write a program that dynamically allocates memory to a matrix. Add two matrices, display the resultant matrix, and finally free the memory space.

```
#include<iostream.h>
class Matrix
{ private:
   int **parr;
   int rows;
   int cols;
  public:
      Matrix()
      { rows = cols = 0;
         parr = NULL;
      }
```

```
        ~Matrix();
        Matrix(int r, int c);
        void get_data();
        void show_data();
        Matrix &add(Matrix &A1, Matrix &A2);
};
Matrix :: Matrix(int r, int c)
{ rows = r;
    cols = c;
    parr = new int *[rows];
    for(int i=0;i<rows;i++)
        parr[i] = new int [cols];
}
Matrix :: ~Matrix()
{ for(int i=0;i<rows;i++)
            delete parr[i];
    delete parr;
    cout<<"\n DESTROYED";
}
void Matrix :: get_data()
{ for(int i=0;i<rows;i++)
    { for(int j=0;j<cols;j++)
        cin>>parr[i][j];
    }
}
void Matrix :: show_data()
{ for(int i=0;i<rows;i++)
    { cout<<"\n";
                for(int j=0;j<cols;j++)
                cout<<" "<<parr[i][j];
    }
}
```

```
Matrix & Matrix :: add(Matrix &A1, Matrix &A2)
{ for(int i=0;i<rows;i++)
   { for(int j=0;j<cols;j++)
         parr[i][j] = A1.parr[i][j] + A2.parr[i][j];
   }
   return *this;
}
main()
{ int r, c;
   cout<<"\n Enter the number of rows and columns of 2D Arrays : ";
   cin>>r>>c;
   Matrix Arr1(r,c), Arr2(r,c), Arr3(r,c);
   cout<<"\n Enter the first matrix : ";
   Arr1.get_data();
   cout<<"\n Enter the second matrix : ";
   Arr2.get_data();
   Arr3.add(Arr1,Arr2);
   cout<<"\n Array 1 is :\n";
   Arr1.show_data();
   cout<<"\n Array 2 is :\n";
   Arr2.show_data();
   cout<<"\n Resultant Array is :\n";
   Arr3.show_data();
}
```

OUTPUT

```
Enter the number of rows and columns of 2D Arrays : 2 2
Enter the first matrix : 1 2 3 4
Enter the second matrix : 5 6 7 8
Resultant Array is :
6  8
10 12
Destroyed
```

```
Destroyed
Destroyed
```

Program 3:

Write a program that dynamically allocates memory to a string. Encrypt this string and de-allocate the memory.

```
using namespace std;
#include<iostream>
#include<string.h>
class String
{         private:
                              int len;
                         char *str;
              public:
                         String(char *s)
                          {         len = strlen(s);
                                    str = new char[len+1];
                                    strcpy (str, s);
                           }
                       ~String()
                       { len = 0;
                       delete []str;
                       }
                       void show_string()
                       { cout<<str;
                       }
                       char * encrypt();
};
char * String :: encrypt()
{ int i = 0;
   while(i<len)
   { str[i] += 3;
          i++;
   }
   show_string();
}
main()
{ String S1("ABCDE");
   cout<<"\n Original String : ";
   S1.show_string();
   cout<<"\n Encrypted String : ";
   S1.encrypt();
}
```

OUTPUT

```
Original String : ABCDE
Encrypted String : DEFGH
```

Program 4:

Write a program to sort an array that has been allocated memory dynamically.

```
using namespace std;
#include<iostream>
class Array
{ private:
   int size;
   int *arr;
   public:
      Array()
   { size = 0;
      arr = NULL;
   }
   Array(int n)
   { size = n;
      arr = new int [size];
   }
   ~Array()
   { cout<<"\n Destroyed ";
```

```
        delete [] arr;
        size = 0;
    }
    void get_data();
    void show_data();
    void sort();
};
void Array :: get_data()
{ for(int i=0;i<size;i++)
    cin>>arr[i];
}

void Array :: show_data()
{ for(int i=0;i<size;i++)
    cout<<" "<<arr[i];
}
void Array :: sort()
{ for(int i=0;i<size;i++)
    { for(int j=0;j<size-i;j++)
        { if(arr[j]>arr[j+1])
            { int temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
                }
            }
    }
}
main()
{ Array arr(7);
    cout<<"\n Enter the array elements : ";
    arr.get_data();
    cout<<"\n Original Array : ";
    arr.show_data();
    arr.sort();
    cout<<"\n Sorted Array : ";
    arr.show_data();
}
OUTPUT
Enter the array elements : 9 1 8 3 645
Original Array : 9 1 8 3645
Sorted Array : 1 3 4 5689
Destroyed
```

Note Unlike local objects, static objects are constructed only the first time they are defined. These objects are destroyed at the end of the program.

3.7.2 Interesting Points about Constructors and Destructors

- You can call a constructor from main() by writing

```
object_name.constructor name
```

However, to call a destructor from main(), you must specify the object name as well as the class name. For example, if we have a class Sample, and s as the object of Sample, then to call its constructor we must write,

```
s.Sample();
```

To call the destructor, you need to write

```
s.Sample :: ~Sample();
```

- You can call a constructor from a destructor. You can also call a destructor from a constructor as shown in Fig. 10.3

class Sample
{ -------
------- 	
Sample()
{ -------
Sample :: ~ Sample();
}
------- 	
};
Note: The destructor is being called using the class name.
	class Sample
	{ --------
	-------- 	
	~Sample()
	{ Sample();
	-------- 	
	}
	};
	Note: The constructor is being called without using the class name. However, it may result in recursive call of constructor-destructor till stack overflows and abnormal program termination occurs.

class Sample
{ --------
-------- 	
Sample()
{ ---- ----
Sample :: ~ Sample();
}
-------- 	
}; Note: The destructor is being called using the class name.
	class Sample
	{ ------
	------ 	
	~Sample()
	{ Sample();
	----- ---
	}
	};
	Note: The constructor is being called without using the class name.

class Sample
{ --------
-------- 	
Sample()
{ --------
Sample :: ~ Sample();
}
------- 	
};
Note: The destructor is being called using the class name.
	class Sample
	{ -------
	------- 	
	~Sample()
	{ Sample();
	---- ---
	}
	};
	Note: The constructor is being called without using the class name.

Figure 3.3 Calling destructor

- If calling a constructor or a destructor is based on some decision, then it is called a conditional constructor or a conditional destructor, as shown in Fig. 3.3.
- C + + allows constructors and destructors to be defined in the private section. However, it is a good programming habit to declare them in the public section. When a constructor or a destructor is declared as private, it cannot be called implicitly. They have to be called explicitly through a public member function as shown here.

class Sample
{ --------
-------- 	
Sample()
{ if(cond)
Sample :: ~ Sample();
}
------- 	
};
	class Sample
	{ -------
	------- 	
	~Sample()
	{ if(cond)
	Sample();
	------ 	
	}
	};

Figure 3.4 Conditional destructor

```
using namespace std;
#include<iostream>
class Sample
{
   int val; //Stores the Data
   Sample() //Private Constructor
   {
      val = 10;
   }
   public:
      static Sample Initialize()
//Static Member Function generally called the Factory Method.
   {
      return Sample(); //Return the Object.
   }
   void Display() //Used to display the result.
   {
      cout<<"Value of Data is "<<val<<endl;
   }
};
main()
{
   Sample Obj = Sample::Initialize();
//Static Member function called to return the object.
   Obj.Display();
}
```

OUTPUT

```
Value of Data is 10
```

- C++ allows programmers to have a class named main. However, when there is a class named main, the use of keyword class before main (the class_name) is mandatory. This is shown in the code given below.

```
class main
{ private:
   int x;
  public:
       main()
       { x=10
         cout<<"\n x = "<<x;
       }
~main()
       {  cout<<"\n Destroyed";
       }
   };
main()
{ class main M;
}
```

OUTPUT

```
   x = 10
   Destroyed
```

- You may call a constructor recursively.
- We know that program execution starts with main(). However, when the program has a global object (one defined before main()), the constructor of the global objects gets called before the main(). However, the destructor of the global object is executed after the execution of main(). Execute the code given here to understand this concept.

Example 2: Global object

```
using namespace std;
#include<iostream>
class Sample
{ public:
   Sample()
   { cout<<"\n IN CONSTRUCTOR"; }
   ~Sample()
   { cout<<"\n IN DESTRUCTOR"; }
};
Sample s;
main()
{ cout<<"\n IN MAIN";
}
```

OUTPUT

```
IN CONSTRUCTOR
IN MAIN
IN DESTRUCTOR
```

Exercises

1.What is the significance of a constructor?

2. Is it mandatory to define a constructor for every class**?**

3. Can a C++ program that has a class execute with out a constructor?

4. Why a constructor is called a special member function?

5. Explain some key features of a constructor.

6. Discuss the different types of constructors.

7. Why does copy constructor accept the objects by reference and not by value?

8. Explain the features of a destructor function.

9. Differentiate between a constructor and a destructor.

10. How can a destructor be called from a constructor?

Programming Exercises

1. Write a program that demonstrates the use of a parameterized constructor.
2. Write a program that demonstrates the use of a copy constructor.
3. Write a program that demonstrates the use of default arguments in a constructor.
4. Write a class that stores a string and its status details such as number of lower case characters, consonants, and so on.
5. Write a program that demonstrates the use of constant objects.

Unit IV

Introduction, defining operator over loading, over loading unary operator, Overloading Binary operator, Overloading Binary operators using Friends, Manipulation of strings using operators, Some other Operator over loading examples, Rules for Over loading Operators

Chapter 4: Sections 4.1-4.6.

4.1. Introduction:

The utility of operators such as +, =, *, /, >,and so on is predefined in any programming language. Programmers can use them directly on built-in data types to write their programs. However, these operators do not work for user-defined types such as objects. Therefore, C++ allows programmers to redefine the meaning of operators when they operate on class objects. This feature is called operator overloading. With this feature, the overloading principle is not only applied to functions but also on operators.

Operator overloading allows programmers to extend the meaning of existing operators so that in addition to the basic data types, they can be also applied to user-defined data types.

4.2. Scope of Operator Overloading:

With operator overloading, a programmer is allowed to provide his own definition for an operator to a class by overloading the built-in operator. This enables the programmer to perform specific computation when the operator is applied on class objects and apply a standard definition when the same operator is applied on a built-in data type. Therefore, while evaluating an expression with operators, C++ looks at the operands around the operator. If the operands are of built-in types, C++ calls a built-in routine. If the operator is being applied on user-defined operand(s), the C++ compiler checks if the programmer has an overloaded operator function that it can call. If such a function whose parameters match the type(s) and number of the operands exists in the program, the function is called; otherwise, a compiler error is generated.

Another Form of Polymorphism

Like function overloading, operator overloading is also a form of compile time polymorphism. Operator overloading is, therefore, less commonly known as operator adhoc polymorphism since different operators have different implementations depending on their arguments. Operator overloading is generally defined by the language, the programmer, or both.

Advantages of Operator Overloading

We can easily write C++ programs without the knowledge of operator overloading; however, the knowledge and use of this feature can help us in many ways. Some of them are as follows: Operator overloading enables programmers to use notation closer to the target domain. For example, to add two matrices, we can simply write M1 + M2, rather than writing M1.add (M2). With operator overloading, a similar level of syntactic support is provided to user-defined types as provided to the built-in types.

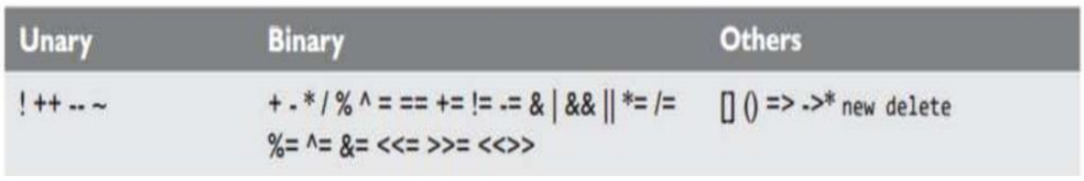
In scientific computing where computational representation of mathematical objects is required, operator overloading provides great ease to understand the concept.

Operator overloading makes the program clear. For example, the statement div(mul(M1, M2), add(M1,M2)); can be better written as M1 * M2 / M1+M2

4.3. Operators that can and cannot be overloaded:

In C++, barring a few operators, programmers can overload almost any operator. The operators that can be overloaded are given in Table 4.1.

Table 4.1 Operators that can be overloaded



Unary	Binary	Others
! ++ -- ~	+ - * / % ^ = == += != -= & | && || *= /= %= ^= &= <<= >>= <<>>	[] () => ->* new delete

However, the exceptional operators that cannot be overloaded are as follows:

- Scope resolution operator (::)
- Member selection operator (.)
- Member selection through a pointer to a function (.*)
- Ternary operator (?:)
- Size of operator (size of)

Important Points about Operator Overloading

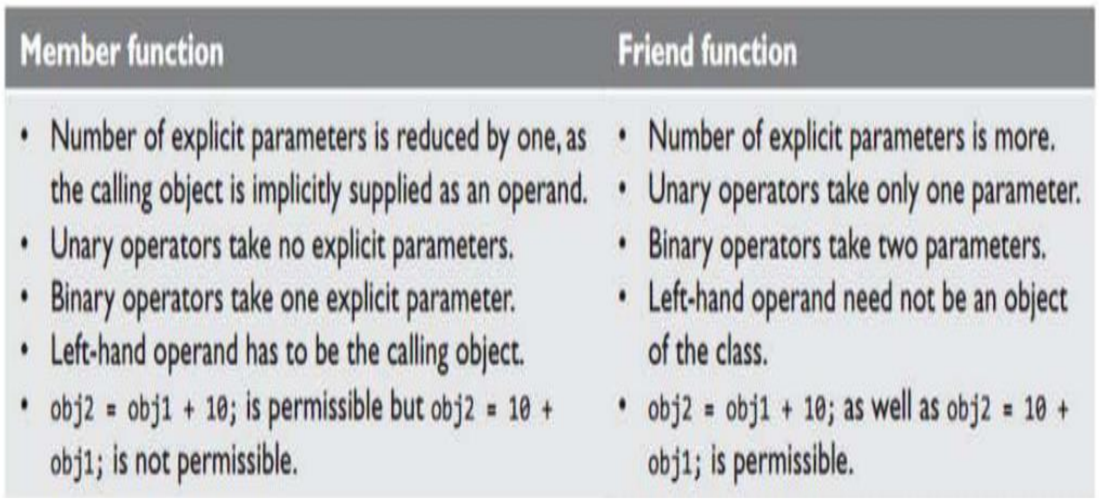
- Operator overloading should not change the operation performed by an operator. You cannot redefine the meaning of an operator. Therefore, when the + operator will be overloaded, it will always perform addition and not subtraction, multiplication, or any other operation.
- The two operators-the assignment operator (=) and the address operator(&)-need not be overloaded. This is because these two operators are already overloaded in the C++ library. For example, when you write Complex1 = Complex2, then the contents of object Complex2 will be copied into the contents of object Complex1 automatically. Similarly, the address operator returns the address of every object in memory.
- Operator overloading cannot alter the precedence and the associativity of operators. However, you may use parenthesis to change the order of evaluation.
- New operators such as 1ike **, <>, 1&, and so on cannot be created. Only existing operators can be overloaded.
- When operators such as &&, ||, and, are overloaded, they lose their special properties of short-circuit evaluation and sequencing. This means that after overloading, say && operator, you cannot expect &&= to perform the logical AND as well as assignment. To perform both the operations, you must specifically overload &&= operator and not just && operator.
- Overloaded operators cannot have default arguments.
- All overloaded operators except the assignment operator are inherited by derived classes. This will be clear in Chapter 12 on Inheritance.
- Arity or the number of operands cannot be changed. Therefore, a unary operator such as ++, --, !, and so on will always be applied on one operand and binary operator such as +, -, *, and so on will be applied on two operands.
- Overloading an operator that is not associated with the scope of a class is not permissible. This means that the overloaded operator must have access to at least one object of the class in which it is being overloaded. This object may be accessed implicitly (using member function) or explicitly (using the friend function).

4.4. Implementing Operator Overloading:

Operator overloading is usually implemented in two ways as follows:

- Through member function
- Through friend function

Although operators can be easily overloaded using any of these techniques, the choice of technique is just a matter of programmer's convenience. However, you must consider the major difference when overloading using a member and/or a friend function as shown in Table 2.



Member function	Friend function
• Number of explicit parameters is reduced by one, as the calling object is implicitly supplied as an operand.	• Number of explicit parameters is more.
• Unary operators take no explicit parameters.	• Unary operators take only one parameter.
• Binary operators take one explicit parameter.	• Binary operators take two parameters.
• Left-hand operand has to be the calling object.	• Left-hand operand need not be an object of the class.
• `obj2 = obj1 + 10;` is permissible but `obj2 = 10 + obj1;` is not permissible.	• `obj2 = obj1 + 10;` as well as `obj2 = 10 + obj1;` is permissible.

4.5. Overloading Unary Operators:

Unary operators work only on a single operand. Some examples of unary operators are as follows:

- Increment operator (++)
- Decrement operator (--)
- Unary minus operator (-)
- Logical not operator (!)

As stated earlier, unary operators can be overloaded using a friend function or a member function. In both the cases, the unary operator operates on the object for which it was called. Usually, the operator is specified on the left side of the object, as in ++obj, -obj, and !obj. However, the operator may appear on the right side of the object when it is a post-fix increment or a post-fix decrement such as obj++ or obj--.

4.5.1. Using a Member Function to Overload a Unary Operator

The syntax for operator overloading using a member function can be given as

return_type operator op()

where return_type is the return type and op is the operator to be overloaded. Let us consider a small program that overloads the unary operator using member function. We know that if we have a number say, 7, then the unary operator when applied to it will make it negative, which is -7. However, if we apply the same operator to a number say -10, then the result would be 10 (positive).

Example 1:

Overloading unary operator with member function

using namespace std;

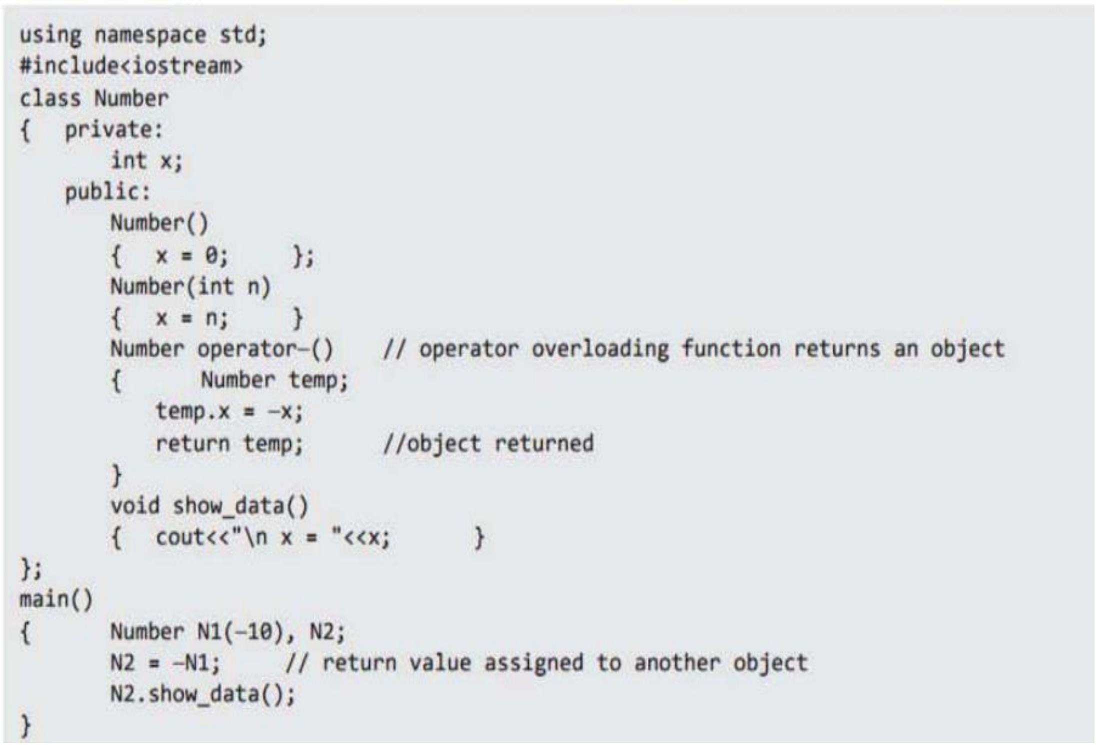
 #include <iostream>

```
class Number
{   private:
        int x;
    public:
        Number()
        {   x = 0;     };
        Number(int n)  //parameterized constructor
        {   x = n;     }
        void operator-()   // operator overloaded function
        {   x = -x; }
        void show_data()
        {   cout<<"\n x = "<<x;       }
};
main()
{   Number N(7);   // create object
    N.show_data();
    -N;            // invoke operator overloaded function
    N.show_data();
}

OUTPUT
x = 7
x = -7
```

4.5.2. Returning Object:

The operator overloading function in Example 11.1 does not return any value. However, if we want to have a statement like N2 =-N1; then the above code must return a value (an object). Therefore, the function code can be modified to return a value as shown in Example 11.2 given here.

Example 1: Returning Object



```
using namespace std;
#include<iostream>
class Number
{   private:
        int x;
    public:
        Number()
        {   x = 0;     };
        Number(int n)
        {   x = n;     }
        Number operator-()    // operator overloading function returns an object
        {       Number temp;
            temp.x = -x;
            return temp;      //object returned
        }
        void show_data()
        {   cout<<"\n x = "<<x;       }
};
main()
{       Number N1(-10), N2;
        N2 = -N1;     // return value assigned to another object
        N2.show_data();
}
```

OUTPUT

```
x = 10
```

4.5.3. Returning a Nameless Object:

The function code operator overloading could be made simpler and shorter by returning a nameless object. This could be done by writing

```
Number operator -))
 {              x = -x;
               return Number(x);
```

}

Note:

There is no restriction on the return types of the unary operators.

4.5.4. Using a Friend Function to Overload a Unary Operator:

When a unary operator is overloaded using a friend function, then you must ensure the following:

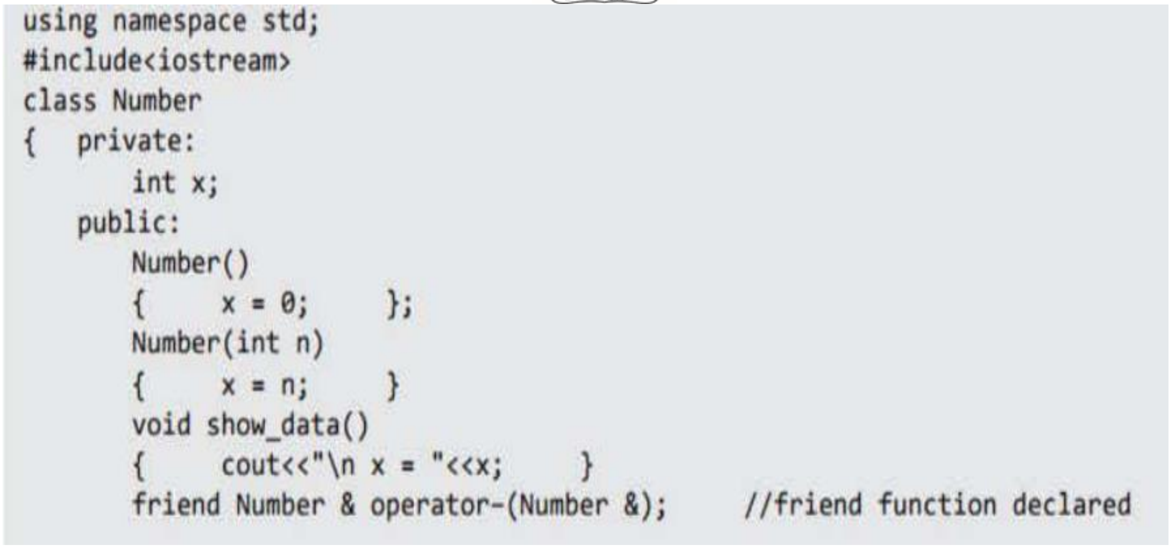
- The function will take one operand as an argument.
- This operand will be an object of the class.
- The function will use the private members of the class only with the object name.
- The function may take the object by using value or by reference. However, if the object is passed using value, then any changes made to the data members of the object in the function will not be reflected back in the calling function. Therefore, if you want the changes to persist, pass the object by reference.
- The function may or may not return any value. It depends on the usage. Since we had to assign the value to another object we have returned.
- The friend function does not have access to this pointer. The syntax for operator overloading using a member function can be given as

friend return_type operator op (class_name object)

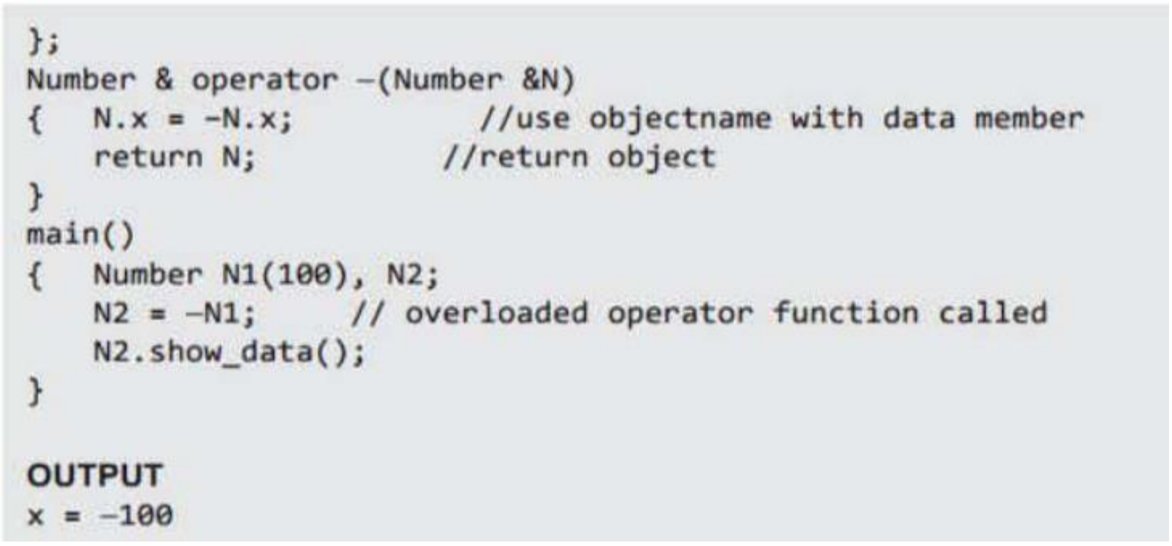
where return_type is the return type, and op the operator to be overloaded and object is an instance of the class on which the operator has to be applied. The program given here performs the same operation but using a friend function.

Example 1:

Friend function for operator overloading



```
using namespace std;
#include<iostream>
class Number
{   private:
        int x;
    public:
        Number()
        {     x = 0;     };
        Number(int n)
        {     x = n;     }
        void show_data()
        {     cout<<"\n x = "<<x;     }
        friend Number & operator-(Number &);     //friend function declared
```



```
};
Number & operator -(Number &N)
{   N.x = -N.x;           //use objectname with data member
    return N;            //return object
}
main()
{   Number N1(100), N2;
    N2 = -N1;      // overloaded operator function called
    N2.show_data();
}

OUTPUT
x = -100
```

Note:

Operator overloaded functions return values so that a cascaded assignment expression can be formed.

4.5.5. Overloading the Prefix Increment and Decrement Operators:

The syntax for overloading prefix increment and decrement operators is

operator ++ ()

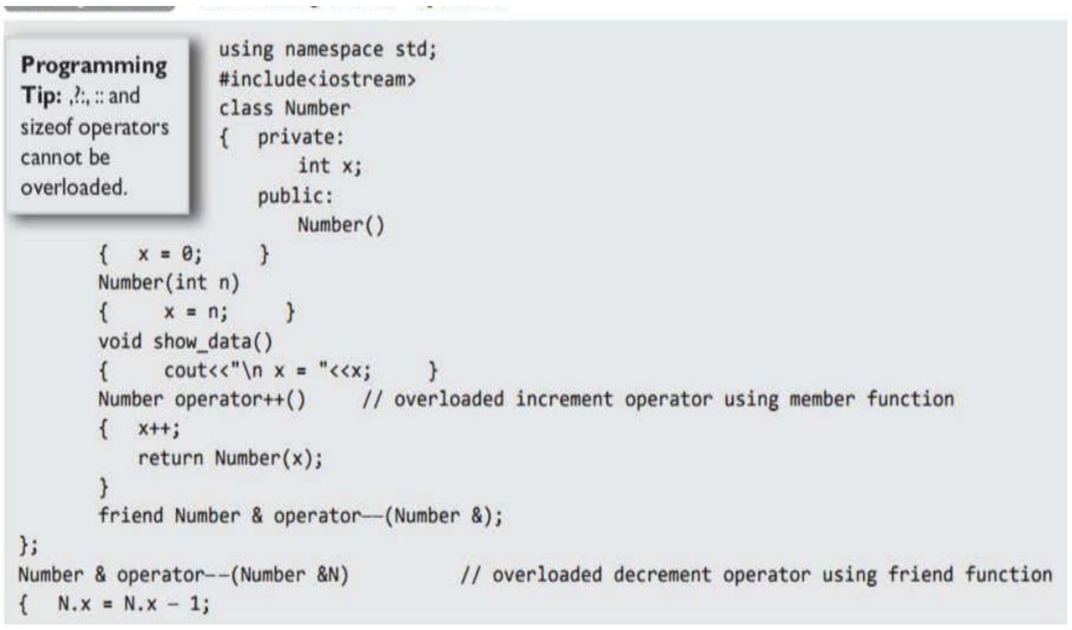
{ // code

}

In this Example 1, we have overloaded the ++ and - operators. While ++ operator has been overloaded using a member function, the operator on the other hand has been overloaded using a friend function. Although operators can be overloaded in any of the two ways, we have used both the techniques for better clarity. Note that the function code for both the operators first changes the original value and then returns the object with the modified value.

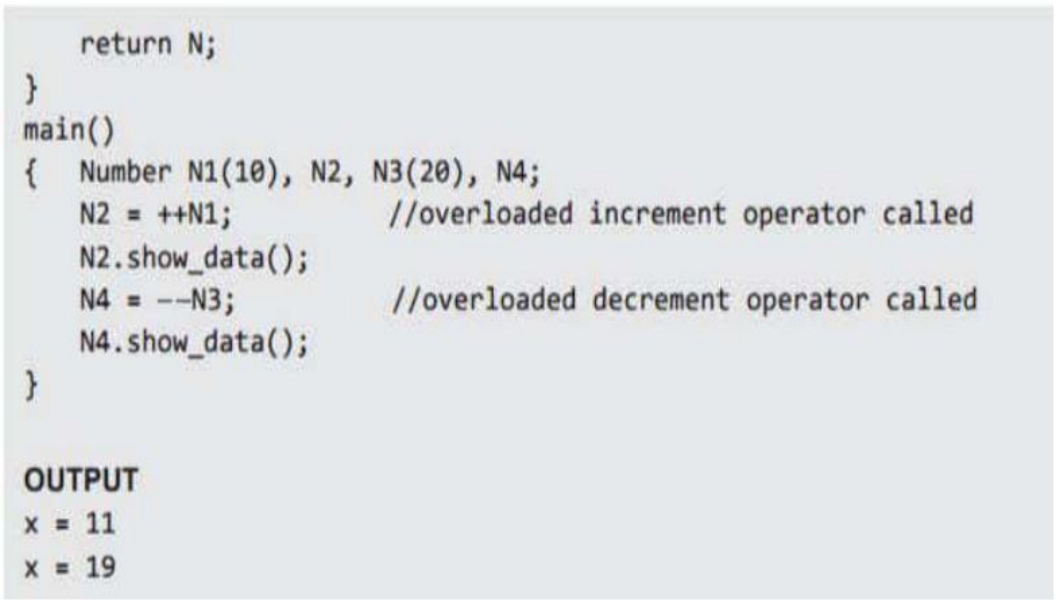
Example 1:

Overloading ++ and – operators



Programming Tip: .,?:, :: and sizeof operators cannot be overloaded.

```
using namespace std;
#include<iostream>
class Number
{   private:
        int x;
    public:
        Number()
        {   x = 0;     }
        Number(int n)
        {     x = n;     }
        void show_data()
        {     cout<<"\n x = "<<x;     }
        Number operator++()     // overloaded increment operator using member function
        {   x++;
            return Number(x);
        }
        friend Number & operator--(Number &);
};
Number & operator--(Number &N)          // overloaded decrement operator using friend function
{   N.x = N.x - 1;
```



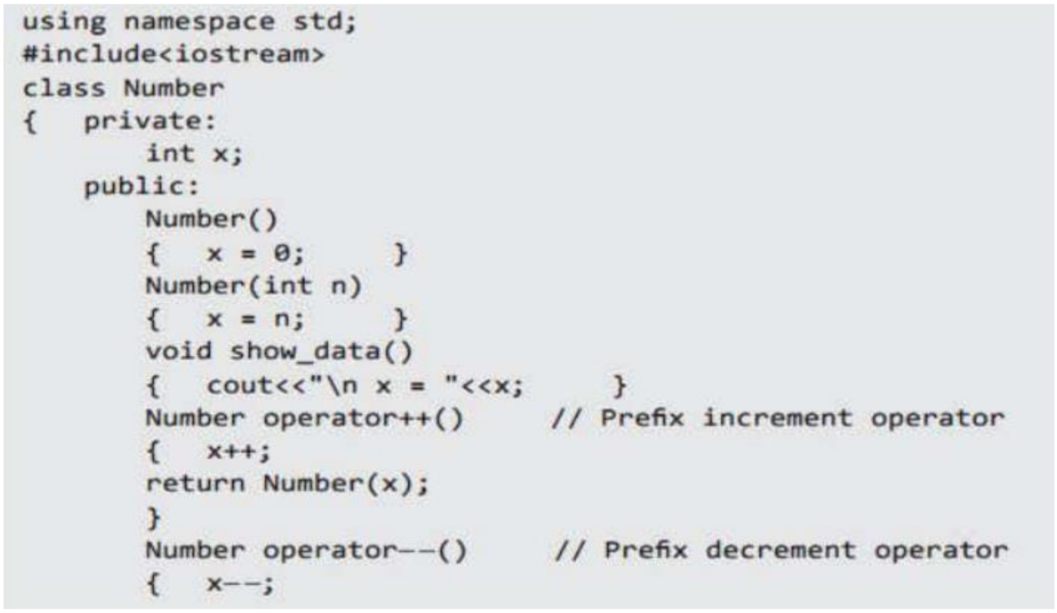
```
    return N;
}
main()
{   Number N1(10), N2, N3(20), N4;
    N2 = ++N1;          //overloaded increment operator called
    N2.show_data();
    N4 = --N3;          //overloaded decrement operator called
    N4.show_data();
}

OUTPUT
x = 11
x = 19
```

4.5.6. Overloading the Post-fix Increment and Post-fix Decrement Operators:

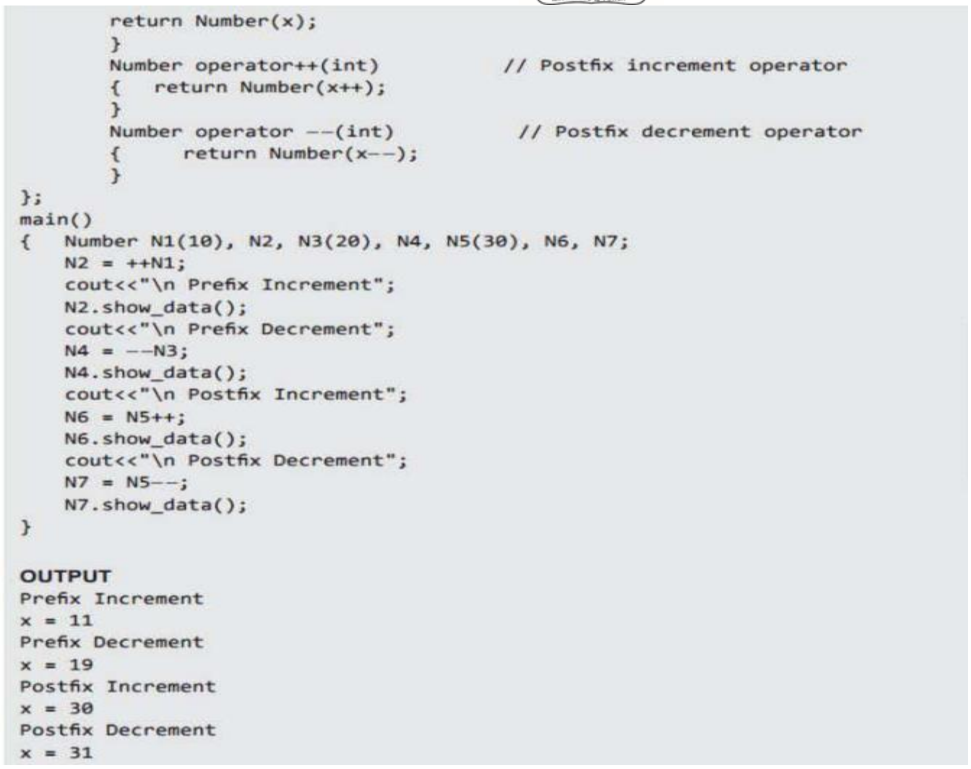
Although the prefix and post-fix increment operators both have the same symbol ++, they have different tasks to perform. Here, the question arises as to how the compiler knows which version to call for. To distinguish between the overloaded prefix and the post-fix operator function, there is a slight change in the signature of post-fix operation. Instead of writing Number operator ++(), we Write Number operator ++(int). Here, int is a dummy argument. Therefore, we have used function overloading for the operator overloaded functions. Always remember that in overloaded post-fix operators, int is just a dummy argument to give a signal to the compiler to create the post-fix notation of the operator. Observe carefully that int is not followed by a variable name just because its value will never be used. Look at the program given here which overloads both the prefix and post-fix increment as well as decrement operators.

Note:

When the compiler sees ++obj (pre-increment), it generates a call to operator++(); but when it sees obj++, it generates a call to operator++(int).



```
using namespace std;
#include<iostream>
class Number
{   private:
        int x;
    public:
        Number()
        {   x = 0;      }
        Number(int n)
        {   x = n;      }
        void show_data()
        {   cout<<"\n x = "<<x;     }
        Number operator++()      // Prefix increment operator
        {   x++;
        return Number(x);
        }
        Number operator--()      // Prefix decrement operator
        {   x--;
```



```
        return Number(x);
        }
        Number operator++(int)          // Postfix increment operator
        {   return Number(x++);
        }
        Number operator --(int)          // Postfix decrement operator
        {      return Number(x--);
        }
};
main()
{   Number N1(10), N2, N3(20), N4, N5(30), N6, N7;
    N2 = ++N1;
    cout<<"\n Prefix Increment";
    N2.show_data();
    cout<<"\n Prefix Decrement";
    N4 = --N3;
    N4.show_data();
    cout<<"\n Postfix Increment";
    N6 = N5++;
    N6.show_data();
    cout<<"\n Postfix Decrement";
    N7 = N5--;
    N7.show_data();
}

OUTPUT
Prefix Increment
x = 11
Prefix Decrement
x = 19
Postfix Increment
x = 30
Postfix Decrement
x = 31
```

4.6. Overloading binary operators:

Binary means two and binary operators mean operators that work with two opearnds. Like unary operators, binary operators such as +, -, *, /, =, , 1, %, ^, &&, ||, <> can also be overloaded. Binary operators are also overloaded either using member functions or friend functions. While the member function will be invoked using an object of the class and will accept one argument, the friend function, on the other hand, will accept two arguments. The syntax of overloading a binary operator using a member function can be given as return_type operator op(arg)

Here, operator is the keyword, op is the operator to be overloaded, and arg is the argument of any type that will be used in the function. While the first operand or the object which invokes the function, is taken implicitly, the other operand on the other hand, is passed explicitly. Therefore, the first operand's data members can be accessed directly without using the dot operator but the second object's data members must be accessed using the dot operator.

For example, if c1 and c2 are objects of complex type, then overloaded + operator can be called c3 = c1 + c2; this is equivalent to c3 = c1. operator+(c2);

Note:

Binary operators must explicitly return a value. They might not attempt to change the original values of the arguments.

Similarly, the syntax to overload a binary operator using a friend function is

friendreturn_type operator op (arg1, arg2)

Here, arg1 and arg2 are arguments of any type. However, one of them must compulsorily be of the class type. When the binary + operator is overloaded in class complex using friend function, then it will be invoked as c3 = c1 + c2; which is equivalent to writing c3 = operator+(c1, c2);

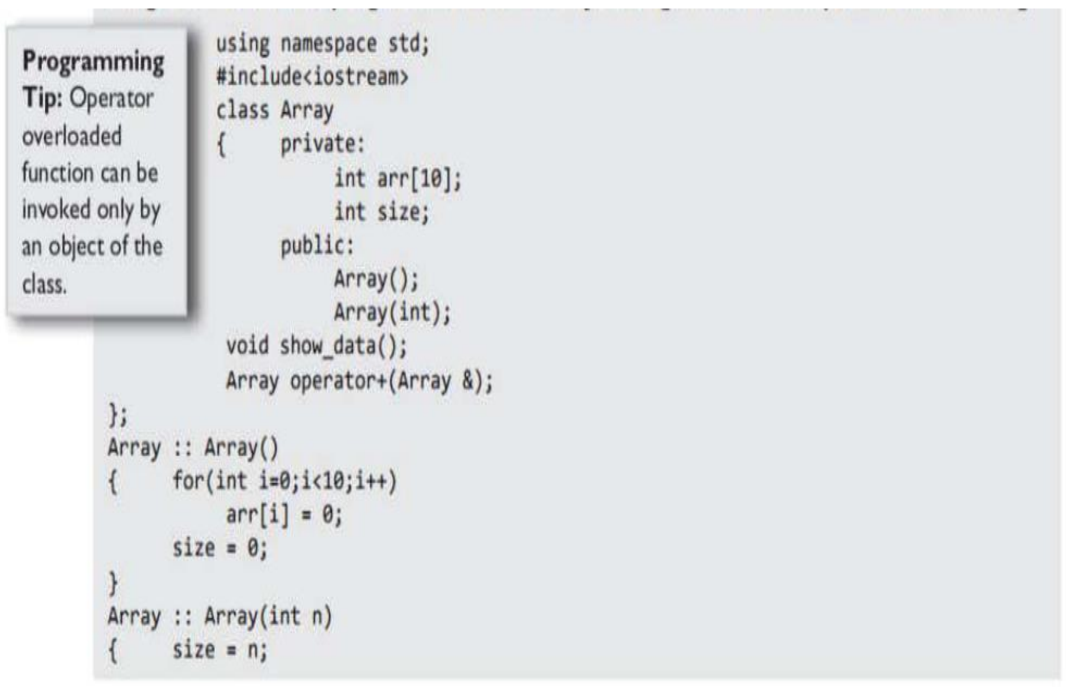
Note:

There is no restriction on the return type of a binary operator overloaded function.

Operators such as =, (), [], and -> cannot be overloaded using friend functions.

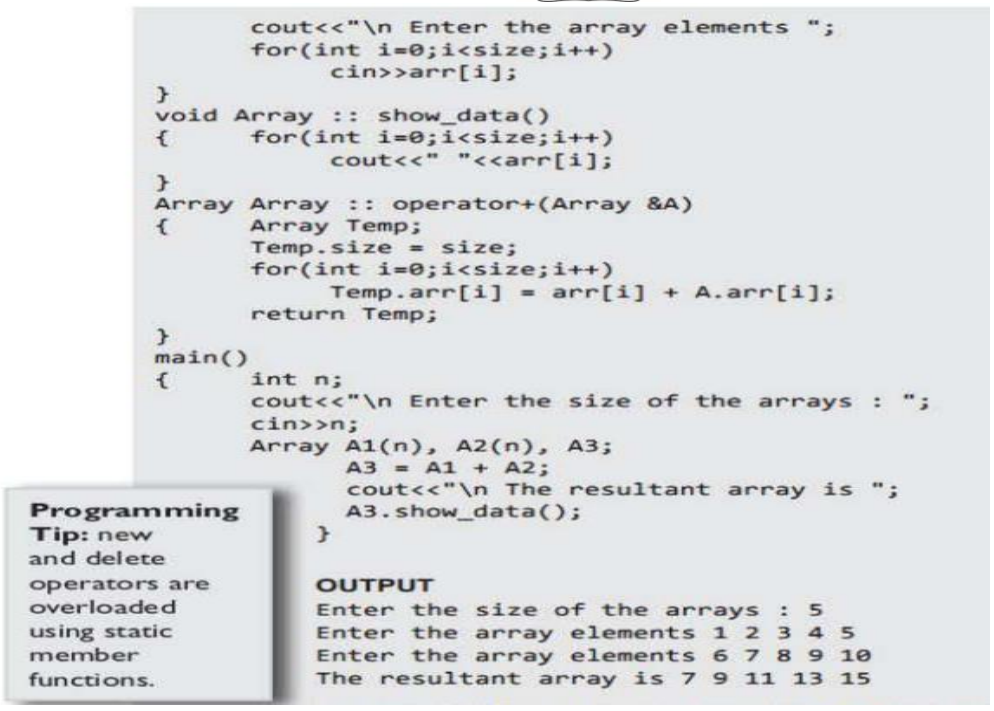
Program 1:

Write a program to add two arrays using classes and operator overloading.



Programming Tip: Operator overloaded function can be invoked only by an object of the class.

```
          using namespace std;
          #include<iostream>
          class Array
          {      private:
                         int arr[10];
                         int size;
                 public:
                         Array();
                         Array(int);
           void show_data();
           Array operator+(Array &);
};
Array :: Array()
{      for(int i=0;i<10;i++)
           arr[i] = 0;
       size = 0;
}
Array :: Array(int n)
{      size = n;
```



```
        cout<<"\n Enter the array elements ";
        for(int i=0;i<size;i++)
             cin>>arr[i];
}
void Array :: show_data()
{       for(int i=0;i<size;i++)
             cout<<" "<<arr[i];
}
Array Array :: operator+(Array &A)
{       Array Temp;
        Temp.size = size;
        for(int i=0;i<size;i++)
             Temp.arr[i] = arr[i] + A.arr[i];
        return Temp;
}
main()
{       int n;
        cout<<"\n Enter the size of the arrays : ";
        cin>>n;
        Array A1(n), A2(n), A3;
              A3 = A1 + A2;
              cout<<"\n The resultant array is ";
              A3.show_data();
    }
```

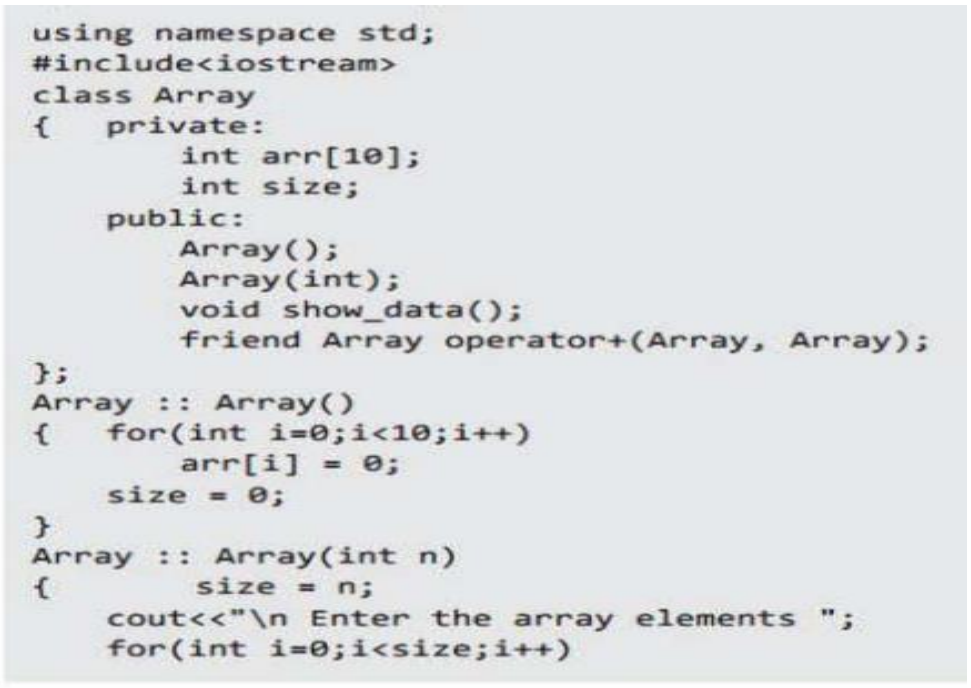
Programming Tip: new and delete operators are overloaded using static member functions.

OUTPUT

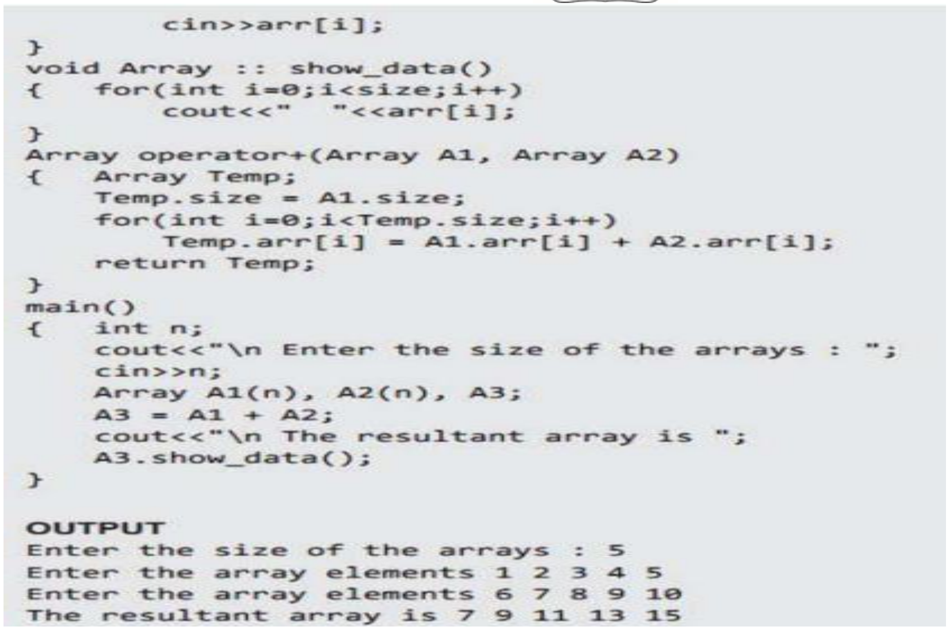
```
Enter the size of the arrays : 5
Enter the array elements 1 2 3 4 5
Enter the array elements 6 7 8 9 10
The resultant array is 7 9 11 13 15
```

Program 2:

Write a program to add two arrays using friend function and operator overloading.



```
using namespace std;
#include<iostream>
class Array
{   private:
         int arr[10];
         int size;
    public:
         Array();
         Array(int);
         void show_data();
         friend Array operator+(Array, Array);
};
Array :: Array()
{   for(int i=0;i<10;i++)
         arr[i] = 0;
    size = 0;
}
Array :: Array(int n)
{         size = n;
    cout<<"\n Enter the array elements ";
    for(int i=0;i<size;i++)
```



```
          cin>>arr[i];
}
void Array :: show_data()
{    for(int i=0;i<size;i++)
          cout<<"  "<<arr[i];
}
Array operator+(Array A1, Array A2)
{    Array Temp;
     Temp.size = A1.size;
     for(int i=0;i<Temp.size;i++)
          Temp.arr[i] = A1.arr[i] + A2.arr[i];
     return Temp;
}
main()
{    int n;
     cout<<"\n Enter the size of the arrays : ";
     cin>>n;
     Array A1(n), A2(n), A3;
     A3 = A1 + A2;
     cout<<"\n The resultant array is ";
     A3.show_data();
}

OUTPUT
Enter the size of the arrays : 5
Enter the array elements 1 2 3 4 5
Enter the array elements 6 7 8 9 10
The resultant array is 7 9 11 13 15
```

4.7. Overloading Special Operators:

In this section, we will discuss about overloading some special operators such as <<, >>, [], (), ->, new, and delete. Before we discuss overloading of these operators, let us quickly revise their functions.

new-to dynamically allocate memory

delete-to free the memory allocated dynamically

<<-to display a message

>>-to accept input from users

[] and ()—are subscript operators

->-to access a class member

4.7.1. Overloading New and Delete Operators

C++ allows programmers to overload the new and delete operators because of the following reasons:

• To allow users to allocate memory in a customized way.

• To allow users to debug the program and keep track of memory allocation and deallocation in their programs.

• To allow users to perform additional operations while allocating or deallocating memory.

The syntax for overloading the new operator can be given as follows:

void* operator new(size_t size**);**

The overloaded new operator receives a parameter size of type size_t, which specifies the number of bytes of memory to be allocated.

The return type of the overloaded new must be void*. The overloaded function returns a pointer to the beginning of the block of memory allocated.

Similarly, the syntax for the overloaded delete operator can be given as

void operator delete(**void***)**;**

The function receives a parameter ptr of type void* which has to be deleted. Note that the function should not return anything.

Note that both of these overloaded new and delete operator functions are static members by default. Therefore, they do not have access to the this pointer.

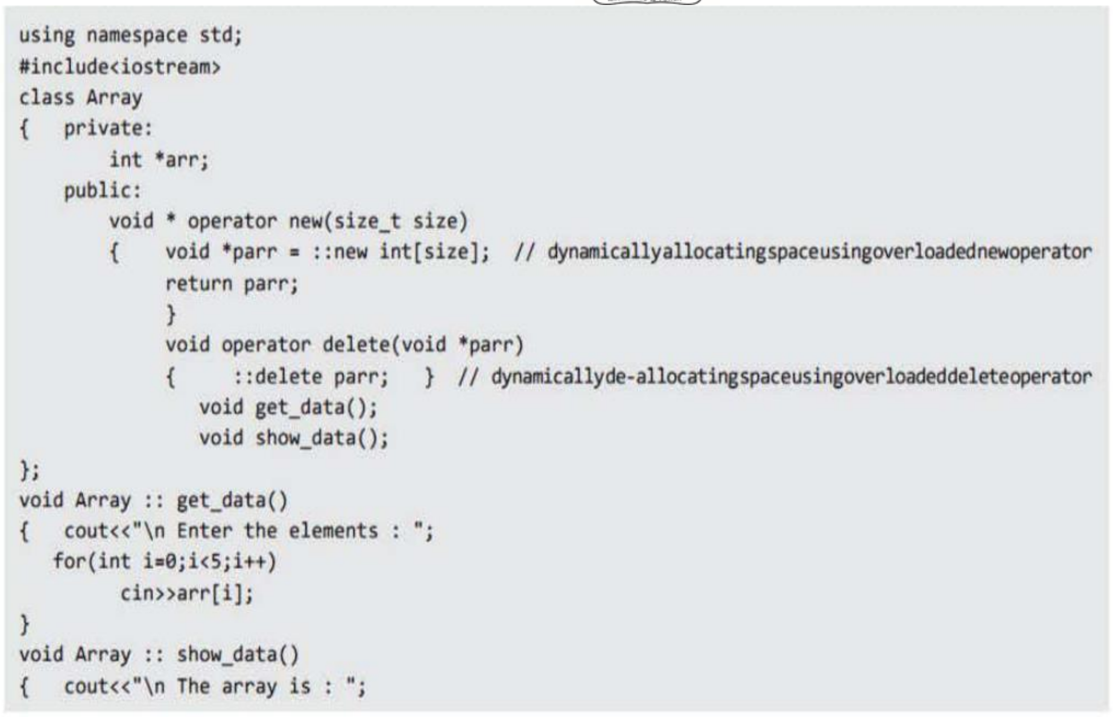
Note:

To delete an array of objects, the operator delete [] must be overloaded.

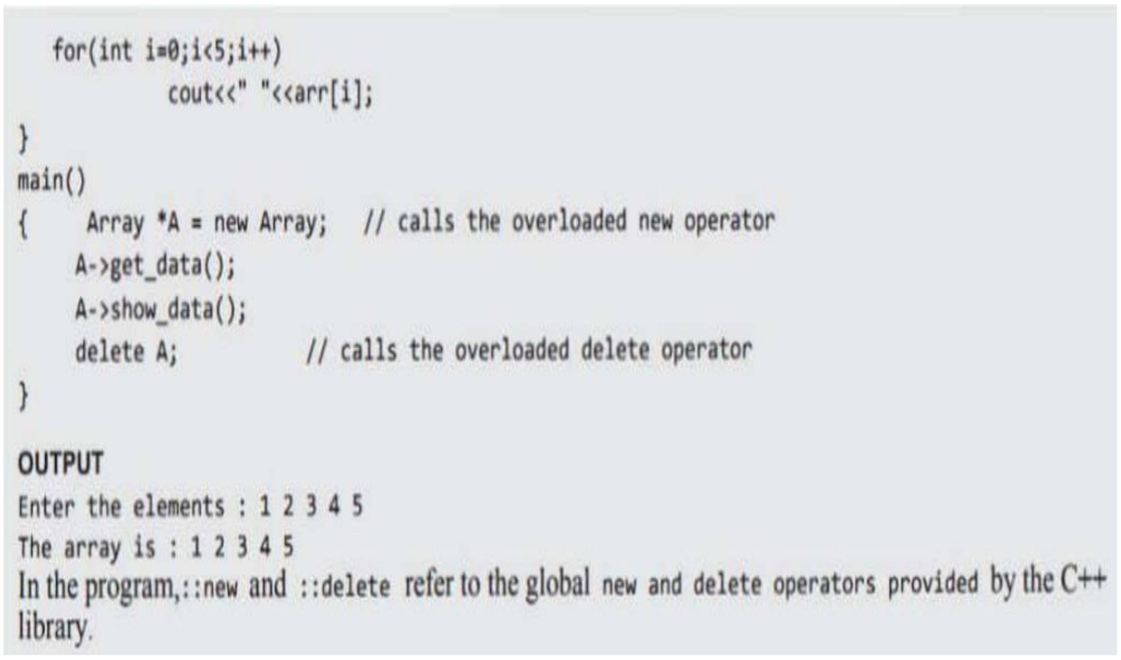
The program code given here demonstrates the use of overloaded new and delete operators.

Example 1:

Overloading new and delete operators



```
using namespace std;
#include<iostream>
class Array
{   private:
        int *arr;
    public:
        void * operator new(size_t size)
        {    void *parr = ::new int[size];  // dynamicallyallocatingspaceusingoverloadednewoperator
             return parr;
             }
             void operator delete(void *parr)
             {     ::delete parr;   }  // dynamicallyde-allocatingspaceusingoverloadeddeleteoperator
                void get_data();
                void show_data();
};
void Array :: get_data()
{   cout<<"\n Enter the elements : ";
   for(int i=0;i<5;i++)
         cin>>arr[i];
}
void Array :: show_data()
{   cout<<"\n The array is : ";
```



```
   for(int i=0;i<5;i++)
            cout<<" "<<arr[i];
}
main()
{     Array *A = new Array;   // calls the overloaded new operator
     A->get_data();
     A->show_data();
     delete A;           // calls the overloaded delete operator
}
```

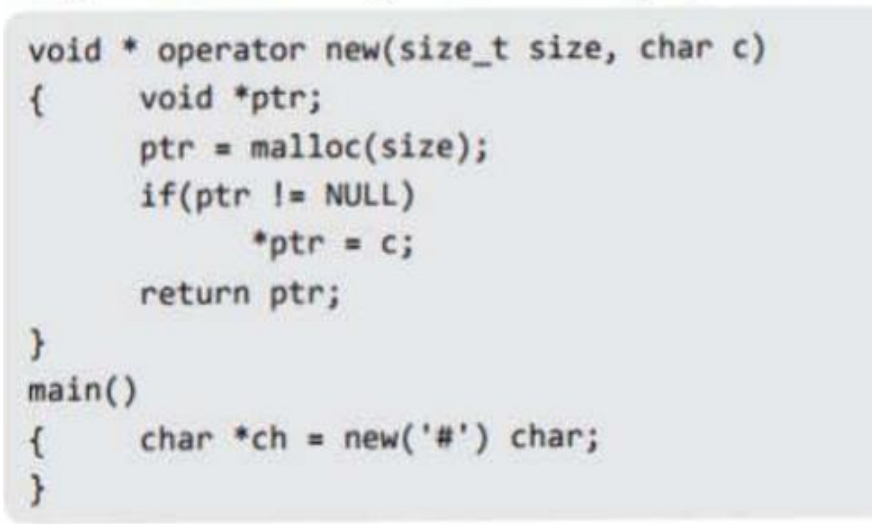
OUTPUT

```
Enter the elements : 1 2 3 4 5
The array is : 1 2 3 4 5
```

In the program,`::new` and `::delete` refer to the global `new` and `delete` operators provided by the C++ library.

Advantages of Overloading

The overloaded new operator function can accept arguments; therefore, a class can have multiple overloaded new operator functions. This gives the programmer more flexibility in customizing memory allocation for objects. For example,



```
void * operator new(size_t size, char c)
{       void *ptr;
        ptr = malloc(size);
        if(ptr != NULL)
                *ptr = c;
        return ptr;
}
main()
{       char *ch = new('#') char;
}
```

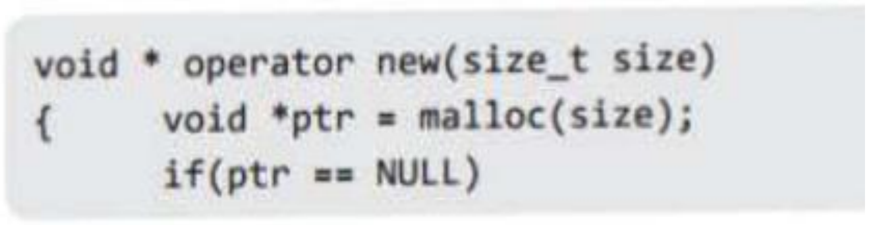
This code will not only allocate memory for a single character but will also initialize the allocated memory with the #character.

The overloaded new operator also enables programmers to squeeze some extra performance out of their programs. For example, in a class, to speed up the allocation of new nodes, a list of deleted nodes is maintained so that their memory can be reused when new nodes are allocated. In this case, the overloaded delete operator will add nodes to the list of deleted nodes and the overloaded new operator will allocate memory from this list rather than from the heap to speedup memory allocation. The global new operator can be used only when the list of deleted nodes is empty.

• Overloaded new or delete operators also provide garbage collection for class's objects.

• Programmers can add exception handling routine in the overloaded new operator function.

• Programmers can use C++ memory allocation functions such as malloc() and realloc() to allocate and re-allocate memory dynamically. For example, the function codes given here allocate, re-allocate, and free memory dynamically.



```
void * operator new(size_t size)
{       void *ptr = malloc(size);
        if(ptr == NULL)
        {     cout<<"\n Memory could not be allocated";
                exit(1);
        }
        else return *ptr;
}
void * realloc(void * ptr, size_t size);
void operator delete(void * ptr)
{       free(ptr);
}
```

The realloc() is used to change the size of the allocated block at address ptr with the size. The address pointed to by ptr may change if the block is shifted to another location in memory. This can happen when size bytes are not available in the previous allocated space. For example, if 10 bytes were allocated and now the user has called realloc() to allocate 20 bytes, then the address pointed by ptr may change to point at the address where 20 bytes are available contiguously. The realloc() returns the new value of ptr. However, if realloc() fails, it returns NULL and does not free the origi- nal memory. Therefore, when using realloc(), you must save the previous pointer value.

Note While a user can have any number of overloaded new operators, there can be only one over-

loaded delete operator. delete is different from delete[].

4.7.2 Overloading Subscript Operators [] and ()

The subscript operator-[]-is used to access array elements and can be overloaded to enhance its functionality with classes. The syntax of overloading the subscript operator can be given as follows:

Identifier [expression]

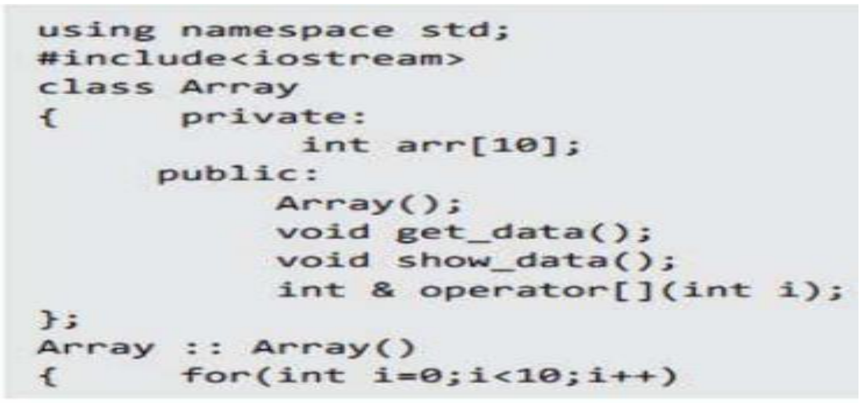
where identifier is the object of the class. The syntax is interpreted as

identifier.operator[] (expression)

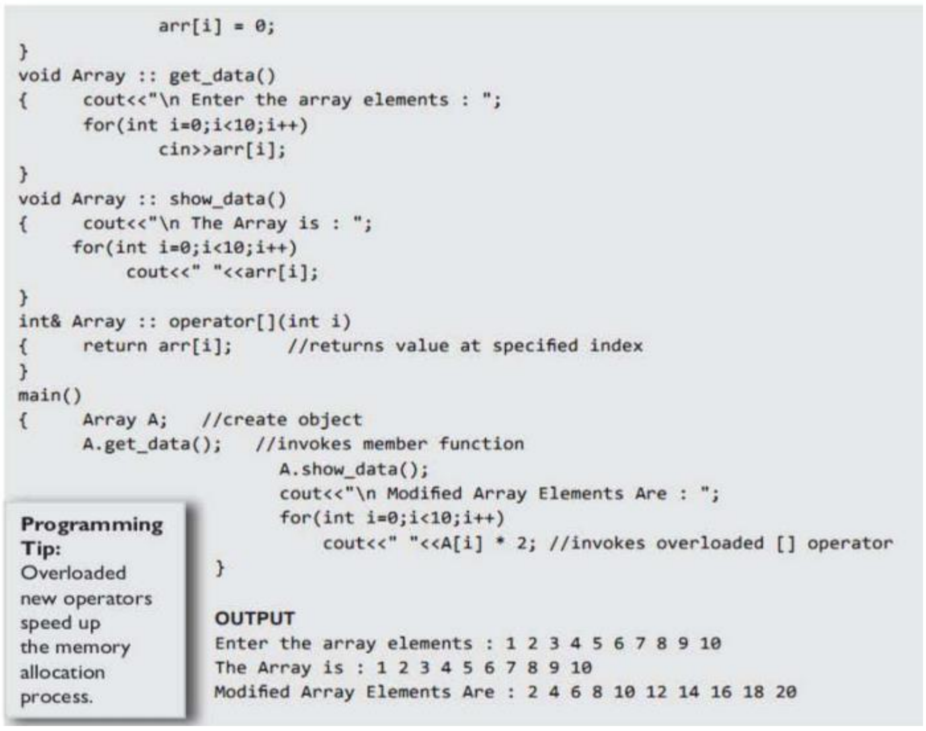
From the syntax, it is clear that the subscript operator is a binary operator in which the first operator is an object of the class and the second operand is an integer index.

Note:

The overloaded operator [] () must be defined as a non-static member function of a **class**.



```
using namespace std;
#include<iostream>
class Array
{       private:
                int arr[10];
        public:
                Array();
                void get_data();
                void show_data();
                int & operator[](int i);
};
Array :: Array()
{       for(int i=0;i<10;i++)
```

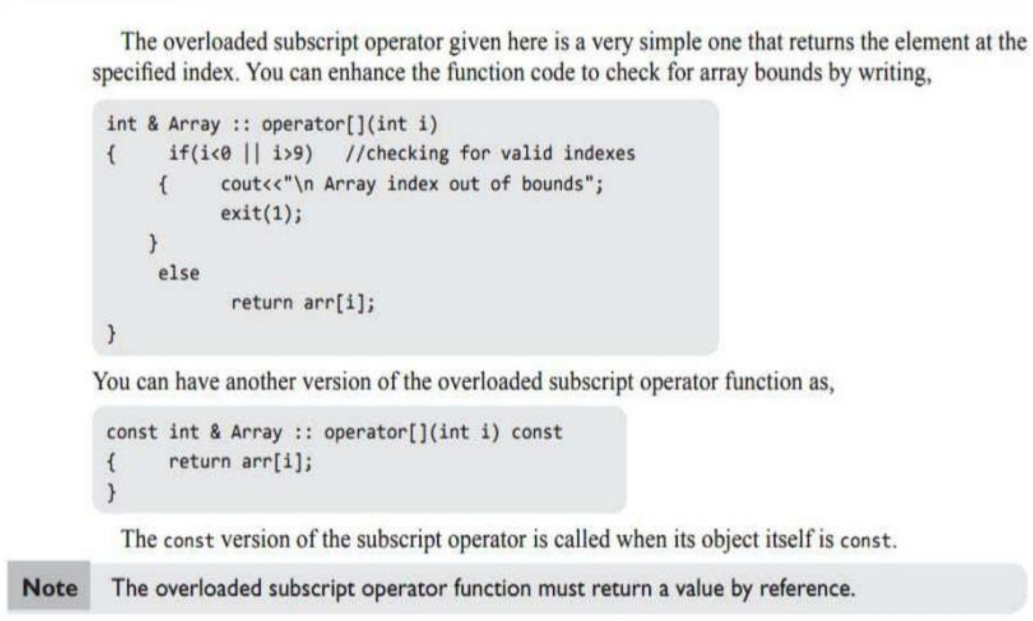


```
                arr[i] = 0;
}
void Array :: get_data()
{       cout<<"\n Enter the array elements : ";
        for(int i=0;i<10;i++)
                cin>>arr[i];
}
void Array :: show_data()
{       cout<<"\n The Array is : ";
      for(int i=0;i<10;i++)
            cout<<" "<<arr[i];
}
int& Array :: operator[](int i)
{       return arr[i];      //returns value at specified index
}
main()
{       Array A;    //create object
        A.get_data();    //invokes member function
                A.show_data();
                cout<<"\n Modified Array Elements Are : ";
                for(int i=0;i<10;i++)
                        cout<<" "<<A[i] * 2; //invokes overloaded [] operator
}
```

Programming Tip: Overloaded new operators speed up the memory allocation process.

OUTPUT

```
Enter the array elements : 1 2 3 4 5 6 7 8 9 10
The Array is : 1 2 3 4 5 6 7 8 9 10
Modified Array Elements Are : 2 4 6 8 10 12 14 16 18 20
```



The overloaded subscript operator given here is a very simple one that returns the element at the specified index. You can enhance the function code to check for array bounds by writing,

```
int & Array :: operator[](int i)
{     if(i<0 || i>9)   //checking for valid indexes
      {     cout<<"\n Array index out of bounds";
            exit(1);
      }
      else
             return arr[i];
}
```

You can have another version of the overloaded subscript operator function as,

```
const int & Array :: operator[](int i) const
{     return arr[i];
}
```

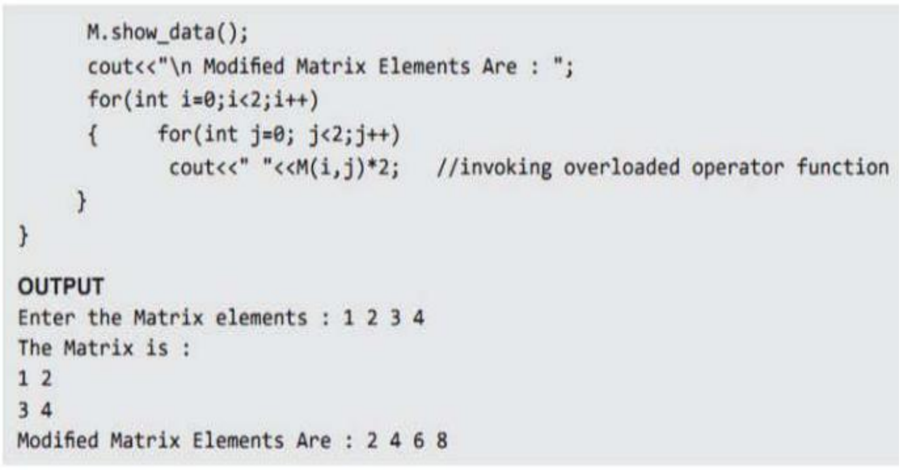
The const version of the subscript operator is called when its object itself is const.

Note The overloaded subscript operator function must return a value by reference.

Overloading Subscript Operator () rather than []

When there are multiple subscripts, it is better to overload the operator() rather than operator[]. This is because the operator[] always takes exactly one parameter, but operator() can take any number of parameters. Let us consider a small piece of code to learn how operator() is overloaded for a two-dimensional matrix.

```
using namespace std;
#include<iostream>
#include<stdlib.h>
class Matrix
{       private:
            int arr[2][2];
        public:
            Matrix();
            void get_data();
            void show_data();
            int& operator()(int i, int j); //()operator overloaded function declaration
};
Matrix :: Matrix()
{       for(int i=0;i<2;i++)
            for(int j=0;j<2;j++)
               arr[i][j] = 0;
}
void Matrix :: get_data()
{       cout<<"\n Enter the Matrix elements : ";
        for(int i=0;i<2;i++)
            for(int j=0;j<2;j++)
                 cin>>arr[i][j];
}
void Matrix :: show_data()
{       cout<<"\n The Matrix is : ";
        for(int i=0;i<2;i++)
        {    cout<<"\n";
              for(int j=0;j<2;j++)
               cout<<" "<<arr[i][j];
        }
}
int & Matrix :: operator()(int i, int j) // overloaded () operator
{      if(i<0 || i>9 || j<0 || j>9)
       {           cout<<"\n Matrix index out of bounds";
                   exit(1);
       }
            else
            return arr[i][j];
}
main()
{       Matrix M;
        M.get_data();
```



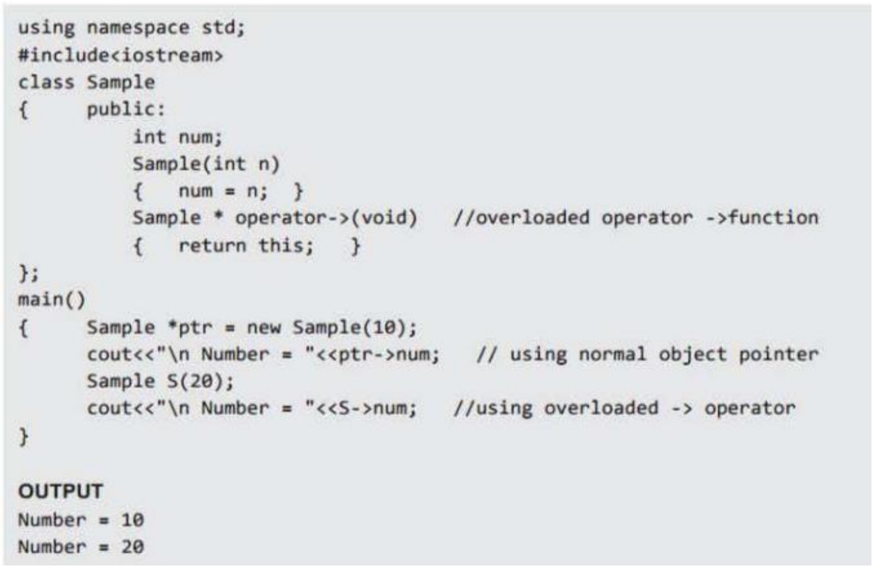
```
        M.show_data();
        cout<<"\n Modified Matrix Elements Are : ";
        for(int i=0;i<2;i++)
        {     for(int j=0; j<2;j++)
                cout<<" "<<M(i,j)*2;   //invoking overloaded operator function
        }
}
OUTPUT
Enter the Matrix elements : 1 2 3 4
The Matrix is :
1 2
3 4
Modified Matrix Elements Are : 2 4 6 8
```

4.7.3. Overloading Class Member Access Operator (->) :

C++ allows programmers to control class member access by overloading the member access oper- ator (->). The -> operator is a unary operator as it takes only one operand, that is the object of the class. The syntax for overloading the -> operator can be given as,

class_ptr *operator->**()**

where class_ptr is a pointer of class type. Before overloading the class member access operator, the overloaded -> operator function must be a non-static member function of the class. The program given here demonstrates the functionality of overloaded member access operator.



```
using namespace std;
#include<iostream>
class Sample
{      public:
           int num;
           Sample(int n)
           {    num = n;  }
           Sample * operator->(void)    //overloaded operator ->function
           {    return this;    }
};
main()
{      Sample *ptr = new Sample(10);
       cout<<"\n Number = "<<ptr->num;    // using normal object pointer
       Sample S(20);
       cout<<"\n Number = "<<S->num;    //using overloaded -> operator
}

OUTPUT
Number = 10
Number = 20
```

6.7.4. Overloading Input and Output Operators:

C++ allows input and output of built-in data types using the stream extraction operator >> and the stream insertion operator <<. The capabilities of these operators can be extended to perform input and output for user-defined types. However, before overloading the extraction and insertion opera- tors, you must know the following aspects:

cin and cout are defined in class iostream.

• While cin is an object of istream class, cout is an object of the ostream class. We will read more on the relationship between istream, ostream, iostream, cin, and cout later in chapter File Handling.

• The insertion and extraction of operators will be overloaded by using a friend func- tion because we need to call these functions without any object.

. The insertion and extraction operators must return the value of the left operand-the ostream or istream object-so that multiple << or >> operators may be used in the same statement. The syntax for overloading the >> function can be given as

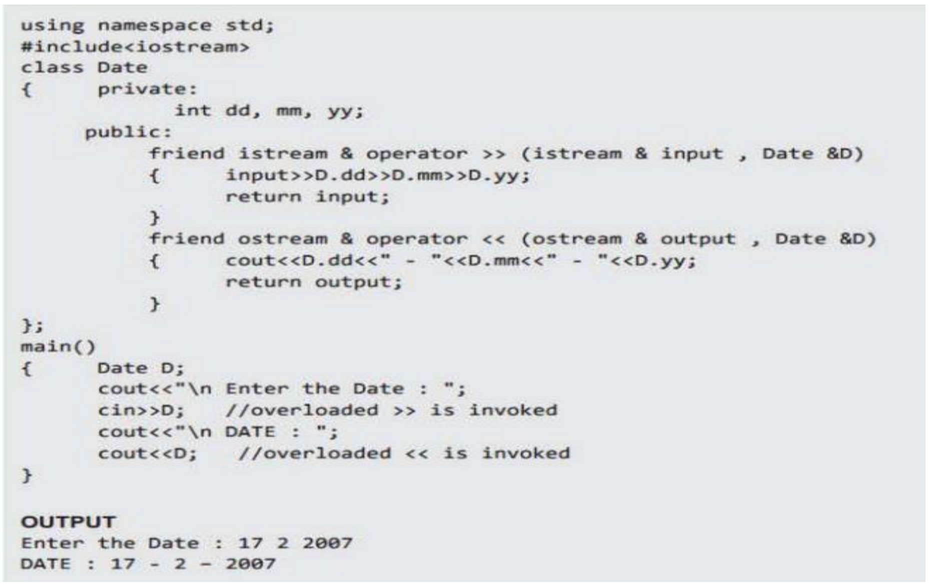
friend ostream & operator << **(**ostream & output, My_class&obj)

{ // code

}

Example 1:

Overloading<<and >>operators



```
using namespace std;
#include<iostream>
class Date
{       private:
                int dd, mm, yy;
     public:
          friend istream & operator >> (istream & input , Date &D)
          {       input>>D.dd>>D.mm>>D.yy;
                  return input;
          }
          friend ostream & operator << (ostream & output , Date &D)
          {       cout<<D.dd<<" - "<<D.mm<<" - "<<D.yy;
                  return output;
          }
};
main()
{       Date D;
        cout<<"\n Enter the Date : ";
        cin>>D;    //overloaded >> is invoked
        cout<<"\n DATE : ";
        cout<<D;    //overloaded << is invoked
}

OUTPUT
Enter the Date : 17 2 2007
DATE : 17 - 2 - 2007
```

Exercises:

1. Define the term 'operator overloading' and explain its significance.

2. How does operator overloading support the con-cept of polymorphism?

3. Can we perform function overloading on operator overloaded operators? If yes, how?

4. What are the advantages of overloading the operators?

5. Explain the syntax of overloaded operator func-tion defined as a class member function.

6. Give the syntax of overloaded operator function defined as a friend function.

7. List at least five operators that can be overloaded. Give the implementation of any one of them in detail.

8. List the operators that cannot be overloaded.

9. What advantage do we get when operator over- loaded function returns an object?

10. What are the two techniques in which an operator can be overloaded?

Unit V

Introduction, Defining Derived classes, Single inheritance, Making a private member in heritable, Multi-level in heritance, Multiple inheritance, Hierarchical inheritance, Hybrid inheritance

Chapter 5: Sections 5.1-5.8

5.1. Introduction:

Reusability is an important feature of object oriented programming (OOP). Reusing an existing piece of code offers manifold benefits. It not only saves the effort and cost required to build a software product but also enhances its reliability. Therefore, programmers need not re-write, re-debug, and re-test the code that has already been tested and being used in existing software. To support reusability, C++ supports the concept of reusing existing classes as it allows programmers to create new classes that reuse prewritten and tested classes. The existing classes are adapted as per user's requirements so that the newly formed classes can be incorporated in the current software application being developed. The technique of creating a new class from an existing class is called inheritance. The old or existing class is called the base class and the new class is known as the derived class or sub-class. The derived classes are created by first inheriting the data and methods of the base class and then adding new specialized data and functions in it. During the process of inheritance, the base class remains unchanged. The concept of inheritance is, therefore, frequently used to implement the is-a relationship. For example, teacher is-a person, student is-a person; while both teacher and student are a person in the first place, both also have some distinguishing features. Therefore, all the common traits of teacher and student are specified in the Person class and specialized features are incorporated in two separate classes of Teacher and Student. Similarly, a dentist or a surgeon is a doctor and doctor is a person. Fig. 5.1 illustrates the concept of inheritance which follows a top-down approach to problem solving. In top-down design approach, generalized classes are designed first and specialized classes are derived by inheriting or extending generalized classes.

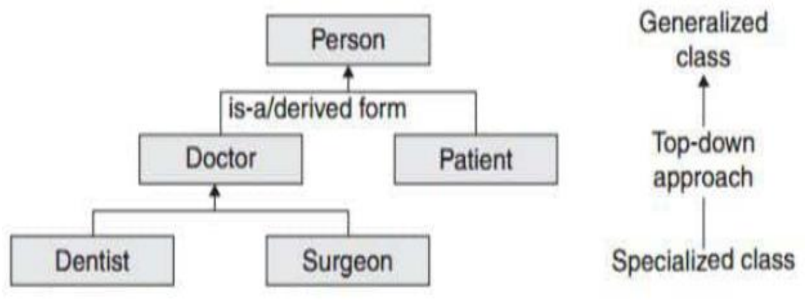
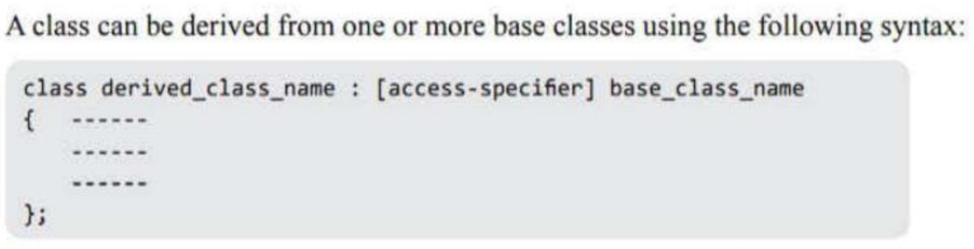


Figure 5.1. Is-a relationship between classes

5.2.Defining Derived Classes:

A class can be derived from one or more base classes using the following syntax:



A class can be derived from one or more base classes using the following syntax:

```
class derived_class_name : [access-specifier] base_class_name
{   ------
    ------
    ------
};
```

According to the syntax, the derived_class_name is derived from the base_class_name, thereby inheriting some or all of its members. The access-specifier, also known as the visibility-mode, is optional and if present, can be either public, private, or protected. If no access-specifier is written, then by default, the class will be derived in private mode.

5.3.Making a private member in heritable:

Private:

Private is the highest level of data hiding. When a base class is privately inherited by a derived class, then public members of the base class become private members of the derived class. The access to private members cannot be inherited but the derived class can access them using public member functions of the base class. This means that the object of a privately inherited class cannot access the inherited members.

Note:

Private members of the base class can never become members of the derived class.

Inheriting Protected Members

When a protected member is inherited in private mode, the protected member of the base class becomes private member of the derived class. Although these members can be accessed in the derived class, these are not available for further inheritance (in case of multi-level inheritance) because private members cannot be inherited as shown in Fig. 5.2.

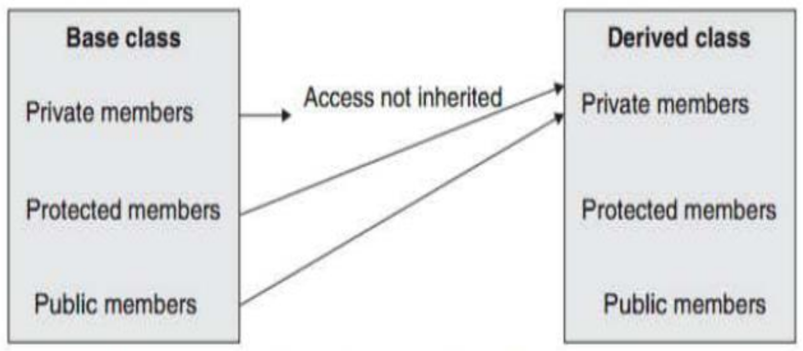


Figure.5.2. A class derived in private mode

5.4. Single Inheritance:

When a derived class inherits features from a single base class, it is called single inheritance as given in Fig. 5.3. Consider the program given here that uses single inheritance to display the details of a student.

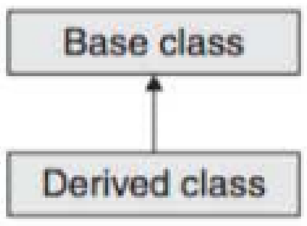


Figure 5.3. Single inheritance

Example 5.1. Single inheritance

```
using namespace std;
#include<iostream>
class Student
{   private:
        int roll_no;
    protected:
        char course[10];
    public:
        void get_rno()
        {   cout<<"\n Enter the roll number : ";
            cin>>roll_no;
```

```
        }
        void show_rno()
        {   cout<<"\n ROLL NO : "<<roll_no;
        }
};
class Result : public Student
{   private:
        int marks[3];
    public:
        void get_data();
        int total();
        void display()
        {   show_rno();
            cout<<"\n COURSE : "<<course;
            cout<<"\n TOTAL MARKS : "<<total();
        }
};
void Result :: get_data()
{   get_rno();
    cout<<"\n Enter the course : ";
    cin.ignore();
    cin.getline(course,10);
    cout<<"\n Enter marks in three subjects : ";
    for(int i=0;i<3;i++)
        cin>>marks[i];
}
int Result :: total()
{       int tot_marks = 0;
    for(int i=0;i<3;i++)
        tot_marks += marks[i];
    return tot_marks;
}
main()
{   Result R;
    R.get_data();
    R.display();
    //R.get_rno();                  //Ok
    //R.roll_no = 12;               // not allowed
    //strcpy(R.course, "BCA") ;     // not allowed
}
```

OUTPUT

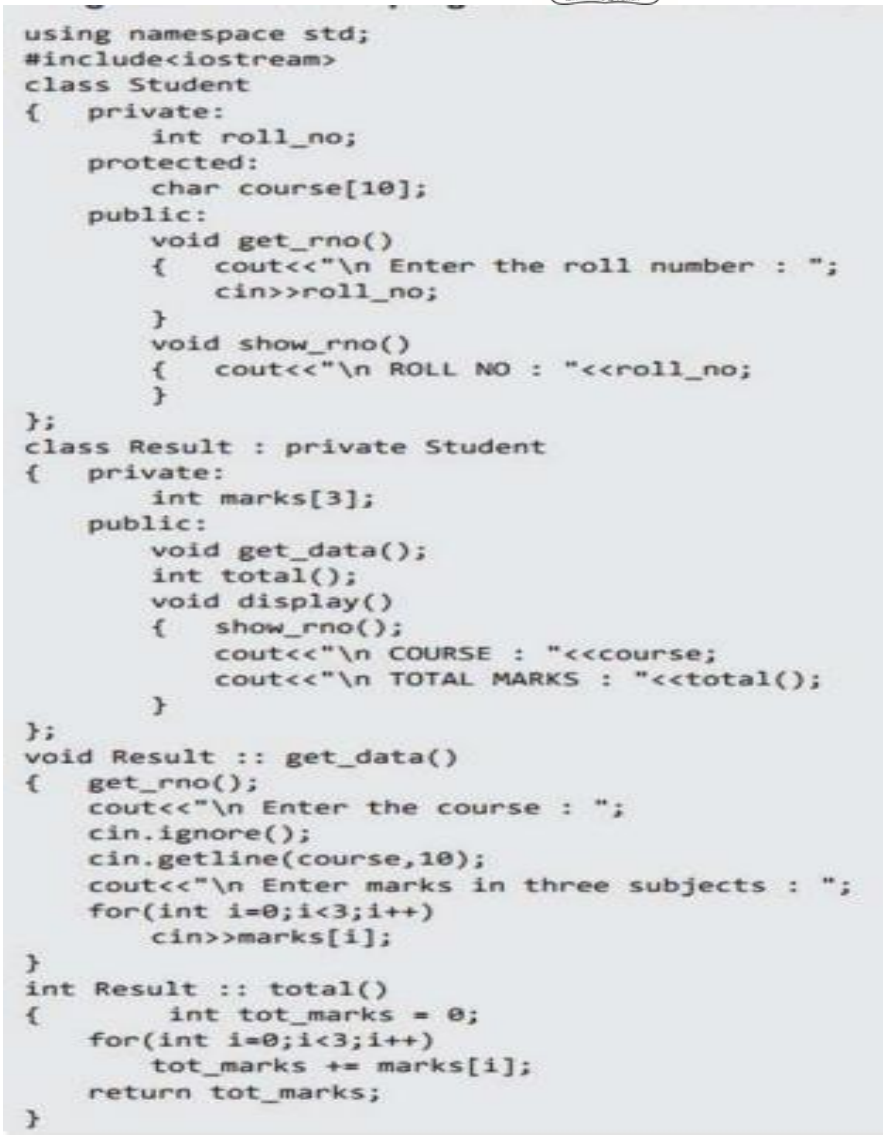
```
Enter the roll number : 1234
Enter the course : BCA
Enter marks in three subjects :   67 78 89
ROLL NO : 1234
COURSE : BCA
TOTAL MARKS : 234
```

Explanation:

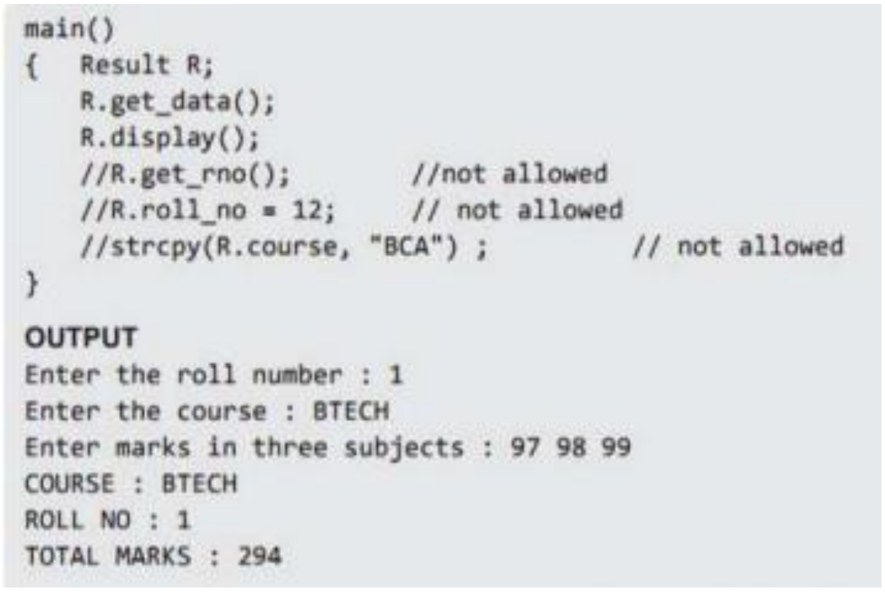
In the program, there are two classes- Student and Result. The class Student hold information about student (roll number, course) and has functions to read and display the roll number. The class Result stores the marks of the student in three subjects, calculates the total marks, and displays all the information.

Note that Result is derived in public from class Student. Therefore, private member of ro1l_no cannot be accessed, protected member of course becomes protected member of Result, and the two public member functions becomes public members of the derived class. This means that all the members of the base class except the ro11_no are accessible by the derived class. In the get_data (), from Result class you can directly access course and get_rno () but not the ro11_ no. To access the ro11_no you have to access the private member indirectly through a public member function of the class. In main (), the object of the derived class can access the public member function of the base class but not the private or the protected members.

Program 5.1. Write a program in which class result inherits students in private mode.



```
using namespace std;
#include<iostream>
class Student
{   private:
        int roll_no;
    protected:
        char course[10];
    public:
        void get_rno()
        {   cout<<"\n Enter the roll number : ";
            cin>>roll_no;
        }
        void show_rno()
        {   cout<<"\n ROLL NO : "<<roll_no;
        }
};
class Result : private Student
{   private:
        int marks[3];
    public:
        void get_data();
        int total();
        void display()
        {   show_rno();
            cout<<"\n COURSE : "<<course;
            cout<<"\n TOTAL MARKS : "<<total();
        }
};
void Result :: get_data()
{   get_rno();
    cout<<"\n Enter the course : ";
    cin.ignore();
    cin.getline(course,10);
    cout<<"\n Enter marks in three subjects : ";
    for(int i=0;i<3;i++)
        cin>>marks[i];
}
int Result :: total()
{        int tot_marks = 0;
    for(int i=0;i<3;i++)
        tot_marks += marks[i];
    return tot_marks;
}
```

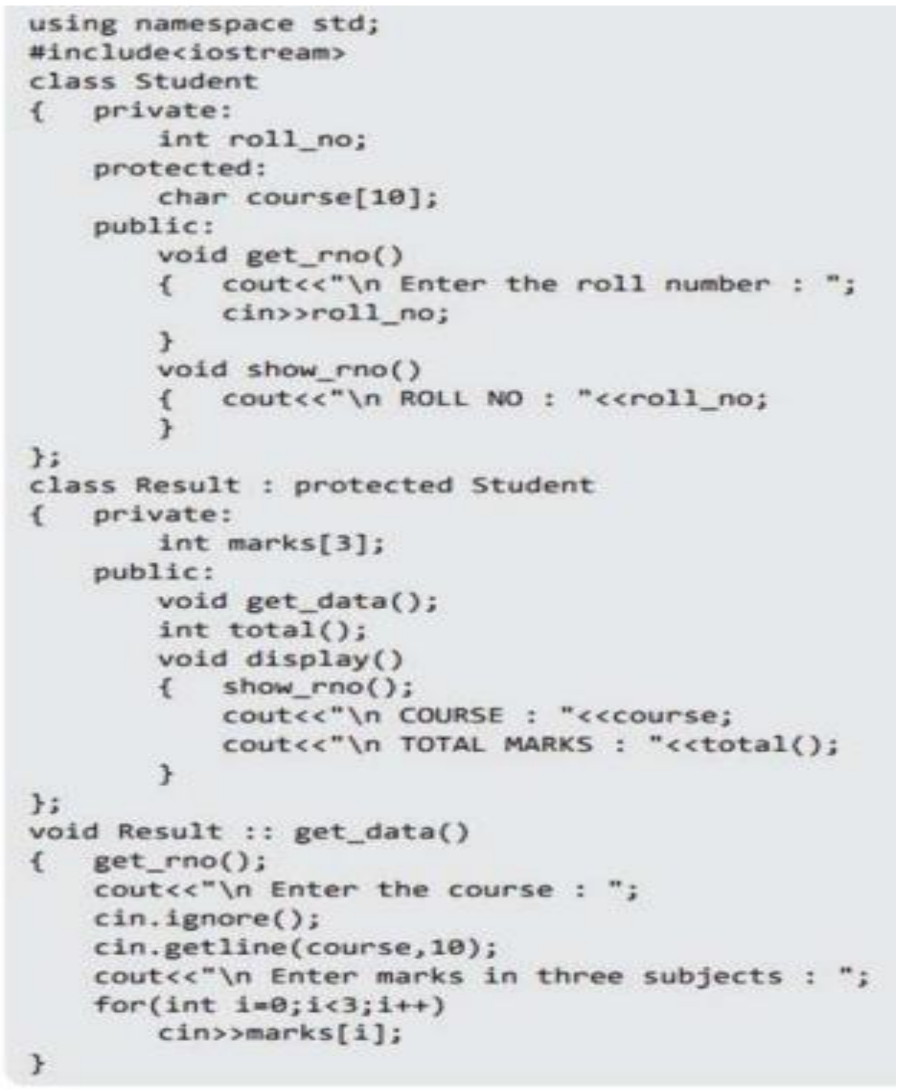


```
main()
{   Result R;
    R.get_data();
    R.display();
    //R.get_rno();        //not allowed
    //R.roll_no = 12;     // not allowed
    //strcpy(R.course, "BCA") ;          // not allowed
}
```

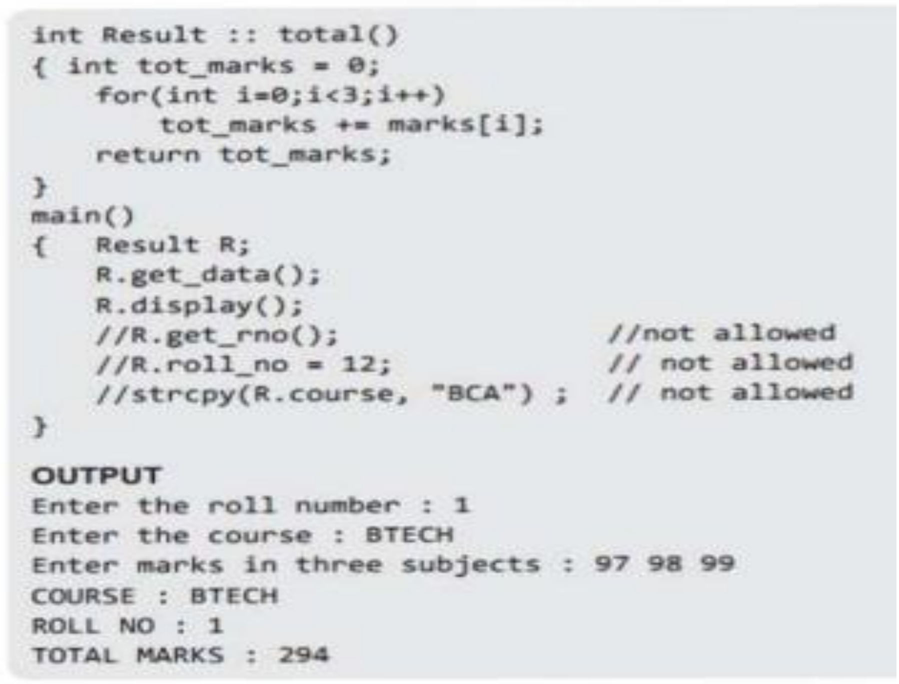
OUTPUT

```
Enter the roll number : 1
Enter the course : BTECH
Enter marks in three subjects : 97 98 99
COURSE : BTECH
ROLL NO : 1
TOTAL MARKS : 294
```

Program 5.2. Write a program in which class result inherits students in protected mode.



```
using namespace std;
#include<iostream>
class Student
{   private:
        int roll_no;
    protected:
        char course[10];
    public:
        void get_rno()
        {   cout<<"\n Enter the roll number : ";
            cin>>roll_no;
        }
        void show_rno()
        {   cout<<"\n ROLL NO : "<<roll_no;
        }
};
class Result : protected Student
{   private:
        int marks[3];
    public:
        void get_data();
        int total();
        void display()
        {   show_rno();
            cout<<"\n COURSE : "<<course;
            cout<<"\n TOTAL MARKS : "<<total();
        }
};
void Result :: get_data()
{   get_rno();
    cout<<"\n Enter the course : ";
    cin.ignore();
    cin.getline(course,10);
    cout<<"\n Enter marks in three subjects : ";
    for(int i=0;i<3;i++)
        cin>>marks[i];
}
```



```
int Result :: total()
{ int tot_marks = 0;
    for(int i=0;i<3;i++)
        tot_marks += marks[i];
    return tot_marks;
}
main()
{   Result R;
    R.get_data();
    R.display();
    //R.get_rno();                  //not allowed
    //R.roll_no = 12;               // not allowed
    //strcpy(R.course, "BCA") ;     // not allowed
}

OUTPUT
Enter the roll number : 1
Enter the course : BTECH
Enter marks in three subjects : 97 98 99
COURSE : BTECH
ROLL NO : 1
TOTAL MARKS : 294
```

5.5. Multi-Level Inheritance:

The technique of deriving a class from an already derived class is called multi-level inheritance. In Fig. 5.4, base class acts as the base for derived class 1, which in turn, acts as a base for derived class 2. Therefore, derived class 1 is known as the intermediate base class as this class provides a link for inheritance between the base class and the derived class 2. The chain of classes from base class ->derived class 1 ->derived class 2 is known as the inheritance path. In multilevel inheritance, the number of levels can go up to any number based on the requirement. Consider the same program with multi-level inheritance.

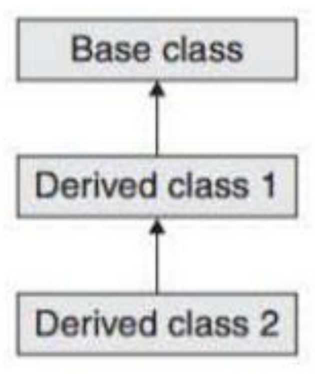
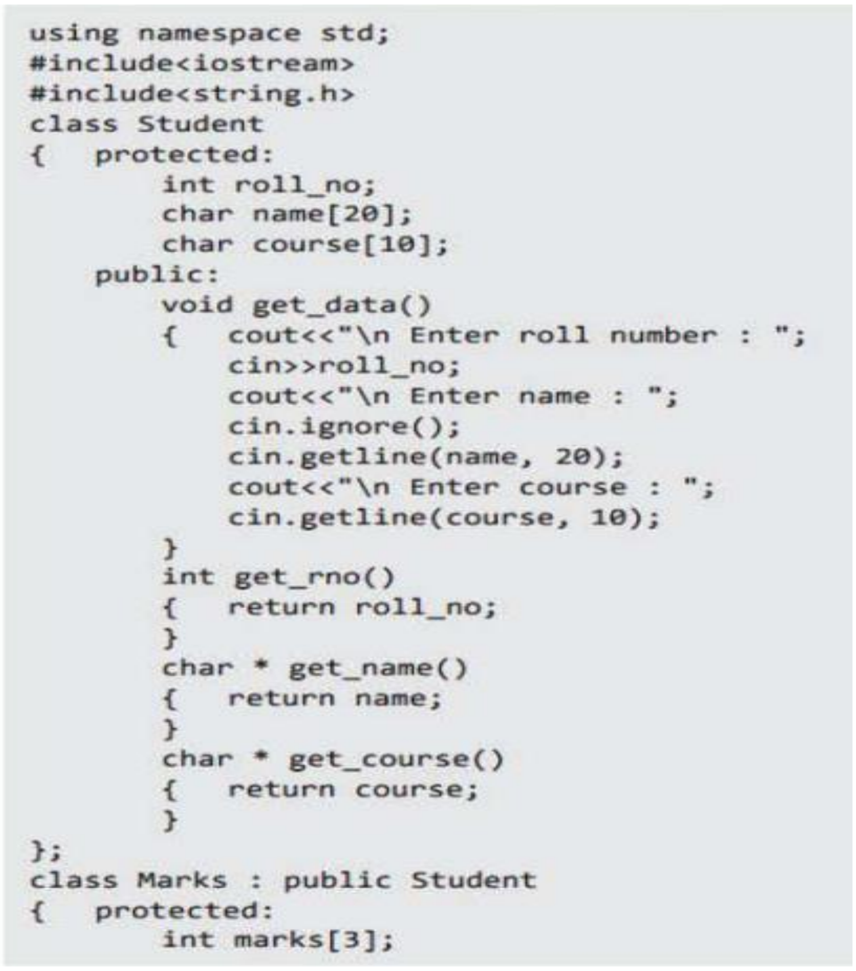


Figure 5.4. Multi-level inheritance

Example 1. Multi-level inheritance



```
using namespace std;
#include<iostream>
#include<string.h>
class Student
{   protected:
        int roll_no;
        char name[20];
        char course[10];
    public:
        void get_data()
        {   cout<<"\n Enter roll number : ";
            cin>>roll_no;
            cout<<"\n Enter name : ";
            cin.ignore();
            cin.getline(name, 20);
            cout<<"\n Enter course : ";
            cin.getline(course, 10);
        }
        int get_rno()
        {   return roll_no;
        }
        char * get_name()
        {   return name;
        }
        char * get_course()
        {   return course;
        }
};
class Marks : public Student
{   protected:
        int marks[3];
    public:
        void get_marks()
        {   cout<<"\n Enter marks in three subjects : ";
            cin>>marks[0]>>marks[1]>>marks[2];
        }
        int total()
        {   return (marks[0] + marks[1] + marks[2]);
        }
};
class Result : public Marks
{   public:
        void display()
        {   cout<<"\n ROLL NUMBER : "<<get_rno();
            cout<<"\n NAME : "<<get_name();
            cout<<"\n COURSE : "<<get_course();
            cout<<"\n TOTAL MARKS : "<<total();
        }
};
main()
{   Result R;
    R.get_data();
    R.get_marks();
    R.display();
}
```

OUTPUT

```
Enter roll number : 1234
Enter name : Aditi Raj
Enter course : MCA
Enter marks in three subjects : 99 98 100
ROLL NUMBER : 1234
NAME : Aditi Raj
COURSE : MCA
TOTAL MARKS : 297
```

Explanation:

In the program, class Result is derived from Student in public mode. It can use all the members of the class, except private members. However, these private members are being used through the public member function total ().

5.6. Multiple Inheritance:

When a derived class inherits features from more than one base class shown in Fig. 5.5, it is called multiple inheritance. The syntax for multiple inheritance can be given as follows:

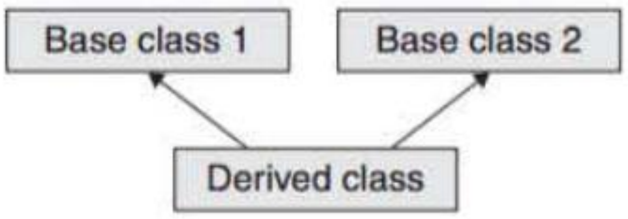


Figure 5.5. Multiple inheritance

```
class derived_class_name : visibility mode base1_
class_name, visibility mode base1_class_
```

```
name, ...... visibility mode basen_class_name
{
    --------
}
```

5.7. Hierarchical Inheritance:

When a class is inherited by more than one class, it is called hierarchical inheritance as shown in Fig. 5.6. This type of inheritance is usually implemented when certain features at one level are to be shared by many features at the next level. Some examples are as follows.

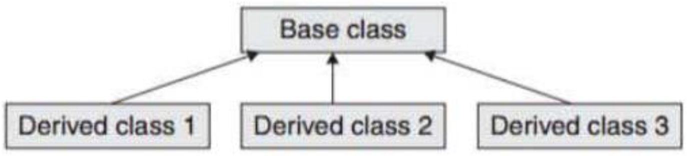
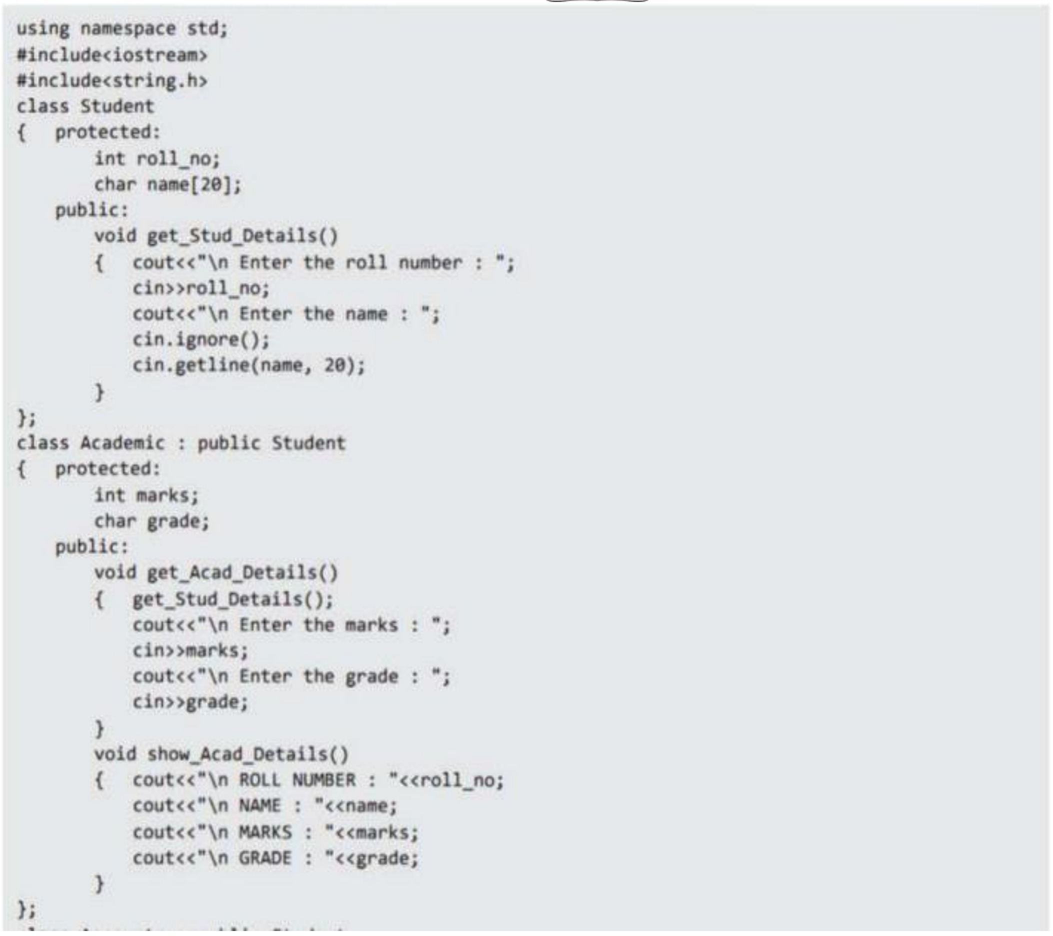


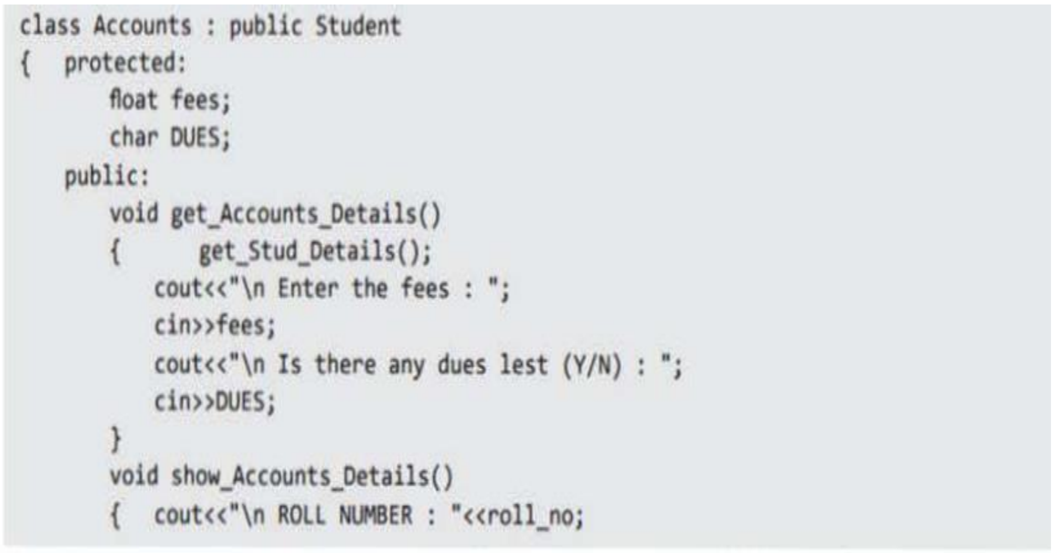
Figure 5.6. Hierarchical inheritance

- Base class account can be inherited by three classes such as savings account, fixed deposit account, and current account.
- Student class can be inherited by three separate classes such as science, commerce, and humanities
- Faculty class can be inherited by n number of departments such as commerce, mathematics, computer science, electronics, and so on.

In all these examples, a class hierarchy is formed in which the base class will include all the features that are common to the derived classes. The derived class can inherit the features of the base class and extend its features to better suit the requirements of the problem at hand. Consider the program given here in which the class Student acts as the base class for two classes-academic details and accounts details.



```
using namespace std;
#include<iostream>
#include<string.h>
class Student
{   protected:
        int roll_no;
        char name[20];
    public:
        void get_Stud_Details()
        {   cout<<"\n Enter the roll number : ";
            cin>>roll_no;
            cout<<"\n Enter the name : ";
            cin.ignore();
            cin.getline(name, 20);
        }
};
class Academic : public Student
{   protected:
        int marks;
        char grade;
    public:
        void get_Acad_Details()
        {   get_Stud_Details();
            cout<<"\n Enter the marks : ";
            cin>>marks;
            cout<<"\n Enter the grade : ";
            cin>>grade;
        }
        void show_Acad_Details()
        {   cout<<"\n ROLL NUMBER : "<<roll_no;
            cout<<"\n NAME : "<<name;
            cout<<"\n MARKS : "<<marks;
            cout<<"\n GRADE : "<<grade;
        }
};
```



```
class Accounts : public Student
{   protected:
        float fees;
        char DUES;
    public:
        void get_Accounts_Details()
        {       get_Stud_Details();
            cout<<"\n Enter the fees : ";
            cin>>fees;
            cout<<"\n Is there any dues lest (Y/N) : ";
            cin>>DUES;
        }
        void show_Accounts_Details()
        {   cout<<"\n ROLL NUMBER : "<<roll_no;
```

```
            cout<<"\n NAME : "<<name;
            cout<<"\n FEES : "<<fees;
            cout<<"\n DUES : "<<DUES;
        }
};
main()
{   Accounts AD;
    AD.get_Accounts_Details();
    AD.show_Accounts_Details();
    Academic AcD;
    AcD.get_Acad_Details();
    AcD.show_Acad_Details();
}
```

OUTPUT

```
Enter the roll number : 1239
Enter the name : Saisha
Enter the fees :45920
Is there any dues lest (Y/N) : N
ROLL NUMBER : 1239
NAME : Saisha
FEES : 45920
DUES : N
Enter the roll number : 809
Enter the name : Dev
Enter the marks : 91
Enter the grade : O
ROLL NUMBER : 809
NAME : Dev
MARKS : 91
GRADE : O
```

Explanation:

The program shows that academic details are stored in a separate class and the accounts details are stored in a separate class. The program uses inheritance to display all the information about a student. Such type programs increases re-usability of codes.

5.8. Hybrid Inheritance:

Deriving a class that involves more than one form of inheritance is called hybrid inheritance. For example, in Fig. 5.6, derived class 3 is derived from base class using multi-level as well as hierarchical inheritance. Let us rewrite the program on student's result to implement hybrid inheritance. In the program, there are two classes-Marks and Activities. The class result will inherit both these classes and generate result. Finally, the result class would act as the base class for student which will display the entire information about a particular user.

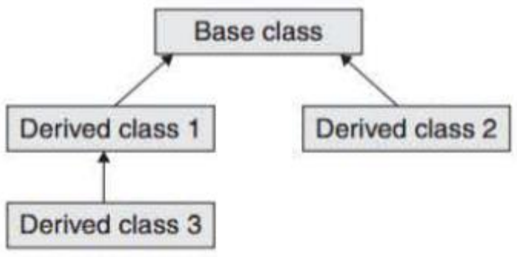


Figure 5.7. Hybrid inheritance

Example 1: Hybrid inheritance

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

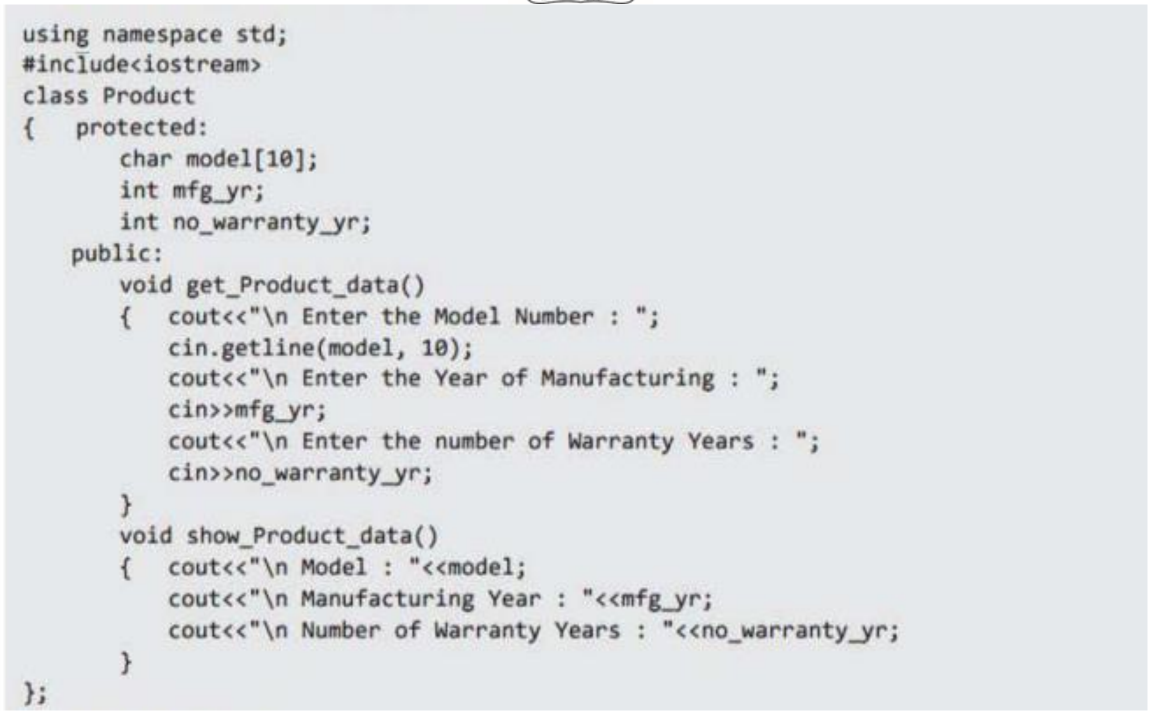
```
class Marks
{   protected:
        int marks[3];
    public:
        void get_test_marks()
        {   cout<<"\n Enter the marks in three subjects : ";
            cin>>marks[0]>>marks[1]>>marks[2];
        }
        int total_test_marks()
        {   return (marks[0] + marks[1] + marks[2]);
        }
};
class Activities
{   protected:
        int acts[2];
    public:
        void get_act_marks()
        {   cout<<"\n Enter marks in two activities : ";
            cin>>acts[0]>>acts[1];
        }
        int total_act_marks()
        {   return (acts[0] + acts[1]);
        }
};
```

```
class Student : public Marks, public Activities
{   protected:
        int roll_no;
        char name[20];
    public:
        void get_details()
        {   cout<<"\n Enter the roll number : ";
            cin>>roll_no;
            cout<<"\n Enter the name : ";
            cin.ignore();
            cin.getline(name, 20);
            get_test_marks();
            get_act_marks();
        }
};
class Result : public Student
{   public:
        void get_data()
        {   get_details();   }
        void show_data()
        {   cout<<"\n ROLL NUMBER : "<<roll_no;
            cout<<"\n NAME : "<<name;
            cout<<"\n TOTAL MARKS : "<<total_test_marks() + total_act_marks();
        }
};
main()
{   Result R;
    R.get_data();
    R.show_data();
}
```

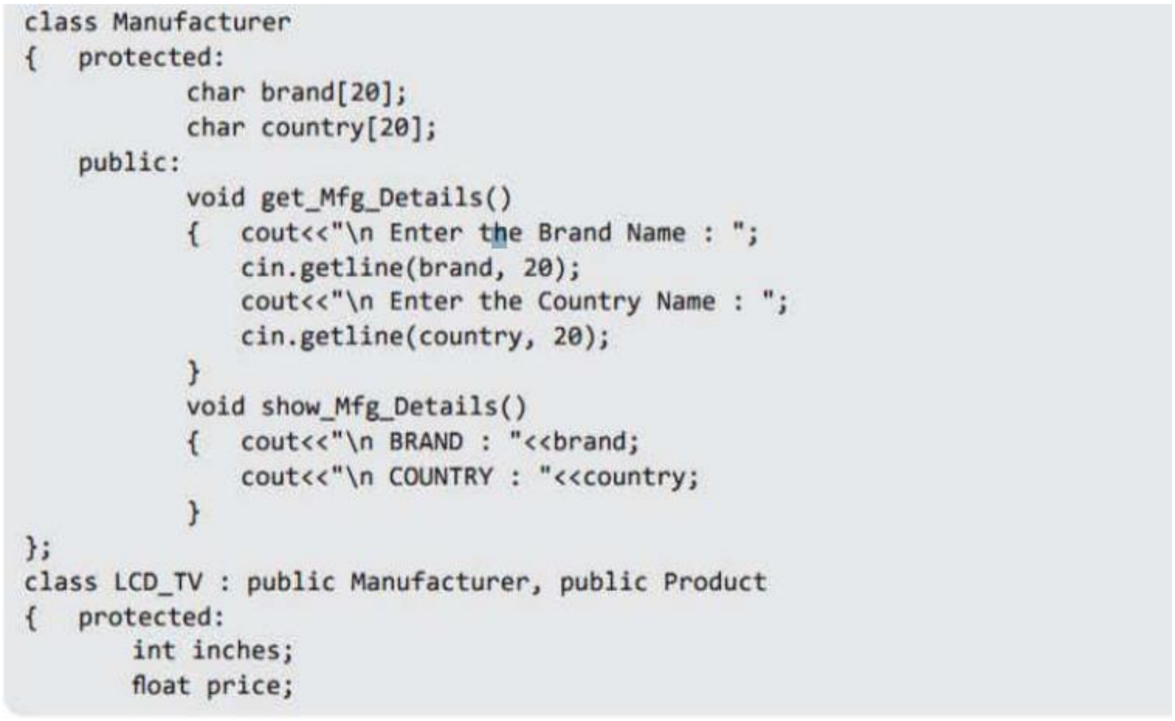
```
OUTPUT
Enter the roll number : 1234
Enter the name : Anushka Lamba
Enter the marks in three subjects : 98 97 87
Enter marks in two activities : 92 85
ROLL NUMBER : 1234
NAME : Anushka Lamba
TOTAL MARKS : 459
```

Problem 2:

Write a program using multiple inheritance that allows class LCD_TV to inherit two classes-product and manufacturer.



```
using namespace std;
#include<iostream>
class Product
{   protected:
        char model[10];
        int mfg_yr;
        int no_warranty_yr;
    public:
        void get_Product_data()
        {   cout<<"\n Enter the Model Number : ";
            cin.getline(model, 10);
            cout<<"\n Enter the Year of Manufacturing : ";
            cin>>mfg_yr;
            cout<<"\n Enter the number of Warranty Years : ";
            cin>>no_warranty_yr;
        }
        void show_Product_data()
        {   cout<<"\n Model : "<<model;
            cout<<"\n Manufacturing Year : "<<mfg_yr;
            cout<<"\n Number of Warranty Years : "<<no_warranty_yr;
        }
};
```



```
class Manufacturer
{   protected:
            char brand[20];
            char country[20];
    public:
            void get_Mfg_Details()
            {   cout<<"\n Enter the Brand Name : ";
                cin.getline(brand, 20);
                cout<<"\n Enter the Country Name : ";
                cin.getline(country, 20);
            }
            void show_Mfg_Details()
            {   cout<<"\n BRAND : "<<brand;
                cout<<"\n COUNTRY : "<<country;
            }
};
class LCD_TV : public Manufacturer, public Product
{   protected:
        int inches;
        float price;
    public:
        void get_data()
        {   get_Mfg_Details();
            get_Product_data();
            cout<<"\n Enter the Size : ";
            cin>>inches;
            cout<<"\n Enter the Price : ";
            cin>>price;
        }
        void show_data()
        {   show_Mfg_Details();
            show_Product_data();
            cout<<"\n SIZE : "<<inches;
            cout<<"\n PRICE : "<<price;
        }
};
main()
{   LCD_TV TV;
    TV.get_data();
    TV.show_data();
}
```

OUTPUT

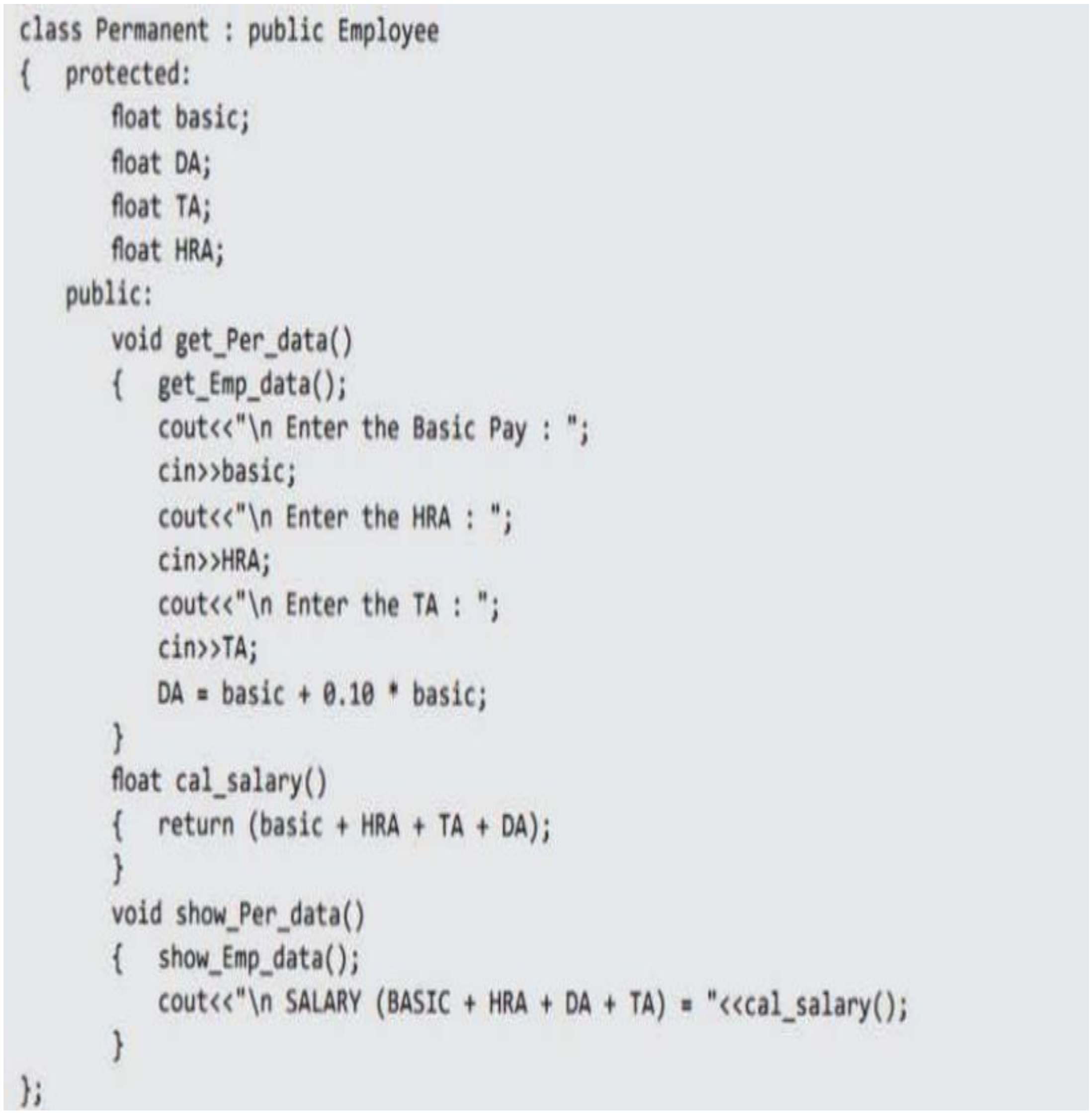
```
Enter the Brand Name : Samsung
Enter the Country Name : Japan
Enter the Model Number : S102
Enter the Year of Manufacturing : 2014
Enter the number of Warranty Years : 3
Enter the Size : 32
Enter the Price : 35000
BRAND : Samsung
COUNTRY : Japan
Model : S102
Manufacturing Year : 2014
Number of Warranty Years : 3
SIZE : 32
PRICE : 35000
```

Problem 3:

Write a program that has a class employee inherited by two classes-contract and permanent.

```
using namespace std;
#include<iostream>
class Employee
{   protected:
        int id;
        char name[20];
        char desig[20];
    public:
        void get_Emp_data()
        {   cout<<"\n Enter Employee ID : ";
            cin>>id;
            cout<<"\n Enter the Name : ";
            cin.ignore();
            cin.getline(name, 20);
```

```
            cout<<"\n Enter the Designation : ";
            cin.getline(desig, 20);
        }
        void show_Emp_data()
        {   cout<<"\n EMP ID : "<<id;
            cout<<"\n NAME : "<<name;
            cout<<"\n DESIGNATION : "<<desig;
        }
};
class Contract : public Employee
{   protected:
        int num_hrs;
        float wages_perhr;
    public:
        void get_Cont_data()
        {   get_Emp_data();
            cout<<"\n Enter the number of hours : ";
            cin>>num_hrs;
            cout<<"\n Enter the wages per hour : ";
            cin>>wages_perhr;
        }
        void show_Cont_data()
        {   show_Emp_data();
            cout<<"\n NUMBER OF HOURS WORKED : "<<num_hrs;
            cout<<"\n WAGES PER HOUR : "<<wages_perhr;
            cout<<"\n SALARY : "<<num_hrs * wages_perhr;
        }
};
```



```cpp
class Permanent : public Employee
{   protected:
        float basic;
        float DA;
        float TA;
        float HRA;
    public:
        void get_Per_data()
        {   get_Emp_data();
            cout<<"\n Enter the Basic Pay : ";
            cin>>basic;
            cout<<"\n Enter the HRA : ";
            cin>>HRA;
            cout<<"\n Enter the TA : ";
            cin>>TA;
            DA = basic + 0.10 * basic;
        }
        float cal_salary()
        {   return (basic + HRA + TA + DA);
        }
        void show_Per_data()
        {   show_Emp_data();
            cout<<"\n SALARY (BASIC + HRA + DA + TA) = "<<cal_salary();
        }
};
```

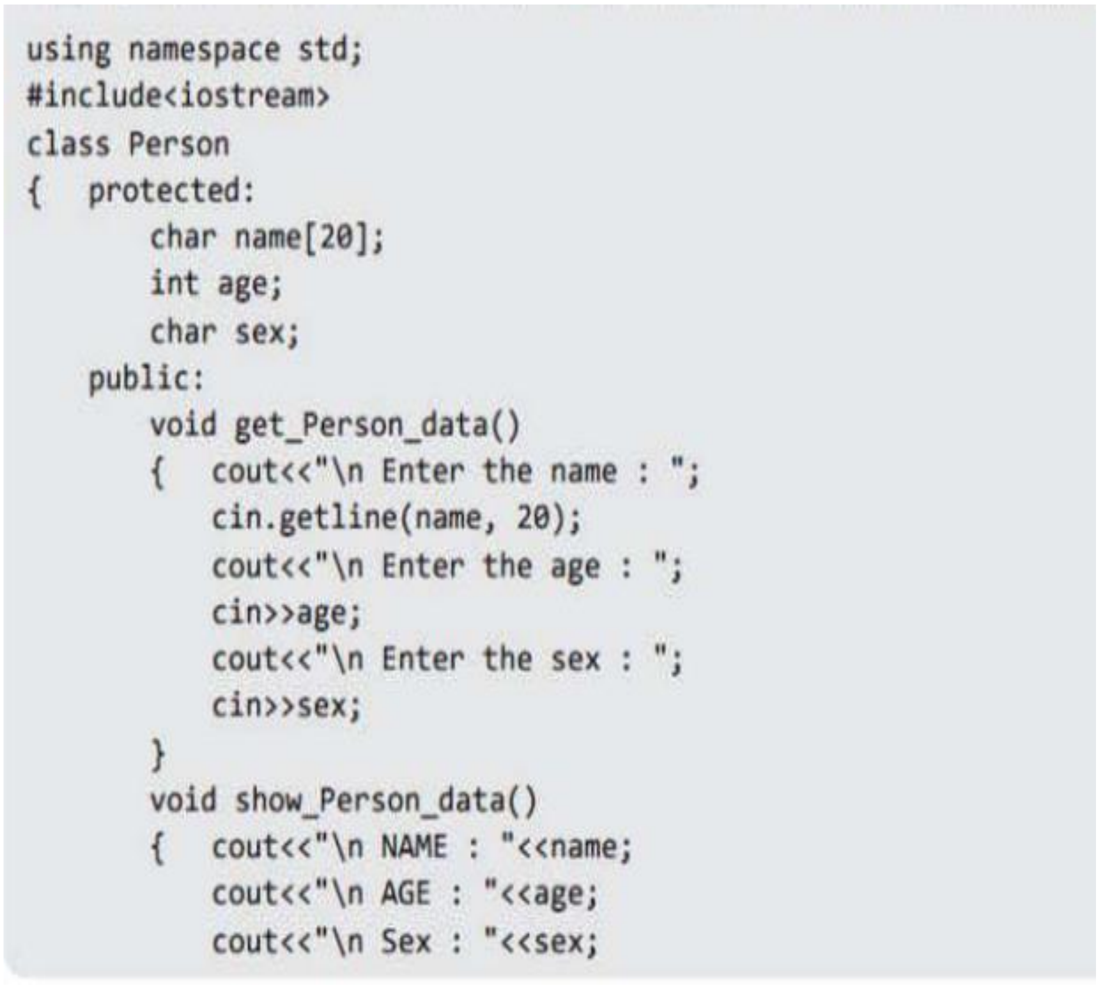
```
main()
{   Contract C;
    C.get_Cont_data();
    C.show_Cont_data();
    Permanent P;
    P.get_Per_data();
    P.show_Per_data();
}
```

OUTPUT

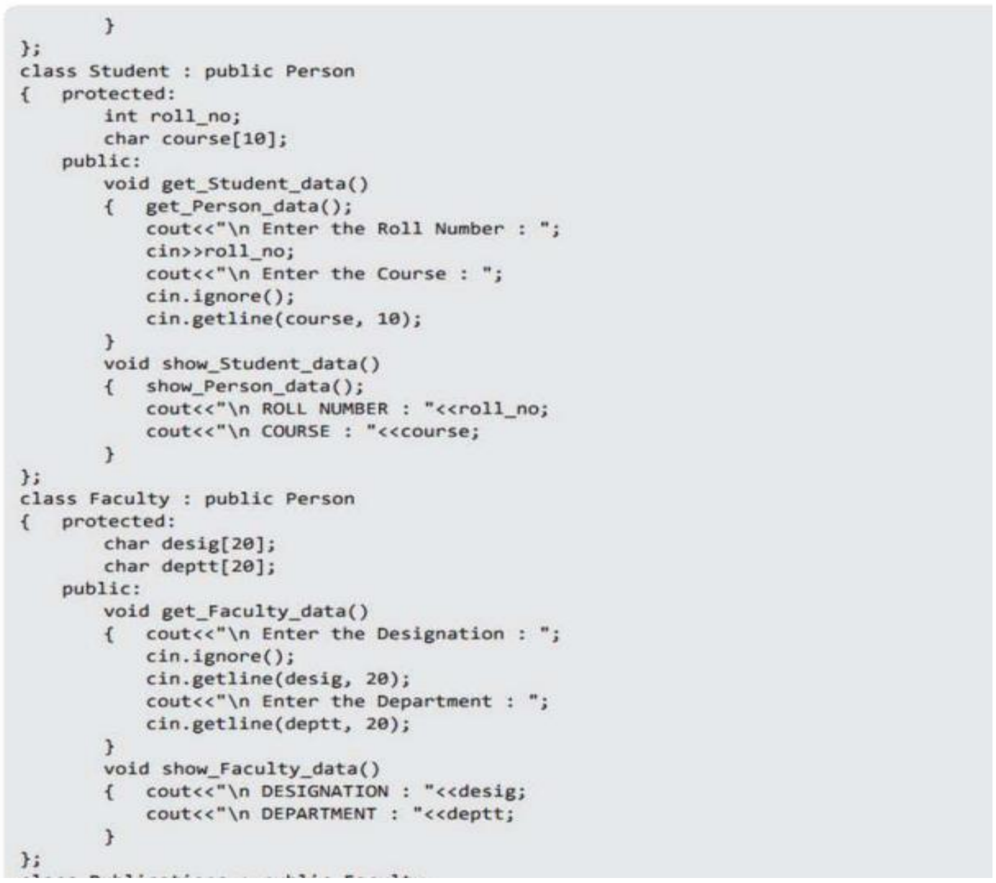
```
Enter Employee ID : 1234
Enter the Name : Anjani Kumar
Enter the Designation : Assistant Professor
Enter the number of hours : 30
Enter the wages per hour : 1000
EMP ID : 1234
NAME : Anjani Kumar
DESIGNATION : Assistant Professor
NUMBER OF HOURS WORKED : 30
WAGES PER HOUR : 1000
SALARY : 30000
Enter Employee ID : 5678
Enter the Name : Kamesh Bathla
Enter the Designation : Senior Manager
Enter the Basic Pay : 12000
Enter the HRA : 30000
Enter the TA : 40000
EMP ID : 5678
NAME : Kamesh Bathla
DESIGNATION : Senior Manager
SALARY (BASIC + HRA + DA + TA) = 95200
```

Problem 4:

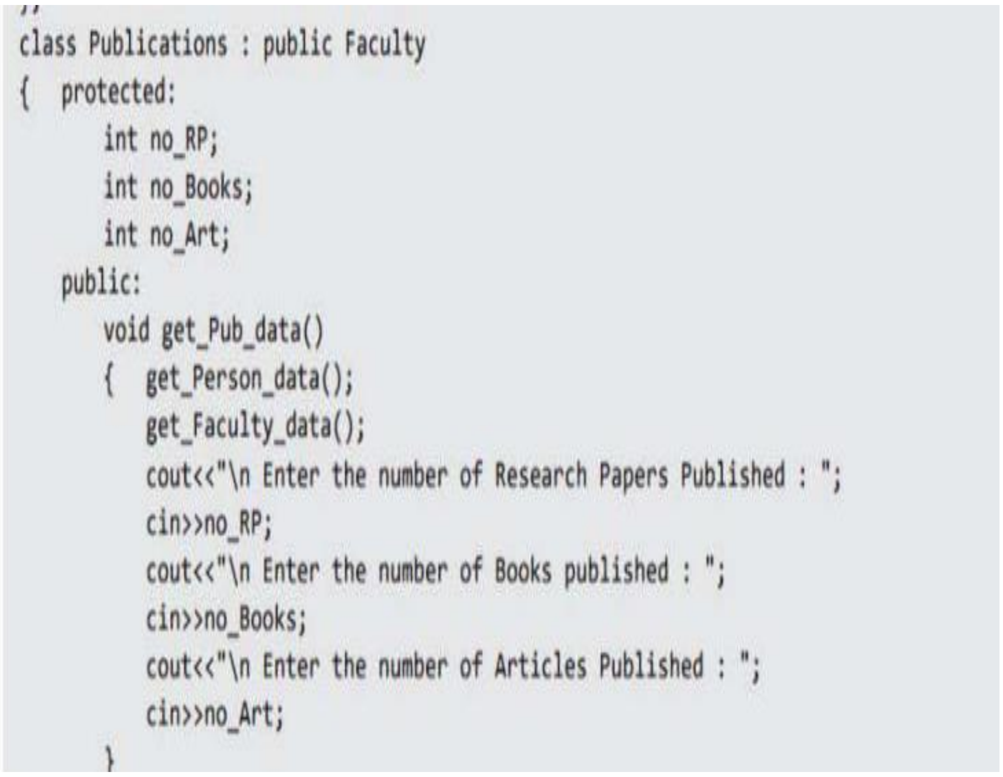
Write a program that implements hybrid inheritance. Classes Student and Faculty inherit the class Person. Class Faculty must be inherited by the class Publications.



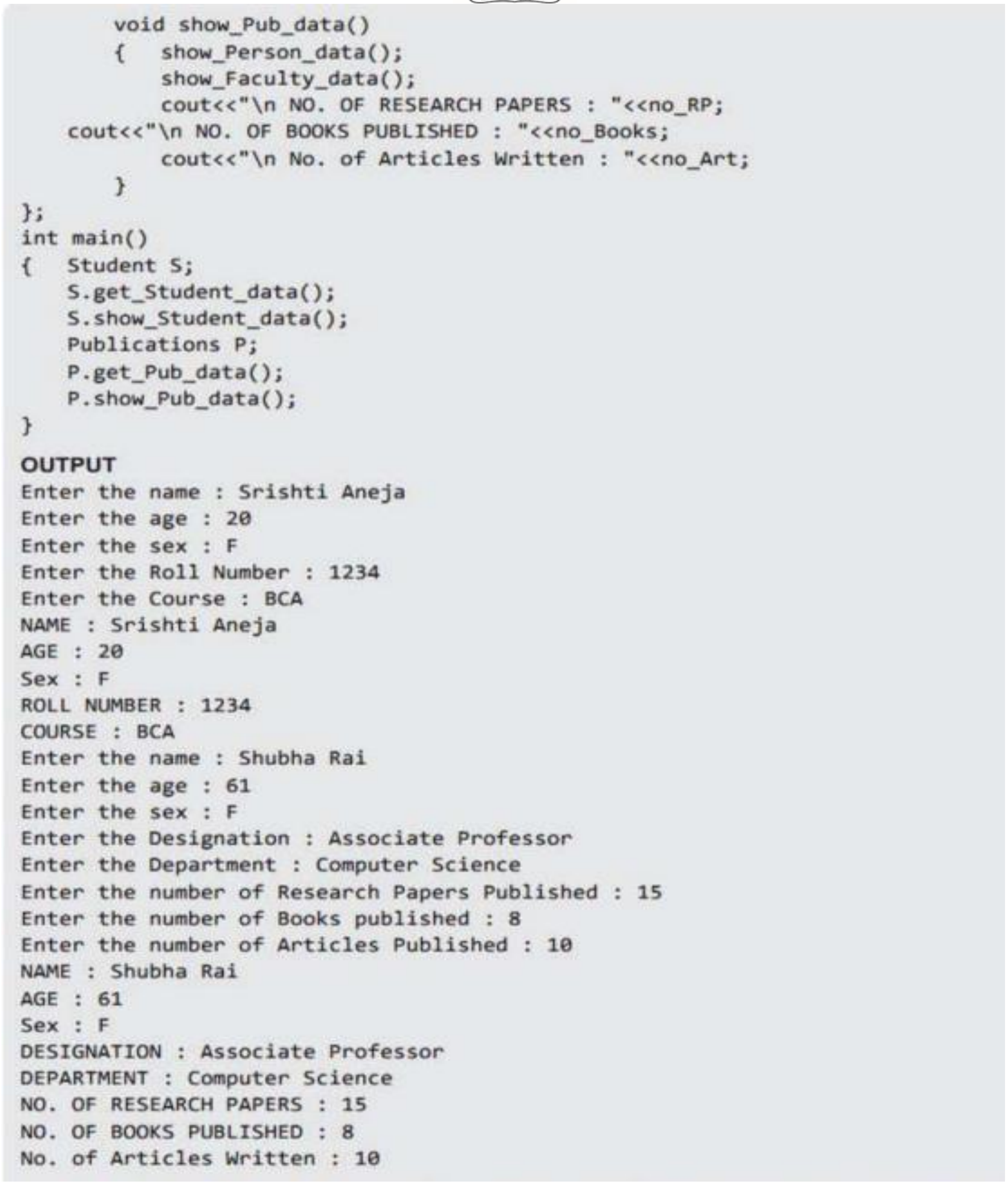
```
using namespace std;
#include<iostream>
class Person
{   protected:
        char name[20];
        int age;
        char sex;
    public:
        void get_Person_data()
        {   cout<<"\n Enter the name : ";
            cin.getline(name, 20);
            cout<<"\n Enter the age : ";
            cin>>age;
            cout<<"\n Enter the sex : ";
            cin>>sex;
        }
        void show_Person_data()
        {   cout<<"\n NAME : "<<name;
            cout<<"\n AGE : "<<age;
            cout<<"\n Sex : "<<sex;
```



```
        }
};
class Student : public Person
{   protected:
        int roll_no;
        char course[10];
    public:
        void get_Student_data()
        {   get_Person_data();
            cout<<"\n Enter the Roll Number : ";
            cin>>roll_no;
            cout<<"\n Enter the Course : ";
            cin.ignore();
            cin.getline(course, 10);
        }
        void show_Student_data()
        {   show_Person_data();
            cout<<"\n ROLL NUMBER : "<<roll_no;
            cout<<"\n COURSE : "<<course;
        }
};
class Faculty : public Person
{   protected:
        char desig[20];
        char deptt[20];
    public:
        void get_Faculty_data()
        {   cout<<"\n Enter the Designation : ";
            cin.ignore();
            cin.getline(desig, 20);
            cout<<"\n Enter the Department : ";
            cin.getline(deptt, 20);
        }
        void show_Faculty_data()
        {   cout<<"\n DESIGNATION : "<<desig;
            cout<<"\n DEPARTMENT : "<<deptt;
        }
};
```



```
};
class Publications : public Faculty
{   protected:
        int no_RP;
        int no_Books;
        int no_Art;
    public:
        void get_Pub_data()
        {   get_Person_data();
            get_Faculty_data();
            cout<<"\n Enter the number of Research Papers Published : ";
            cin>>no_RP;
            cout<<"\n Enter the number of Books published : ";
            cin>>no_Books;
            cout<<"\n Enter the number of Articles Published : ";
            cin>>no_Art;
        }
```



```
        void show_Pub_data()
        {   show_Person_data();
            show_Faculty_data();
            cout<<"\n NO. OF RESEARCH PAPERS : "<<no_RP;
    cout<<"\n NO. OF BOOKS PUBLISHED : "<<no_Books;
            cout<<"\n No. of Articles Written : "<<no_Art;
        }
};
int main()
{   Student S;
    S.get_Student_data();
    S.show_Student_data();
    Publications P;
    P.get_Pub_data();
    P.show_Pub_data();
}
```

OUTPUT

```
Enter the name : Srishti Aneja
Enter the age : 20
Enter the sex : F
Enter the Roll Number : 1234
Enter the Course : BCA
NAME : Srishti Aneja
AGE : 20
Sex : F
ROLL NUMBER : 1234
COURSE : BCA
Enter the name : Shubha Rai
Enter the age : 61
Enter the sex : F
Enter the Designation : Associate Professor
Enter the Department : Computer Science
Enter the number of Research Papers Published : 15
Enter the number of Books published : 8
Enter the number of Articles Published : 10
NAME : Shubha Rai
AGE : 61
Sex : F
DESIGNATION : Associate Professor
DEPARTMENT : Computer Science
NO. OF RESEARCH PAPERS : 15
NO. OF BOOKS PUBLISHED : 8
No. of Articles Written : 10
```

Exercises:

1.Is there any way by which a derived class can access private members of the base class?

If yes, how?

2. What happens when a protected member is inher-ited in private mode and public mode?

3. Explain the different types of inheritance.

4. Explain the concept of single inheritance with the help of a program code.

5. Differentiate between public, private, and protected inheritance using suitable examples.

6. Is it mandatory for the derived class to have a constructor?

7. What are the advantages of calling a base class constructor before a constructor of the derived class?

8. Explain the syntax of defining a derived class constructor.

9. Explain the concept of multi-level inheritance with the help of a suitable program code.

10. Explain the concept of multiple inheritance with the help of a program.

Study Learning Material Prepared by

Dr. S. KALAISELVI M.SC., M.Phil., B.Ed., Ph.D.,

ASSISTANT PROFESSOR,

DEPARTMENT OF MATHEMATICS,

SARAH TUCKER COLLEGE (AUTONOMOUS),

TIRUNELVELI-627007.

TAMIL NADU, INDIA.