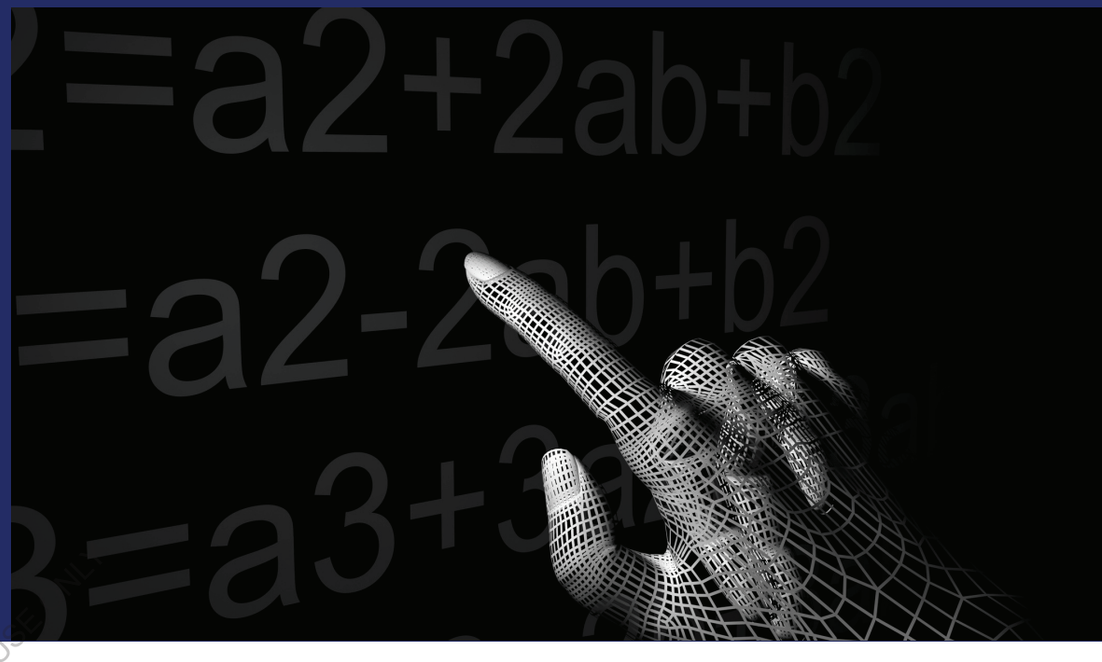
C++ Programming Language: A Practitioner's Approach is a title that provides proficient knowledge about programming in C++. Any beginners can use this book for getting knowledge about the concept of C++. This book can be act as a base to learn programming in C++and to prepare competitive exams and interview questionaries. C++ Programming Language: A Practitioner's Approach includes the concept of basic C++ programming, object-oriented concepts like classes and objects, inheritance, Files, exception handling, templates etc. The title contains programming example with every concept and every program is explained in a very simple manner.

Girish kumar, having the experience of about 15 years and Rydhm Beri having experience of 4 years in the field of programming like C, C++, Fortran77, java, VB6.0, VB.net, Python, C#. Also worked on live projects based on .Net and java with oracle, mysql, SQL server, Sqlite, as back-end. More then 15 national and international paper publications.

978-613-9-93986-2

C++ Programming Language

Kumar, Beri



Girish Kumar
Rydhm Beri

C++ Programming Language

A Practitioner's Approach

LAP LAMBERT Academic Publishing

Girish Kumar
Rydhm Beri

C++ Programming Language

Girish Kumar
Rydhm Beri

C++ Programming Language

A Practitioner's Approach

LAP LAMBERT Academic Publishing

Imprint
Any brand names and product names mentioned in this book are subject to trademark, brand or patent protection and are trademarks or registered trademarks of their respective holders. The use of brand names, product names, common names, trade names, product descriptions etc. even without a particular marking in this work is in no way to be construed to mean that such names may be regarded as unrestricted in respect of trademark and brand protection legislation and could thus be used by anyone.

Cover image: www.ingimage.com

Publisher:
LAP LAMBERT Academic Publishing
is a trademark of
International Book Market Service Ltd., member of OmniScriptum Publishing Group
17 Meldrum Street, Beau Bassin 71504, Mauritius

Printed at: see last page
ISBN: 978-613-9-93986-2

C++ Programming Language

A Practitioner's Approach

Girish Kumar
Assistant Professor
School of Computer Applications
Lovely Professional University,
Phagwara-144411, Punjab, India
girishvansh@gmail.com

Rydhm Beri
Assistant Professor
School of Computer Applications
Lovely Professional University,
Phagwara-144411, Punjab, India
rydhmberi@gmail.com

Table of Contents

Chapter 3 Classes and Objects in C++

Chapter 4 Operator Overloading and Type Conversion

Chapter 5 Inheritance

Chapter 6 Polymorphism

Chapter 7 Files

Chapter 8 Templates

Chapter 9 Exception Handling

Acknowledgement

First and foremost we would like to thank God for giving us courage to bring up this book.

Secondly, we wish to thank the people at Lambert Academic Publishing, to give us the opportunity of work with them for writing book of ***C++ Programming Language-A Practitioner's Approach.*** They give us the opportunity to publish our work with their esteemed publishing organization.

At the outset, we would like to propose a word of thanks for the people who gave us unending support and help in numerous ways from the stage when the idea of the book was conceived.

There are many people to thanks who encourage us and support to work on our book. We would like to thank our parents, our siblings, who motivate us to start work on this book. We are very thankful to our family members who were very supportive in our endeavor and offered emotional support during times of stress.

We would like to acknowledge thanks to our teachers that taught us in wonderful manner and made us capable to work on our book. We are also thankful to our colleagues and friends for their useful discussions and suggestions in several ways.

Lastly, we would like to thanks our critics, as without their criticism, we never became able to move step towards this book.

Girish Kumar

Rydhm Beri

About Authors

Mr Girish Kumar is working as an Assistant Professor in LPU. He has an experience of 15 years in the field of teaching. He has the knowledge of programming languages like core C, C++, Fortran77, C#, Java, Python and many other relevant fields. He has successfully accomplished many live projects. He has currently pursued for PHD in Digital Image Processing. He has a good knowledge on the other relevant fields like Data Structures, Database, Computer Graphics and many others.

Ms. Rydhm Beri working as an Assistant Professor in Lovely Professional University, Phagwara. She has an experience of 4 years in the field of teaching. Ms. Rydhm Beri has done M.Sc. Computer Science, MCA. She is Currently Pursuing Ph.D. in the field of embedded systems and IOT. The area of specialization is programming. As a Researcher till now Ms. Rydhm Beri published research papers in different area of research. She has knowledge of different programming languages like C,C++,JAVA,.Net, Python, PhP etc.

Chapter 1

Introduction to C++ Programming

Structure of the Unit

- Basic C++ I/O Statements
- Basic I/O Manipulators
- C++ Tokens
- Keywords
- Identifiers
- Constants
- Operators
- Data Types
- Built-In Data Types
- User Defined Types
- Derived Data Types
- Control Structures
- Expressions
- Conditional Expressions

Learning Objectives

- To introduce the basic console I/O statements
- To write simple C++ programs
- To write programs using I/O manipulators
- To understand the various C++ tokens
- To discuss the three categories of data types
- To understand the various control structures
- To show how expressions can be used in C++ programs

1.1 Introduction

C++ is an object oriented language. It is an expanded version of C language. C++ was invented by Bjarne Stroustrup in 1979 at Bell laboratories in Murray Hill, New Jersy. C++ was originally called as “C with classes”, however the name was changed to C++ in 1983.

C++ is a superset of C, hence almost all concepts available in C is also available in C++.This chapter discuss about all the fundamental concepts of this programming language. This chapter introduces basic console I/O statements and manipulators. This chapter then discusses all C++ tokens, data types and control structures. This chapter provides the necessary background to understand the C++ concepts presented in the subsequent chapters.

1.2 Basic Console I/O Statements in C++

Since C++ is a superset of C, all elements of the C language are also contained in the C++ language. Therefore, it is possible to write C++ programs that look just like C programs. You can still use functions such as printf( ) and scanf( ), C++ I/O is performed using I/O streams instead of I/O functions. The

output in C++ provided to standard output device using cout stream object of ostream class. The cout is used in conjunction with insertion operator <<
To output to the console, we write
cout << ***expression;***

Where expression can be any valid C++ expression, including another output expression.

cout<< “Hello World.\n”;
cout<<236.99;

The input in C++ taken from standard input device using cin stream object of istream class. The cin is used in conjunction with stream extraction operator >>.

To read input from keyboard, we use:
cin >> variables;

Where variable are the memory area that stores values.

cin>>a;
cin>>b;

Example 1:

The following C++ program makes use of the basic Console I/O statements to read and print two integer variables.

```
#include < iostream.h>
int main( )
{
   // variables declaration
   int i,j;
   cout << "Enter two integers: \n ";
   // input two integers
   cin >> i >> j;
   // output i then j and newline
   cout << "i= " << i << " j= " << j << "\n";
   return 0;
}
```

Output of the Program:

Enter two integers:

10 20

i=10 j=20

You can input as many items as you wish to input in one input statement. As in C, individual data items must be separated by whitespace characters (spaces, tabs, or newlines). When a string is read, input will stop when the first whitespace character is encountered.

NOTE :

A comment in C++ can be given in a same way as we give in C. /* .... */ is a multiline comment and // is a single line comment. For Example

/* This is a multiline comment and can

Span for multiline. This comment
is similar to C */

// This is a single line comment

Have you understood questions?

1. Write the two basic C++ I/O statements.
2. How will you write multiple line comment in C++?

1.3 Basic Input/Output Manipulators

C++ supports some I/O manipulators to format the display. The most frequently used manipulators are endl and setw. To use these functions you have to include iomanip.h header file.

The endl manipulator is similar to the newline character \n. Thus the statement
cout<<a<<endl<<b<<c<<,endl ;
will print the value of a,b and c in newline.

The setw manipulator is used to specify the width of the data to be displayed. Note that in C++ characters are left justified and numbers are right justified. Example
cout<<setw(5)<< 123 ; will output

		1	**2**	**3**

Here we have set the width as 5; since numbers are right justified the first two columns appear blank. Now consider this example
cout<<setw(10)<< Hello ,then the output will be

H	**e**	**l**	**l**	**o**					

Apart from these manipulators C++ allows us to create our own manipulators.

Have you understood these questions?

1. The width of an integer variable has to bet set to 5.How will you set it?
2. Write the manipulator that will cause a line break

1.4 C++ Tokens

In C++ every statement is made up of smallest individual elements called tokens. Tokens are the basic lexical elements. Tokens in C++ include the following

- Keywords
- Identifiers
- Constants
- Operators

1.4.1 Keywords

The list is given below contains all the C++ keywords. Every keyword will perform a specific operation in C++. Keywords in C++ cannot be redefined by a programmer; further keywords cannot be used as identifier name.

and	and_eq	asm	auto	bitand	bitor
bool	break	case	catch	char	class
compl	const	const_cast	continue	default	delete
do	double	dynamic_cast	else	enum	explicit
export	extern	false	float	for	friend
goto	if	inline	int	long	mutable
namespace	new	not	not_ep	operator	or
or_eq	private	protected	public	register	reinterpret_cast
return	short	signed	sizeof	static	static_cast
struct	switch	template	this	throw	true
try	typedef	typeid	typename	union	unsigned
using	virtual	void	volatile	wchart	while
xor	xor_eq				

1.4.2 Identifiers

Every named program component in C++ like variables, functions, arrays, classes etc. must have legal identifier (name) .In C++ all variables must be declared before they are used. Furthermore, variables must be used in a manner consistent with their associated type.

An identifier is formed with a sequence of letters (including '_ ') and digits. The first character must be a letter. Identifiers are case sensitive, i.e., first_name is different from First_name.

Examples of valid variable names are

myname, jj, name123

1.4.3 Constants

Constants refer to values that do not change during the execution of the program. Examples for constants are

1998 Integer constant

25.789 Floating point constant

“Hello World” String constant

‘a’ Character constant

1.4.4 Operators

The operators are the symbols that act upon operands. In other words, operators are the symbols that perform certain task upon operand associated with it. In C++, operators can perform operations on one, two or three operands. The various operators in C++ and their associatively are listed below. The Operators are listed according to their precedence.

Associativity	Operators
Left to Right	::
Left to Right	( ), [ ], ->, ., typeid, casts
Right to Left	!(negation),~(bit-not)
Right to Left	new, delete, ++, --, -(unary),*(unary), &(unary), sizeof
Left to Right	*,/, %,-,+
Left to Right	<<,>>
Left to Right	<,<=,>,>=,==,!=
Left to Right	&(bitwise AND), |(bitwise OR), ^(bitwise XOR)
Left to Right	&&(Logical AND), ||(Logical OR),
Right to Left	?:
Right to Left	=,+=,-=,/=,%=,>>=,&=
Right to Left	Throw
Left to Right	,

Have you Understood Questions?

1. Write the four basic C++ tokens.
2. How does a constant differ from a variable

1.5 Data Types

Data type represents the type of data or value a variable can hold. Every variable can have a data type associated with it which specifies the type of value it holds. C++ supports three categories of data types.

- Built-in Data Types
- User Defined Data Types
- Derived Data Types

1.5.1 Built-in Data Types

C++ supports the following built in data types. They are

Signed integer:

The signed integer types are short int (abbreviation short), int and long int (abbreviation long). Optionally the keyword signed can be used in front of these definitions, e.g. signed int. The size of these variables depends on the implementation of the compiler. The only rule for the length of short, int, long is: short<= int<=long, i. e. they may have all the same size.

Unsigned integer:

The unsigned integer types are unsigned short int (abbreviation unsigned short), unsigned int and unsigned long int (abbreviation unsigned long). The sizes follow the same rules as for singed integers.

Character type:

Character type variables can hold a single character. The character types are char, signed char and unsigned char. Note that a simple char can mean either signed or unsigned, as this is not defined in ANSI C. It depends therefore only on the implementation of the compiler.

Floating point types:

The floating point types are float, double and long double. A float variable has a single precision value. A double or long double variable has at least single precision, but often has double precision (however, this depends on the implementation). The rule for the size of floating point types is: float <= double<=long double.

Boolean type:

Standard C++ has an explicit boolean type bool. (C only has an implicit boolean type: 0 = logical false, nonzero = logical true). This type is used to express results of logical operations. The value of the Boolean type is symbolic, either false or true. To ensure compatibility with C and older C++ compiler, the

boolean type is automatically converted to/from integer values. In these cases an integer value of 0 indicates false and a
non-zero value indicates true.

Empty type:

The special type void is used for three occasions: defining a function without parameters, defining a function that has no return value and for generic pointers. The following table with sizes of data types has been generated on a Windows NT and Linux system

Data Type	Size in Bytes for Windows NT	Size in Bytes for Linux System
Char	1	1
Short int	2	2
Int	4	4
Float	4	4
Double	8	8
Long double	8	12

1.5.2 User Defined Data Types

Structures, Unions and Classes

A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.
A structure in C++ can be declared as

```
struct point
{
        int x;
        int y;
};
```

The keyword struct introduces a structure declaration, which is a list of declarations enclosed in braces. A struct declaration defines a type. The right brace that terminates the list of members may be followed by a list of variables, just as for any basic type. For example the variables of the struct point can be declared as

struct point a,b,c ;

The members of the structures are accessed as follows

struct-name.member (or)

struct –pointer ->member

A union is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment

requirements. Unions provide a way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program.The syntax is based on structures:

```
union u1
{
        int i;
        float f;
        char c;
} u;
```

The variable u will be large enough to hold the largest of the three types; the specific size is implementation-dependent. Any of these types may be assigned to u and then used in expressions, so long as the usage is consistent: the type retrieved must be the type most recently stored. It is the programmer's responsibility to keep track of which type is currently stored in a union; the results are implementation-dependent if something is stored as one type and extracted as another.

Syntactically, members of a union are accessed as

union-name.member

or

union-pointer->member .

Classes are discussed in detail in chapter 3

Enumeration Data Type

An enumeration is a set of named integer constants that specify all the allowable set of values a variable of that type may have.The general syntax of enumeration data type is

enum enum-name{enumeration values} ;

Example

```
enum states{Andhara,Karnataka,Kerala,Tamilnadu} ;
enum currency { dollar,euro,rupees}
```

Once the enumeration is created, we can declare variables of that type. Example

```
coin money;
states s;
```

Given these declarations the following statements are valid. The statements

```
if(s= =kerala)
        cout<<"Welcome to God's Own Country";
```

and

```
cin>>money;
if(money==rupees)
{
        //do this
}
```

are perfectly valid.

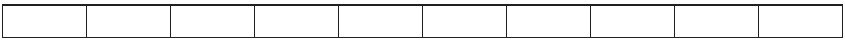
1.5.3 Derived Data Types

Arrays

In C++ arrays are handled in the same way as of C. For example the declaration

int a[10];

defines an array of size 10, that is, a block of 10 consecutive objects named a[0], a[1], ...,a[9].



The notation a[i] refers to the i-th element of the array.

Pointers

A pointer is a variable that contains the address of a variable. Pointers in C++ are declared and initialized as in C.

Examples

```
int a= 1, b= 2
int *ip; /* ip is a pointer to int */
ip = &a /* ip now points to a*/
b= *ip; /* bis now 1 */
*ip = 0; /* a is now 0 */
```

From these examples you can see that the declaration and usage of pointer in C and C++ are similar. Pointers are extensively used in C++ for dynamic memory management and to achieve run time polymorphism.

Functions

Functions provide modular structure to the programming. Functions in C++ were discussed in detail in chapter 2.

Have you Understood Questions?

1. Mention the three categories of data types available in C++
2. How a structure does differ from union?
3. What do you mean by enumeration data type?
4. What is a pointer variable?

1.6 Control Structures

The control-flow of a language specifies the order in which computations are performed. In C++ the program executes sequentially starts from main function. Sometimes, there is a requirement to change the flow of program that can be done using control structures. The various control structures supported by C++ is discussed below.

If-Else

The if-else statement is used for making decisions. The syntax is

```
if (expression)
        statement1
else
        statement2
```

Where the else part is optional. The expression is evaluated; if it is true (that is, if expression has a non-zero value), statement1 is executed. If it is false (expression is zero) and if there is an else part, statement2 is executed instead.

Example

```
if(a>b)
        c=a;
else
        c=b;
```

In this example the value of a will be assigned c if a is greater than b else the value of b will be assigned to c.

Else if

The syntax for else-if ladder structure is given below

```
if (expression)
        statement
else if (expression)
        statement
else if (expression)
        statement
else if (expression)
        statement
else
        statement
```


This sequence of if statements is the most general way of writing a multi-way decision. The expressions are evaluated in order; if an expression is true, the statement associated with it is executed, and this terminates the whole chain.

The last else part handles the default case where none of the other conditions is satisfied.

An example for this construct is given below.

```
if (( a>b) && (a>c))
        cout<<"A is big";
else if ((b>a) && (b>c))
        cout<<"B is big";
else if ((c>a) && (c>b))
        cout<<"C is big";
```

Switch Statement

The switch statement is a multi-way decision that tests whether an expression matches one of a number of constant integer values, and branches accordingly. The syntax of switch statement is given below.

```
switch (expression)
{
        case const-expr:
                statements
        case const-expr:
                statements
        default:
                statements
}
```

Each case is labeled by one or more integer-valued constants or constant expressions. If a case matches the expression value, execution starts at that case. A default is optional; it gets executed if none of the cases is matched. Cases and the default clause can occur in any order. An example for switch case construct is given below.

```
switch(k)
{
        case 1:
                cout<< "The value of k is 1";
                break ;
        case 2:
                cout<< "The value of k is 1";
                break;
        case 3:
                cout<< "The value of k is 1";
                break;
        default:
                cout<<"The value of k is other than 1,2,3";
}
```


Here based on the value of k case 1, case 2 or case 3 will be executed .If the value of k is other than 1, 2 and 3 then default case will be executed. The break statement causes an immediate exit from the switch after the code for one case is done, execution falls through to the next case unless you leave the switch. You can use break statement to explicitly come out of the switch.

Looping Statements

C++ supports three type of looping statements as C language .They are

- While Loop
- Do…While Loop

- For Loop

While Loop

The while loop is entry controlled loop. While entering loop the condition is checked, if it evaluates to true then body of the loop is executed until the condition evaluates to true, otherwise the control move out of the loop. The Syntax of while loop is

```
while (expression)
{
        statements ;
}
```

the expression is evaluated. If it is non-zero (i.e. if the condition given in the expression is true) , statements is executed and expression is re-evaluated. This cycle continues until expression becomes zero (false).

Example

```
i=0; //initial value
while(i<100) //condition
{
        cout<<i;
        i=i+2 //increment
}
```

Here in this example the while loop prints all the even numbers between 0 and 100.Note the while loop checks for the condition i<100 every time when the loop iterates.

For Loop

For loop is one step loop in which initialization, condition checking, and increment/decrement of loop is performed at one step. The for statement is given below

```
for (expr1; expr2; expr3)
{
        statements ;
}
```

is equivalent to the while statement given in the previous example

```
expr1
while (expr2)
{
        statement
        expr3;
}
```

Most commonly, expr1 and expr3 are assignments or function calls and expr2 is a relational expression. Any of the three parts can be omitted, although the semicolons must remain. If expr1 or expr3 is omitted, it is simply dropped from the expansion. If the test, expr2, is not present, it is taken as permanently true.

Example

```
for (i=0 ;i<100 ;i=i+2)
        cout<<i ;
```

This is the for loop version of the previous program for printing even numbers between 0 and 100.Here we can note that in for loop the initilaization,condition and increment statements are kept together in the for statement.

Do while Loop

The while and for loops test the termination condition at the top. But the do-while, tests at the bottom after making each pass through the loop body; the body is always executed at least once. Do while loop is also called exit controlled loop. The Syntax of do while statement is

```
do
{
        Statements ;
}while (expression);
```

The statement is executed, and then expression is evaluated. If it is true, statement is evaluated again, and so on. When the expression becomes false, the loop terminates. do...while loop is essential when we expect the loop to be executed at least once irrespective of the condition.

Example

```
i=0; //initialization
do
{
        cout<<i;
        i+=2; //increment
}while(i<100); //condition.
```

Break and Continue Statements

Sometimes it may require to exit from a loop other than by testing at the top or bottom. The break statement provides an early exit from for, while, and do, just as from switch. A break causes the innermost enclosing loop or switch to be exited immediately. The example given below helps a better understanding about break statement.

```
while (i<100)
{
        cout<<"ENTER NEGATIVE NUMBER TO EXIT";
        cin>>a;
        if(a < 0)
                break;
        sum+=a; }
```

In this example the while loop is written to execute 100 times but however when the user gives a negative input the break statement causes the loop to make an early exit

The continue statement is related to break, it causes the next iteration of the enclosing for, while, or do loop to begin. In the while and do, this means that the test part is executed immediately; in the for, control passes to the increment step. The continue statement applies only to loops, not to switch. Consider the example

```
for (i = 0; i < 100; i++)
{
        cin>>c ;
        if (c <= 0) /* skip the remaining part of the loop */
                continue;
        no_of_positive++ ;
}
```

The continue statement is used here to skip the remaining part of the loop when a negative number is given as input.

Have you Understood Questions?

1. Mention the control structures that is used for multi way decisions
2. How does a while loop differs from do while loop
3. In which situations we will use break and continue statements

1.7 Expressions

An expression is a combination of operands (both variables and constants), operators arranged as per the syntax of the language. An expression may be an simple expression such as

i=i+2

An expression may also use combination of simple expressions such as

i=i * 2 + (m +1)/2

An expression such as

a=a+2

in which the variable on the left side is repeated immediately on the right, can be written in the compressed form

a+= 2

The operator += is called an assignment operator.

Most binary operators (operators like + that have a left and right operand) have a corresponding assignment operator op=, where op is one of

+ - * / % << >> & ^ |

k += m*5 is equivalent to

k=k+m*5

Conditional Expressions

The statements

```
if (a > b)
        c = a;
else
        c = b;
```

Assign c the maximum of a and b. The conditional expression, written with the ternary operator ``?:", provides an alternate way to write this and similar constructions. In the expression

```
expr1 ? expr2 : expr3
```

the expression expr1 is evaluated first. If it is non-zero (true), then the expression expr2 is evaluated, and that is the value of the conditional expression. Otherwise expr3 is evaluated, and that is the value. Only one of expr2 and expr3 is evaluated. Thus to set c to the maximum of a and b.

Have you Understood Questions?

1. Write the expression in short form a=a+b
2. What is a conditional expression?

1.8 Summary

- C++ is an object oriented language. It is an expanded version of C language.
- C++ I/O is performed using I/O operators instead of I/O functions,C++ uses cin and cout statements for performing I/O operations
- C++ supports some I/O manipulators to format the display. The most frequently used manipulators are endl and setw.
- C++ tokens include keywords, operators, identifiers and constants
- Every named program component in C++ like variables, functions, arrays, classes etc. must have legal identifier (name)
- Constants refer to values that do not change during the execution of the program.
- C++ supports three categories of data types, they are built in, user defined and derived data types.
- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.
- A union is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements.
- An enumeration is a set of named integer constants that specify all the allowable set of values a variable of that type may have
- A pointer is a variable that contains the address of a variable
- All basic control structures that are present in C language like if..else,while loop,for loop,break and continue statements are also found in C++ all of them perform the same task as in the C language.

1.9 Exercises

Short Questions

1. Write any two differences between C and C++.
2. cout is an input statement(TRUE/FALSE)
3. A logical AND we have_____ associativity.
4. _____ operator is used to access a structure variables.
5. Write the significance of void datatype.
6. List out the various user defined datatypes available in C++.

Long Questions

1. Explain the various data types available in C++
2. Explain how structures are created and processed in C++
3. Discuss pointers in detail

Programming Exercises

1. Write a program to read a set of integers from the keyboard and determine whether they are ODD or EVEN
2. Write a program to find the sum of the digits of an number
3. Write a program to generate prime numbers in the range 200-999
4. Write a program to count the number of negative numbers in an array

Answers to Have you Understood Questions

Section 1.3

1. cin and cout
2. /* comment lines */

Section 1.4

1. setw(5)
2. endl

Section 1.5

1. Keywords,Identifiers,Constants,Operators
2. A constant will not change its value during program execution, but a variable does.

Section 1.6

1. Built in data type, User defined datatpe and derived data type.
2. A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.
3. An enumeration is a set of named integer constants that specify all the allowable set of values a variable of that type may have.
4. A pointer is a variable that contains the address of a variable.

Section 1.7

1. else.. if ladder structure and switch statement.
2. While loop checks the condition first and then executes the loop on the other hand do while loop checks that condition at the last. We can guarantee that a do while loop will at least once but such guarantee cannot be given for while loop.
3. The break statement provides an early exit from for, while, and do, just as from switch.

Section 1.8

1. a+=b.
2. The conditional expression, written with the ternary operator ``?:", provides an alternate way to write simple if statement.

Chapter 2

Functions

Structure of the Unit

- Function Prototypes
- Functions with Return Values
- Call by Value
- Call by Reference
- Return by Reference
- Inline Functions
- Functions with Default Arguments
- Function Overloading

Learning Objectives

- To introduce the basic concepts of Functions
- To write simple function programs
- To understand the concepts of returning from functions
- To understand call by value and call by reference concept
- To discuss the concept of inline function
- To understand the concept of function overloading

2.1Introduction

Functions are one of the most important features of program development. Functions are the basic building blocks of C++. Functions provide modular structure to a program Functions help to divide a larger program into number of smaller units which is the basis of top down approach. Functions also provide programmers an easy and convenient way of designing programs in which complex computations can be built into functions. To re-use the same part of a program, we can define a function, which takes parameters, and returns a result of various types. This chapter introduces you to create and use your own function. This chapter also discusses various important concepts of functions like call by value and reference, Inline functions and function overloading.

2.2Advantages of Functions

The advantages of writing functions are

- Functions provides Reusability
- Functions give modular structure to a program
- Writing functions make programming easy

2.3Function Prototypes

Typical definition or prototype for a function contains

- The type of the value it returns
- An identifier for its name and
- The list of parameters with their types.

The syntax to write a function is.

<return type> function name(argument list)

{

//Body of the Function.

}

Example 1:

The program given below is a makes use of a simple function that takes a integer parameter.

```
#include <iostream.h>
void show(int); // FUNCTION PROTOTYPE
void main()
{
        int a;
        cout<<"Enter the value for A";
        cin>>a
        show(a);
}
void show(int x)
{
        cout<<"The value passed to function is "<< x;
}
```


Output of the program:

Enter the value for A 10

The value passed to function is 10

This program make use of a function called show( ) with an integer parameter. The function just displays the value of x. Since the function is not returning anything its return type is made void.

2.3.1 Functions with Return Values.

A function may or may not return any value to the calling function. A function can take parameters and also can return value.

Example 2:

This function show() used in this program takes two parameters and returns the sum of the parameters

```
#include <iostream.h>
int mul(int,int); // FUNCTION PROTOTYPE
void main()
{
        int a,b,c;
        cout<<"Enter two values <<endl";
        cin>>a>>b;
        c=mul(a,b);
        cout<<"The Product is "<<c;
}
int show(int x,int y)
{
        int z;
        z=x * y;
        return z;
}
```

Output of the program:

Enter two values 10 20

The Product is 200

The variable a,b are known as actual parameters whose values are copied to the dummy variables x and y. The result of the multiplication is stored in z and this value is copied to c.

2.3.2 Call by Value

By default, functions pass arguments by value, which means that when the function is used, the actual values of arguments are copied into the dummy parameters. The main effect is that even if the dummy parameters are modified the corresponding parameter will not change the argument's value.

Example 3:

The following program makes use of call by value concept to swap the values of two variables.

```
#include<iostream.h>
void swap(int,int);
void main()
{
        int a,b;
        cout<<"enter two values " <<endl;
        cin>>a>>b;
        swap(a,b);
        cout<<"Values in the main program\n";
```

```
        cout<<a<<"\t"<<b;
}
void swap(int x,int y)
{
        int t;
        t=x;
        x=y;
        y=t;
        cout<<"The values after swapping in swap() function \n "<<x<<"\t"<<y;
}
```

Output of the program:

Enter two values 10 20

The values after swapping in swap() function

20 10

Values in the main program 10 20

If suppose you give 10 and 20 as input to a and b which are copied x and y parameters of the swap function(which swaps the value of x and y).The swapped values will not affect the actual parameters a and b thus the cout statement in the main program will print only 20 and 10.

2.3.3 Call by Reference

In certain situations, it is more effcient or convenient to be able to modify the argument(s). To do that, the & symbol specifies that the actual parameter and dummy the argument correspond to the same value. This is called passing an argument by reference.

Example 4:

The following program makes use of call by reference concept to swap the values of two variables.

```
#include <iostream.h>
#include <math.h>
void swap(int &x,int &y)
{
        int t;
        t=x;
        x=y;
        y=t;
        cout<<"The values after swapping in swap() function \n "<<x<<"\t"<<y;
}
```

```
void main( )
{
        int a,b;
        cout<<"enter two values " <<endl;
        cin>>a>>b;
        swap(a,b);
        cout<<"Values in the main program\n";
        cout<<a<<b;
}
```

Output of the program:

Enter two values 10 20

The values after swapping in swap() function

20 10

Values in the main program 20 10

In this example the variables x and y will act as alias names for a and b thus any change in x and y will be reflected in a and b.

2.3.4 Return by Reference

A function can also return by reference.

Example 5:

The program given below illustrates the return by reference concept.

```
#include<iostream.h>
#include<iomanip.h>
int & check(int & a,int &b)
{
        if(a==b)
                return a;
        else
                return b;
}
void main( )
{
        int x,y;
        cout<<"enter two values " <<endl;
        cin>>x>>y;
        check(x,y)=x;
        cout<<"After executing check function "<<endl;
        cout<<x <<"\t"<<y;
}
```

Output of the program:

Enter two values 10 20

After executing check function

10 10

This function returns reference to a or b. The function check returns reference to either x or y based on the condition. Thus whatever value is given as input for x and y the function call assigns the value of x to the reference that is returned thus the cout statement displays the value of x twice.

Have you Understood Questions?

1. Mention the advantages of writing functions.
2. Give the syntax to write functions.
3. What do you mean by Call by reference?
4. How will you declare a reference variable in C++?

2.4Inline Functions

Inline functions are like macros. Inline functions will avoid the time complexity that arises due to context switch over head in a function call. A context switch involves locating the function, moving to it, saving the register status, pushing the arguments in the stack and returning to the calling function. All these overheads can be avoided by making a function inline. Inline functions have some restrictions.

- Function containing looping structure cannot be made inline.
- Recursive functions cannot be made inline.
- A function with many lines of coding cannot be made inline.

The disadvantage of in-line functions is that if they are too large and called to often, the program grows larger. For this reason, in general only short functions are declared as in-line functions.

A Function can be made inline by prefixing it with the keyword inline. The Syntax for writing Inline function is given below.

inline function-prototype
{
    function body
}

Example 6:

This example program makes use of inline function to determine whether the given number is Odd/Even.

```
#include < iostream.h >
#include<iomanip.h>
inline int even(int x)
{
        if (x %2==0)
                return 1;
        else
                return 0;
}
int main( )
{
        int a;
        cout<<"Enter any number"<<endl;
        cin>>a;
        if(even(a))
                cout<<"EVEN NUMBER";
        else
                cout<<"ODD NUMBER";
}
```


Output of the program:

Enter any number

20

EVEN NUMBER

Have you Understood Questions?

1. What do you mean by Inline functions?
2. How will you make a function Inline?

2.5Functions with Default Arguments

In C++ it is possible to call functions without passing all its arguments. This can be made possible by specifying the default arguments. The default arguments will be specified with some default values. Default parameters will be used when there is a missing parameter in the function call.

For Example consider the following function declaration.

int add(int x , int y=100,int z=200);

Here we can note that the default value to the parameter is assigned as an ordinary variable initialization. Thus all the function calls given below are valid.

m=add(15); //two argument missing

m=add(10,20); //one argument missing

m=add(10,20,30)

The following point has to be noted while writing functions with default arguments.

- A function can take default arguments only as indicated in the previous example (int y =100 is a default parameter)
- A default parameter is used only when a parameter is missing.
- Default parameters should be assigned from right to left.

The following function declarations are wrong

int add(int x=100,int y,int z) //default parameters should be assigned from left to right

int add(int x,int y=100,int z) //default parameters should be assigned from left to right

Example 7:

The following example illustrates the concept of default arguments in a function

```
#include<iostream.h>
#include<iomanip.h>
void add(int ,int=5,int=7); // DEFAULT VALUE DECLARED
void main()
{
        int a,b,c;
        cout<<"enter three values"<<endl;
        cin>>a>>b>>c;
        cout<<"Output with only 2 parameters "<<endl;
        add(a,b); \\ Third parameter missing hence default parameter used
        cout<<"Output with all 3 parameters;
        add(a,b,c);
}
void add( int x, int y, int z)
{
        int t;
        t=x+y+z;
        cout<<t;
}
```


Output of the program:

Enter three values

10 20 30

Output with only 2 parameters

37

Output with all 3 parameters 60

Have you Understood Questions?

1. Default parameters are assigned from _____ to _____

2.6Function Overloading

Function overloading is one of the most important features available in C++ two or more functions can share the same name as long as either the type of their arguments differs or the number of their arguments differs - or both. When two or more functions share the same name, they are said overloaded.

Overloaded functions can help reduce the complexity of a program by allowing related operations to be referred to by the same name. To overload a function, simply declare and define all required versions. The compiler will automatically select the correct version based upon the number and/or type of the arguments used to call the function. Function overloading provides the compile time polymorphism feature of OOPS.

The Following set of functions are overloaded.

i. float volume(int x);
   float volume(int x,int y);
ii. int interest (int p);
   int interest (float p);
   float interest (int p,float r);
   float interest (int p ,float r,int n);
iii. float area(int x,float y);
   float area(float m,float n);

The following sets of functions are not overloaded.

i. float mul(int x,int y)
   int mul(int x,int y)
ii. int convert(int x)
   float convert (int j);

Remember that the return type alone is not sufficient to allow function overloading. If two functions differ only in the type of data they return, the compiler will not always be able to select the proper one to call.

Example 8:

The following example illustrates the overloading of sum( ) function.

```
#include<iostream.h>
#include<iomanip.h>
void sum(int,int);
void sum(int,int,int);

void main()

{
        int a ,b, c ;
```

```
        cout<<"Enter three values "<<endl;
        cin>>a>>b>>c;
        cout<<"Calling the function with two parameters"<<endl;
        sum(a,b);
        cout<<"Calling the function with three parameters"<<endl;
        sum(a,b,c);
}
void sum(int x,int y)
{
        int z;
        z=x+y;
        cout<<z;
}
void sum(int x,int y,int z)
{
        int r;
        r=x+y+z;
        cout<<r;
}
```


Output of the program:

Enter three values

10 20 30

Calling with two parameters

30

Output with all 3 parameters 60

The compiler automatically calls the correct version of the function based upon the type of data used as an argument. Overloaded functions can also differ in the number of arguments.

Have you Understood Questions?

1. What do you mean by function overloading?
2. Is the following set of function overloaded?

```
int mul(int m,int n)
float mul(int k,int j)
```

2.7Summary

- Functions provide modular structure to C++ program Functions help to divide a larger program into number of smaller units which is the basis of top down approach.
- In C++ functions can be called by value, called by reference and can return by reference
- By default, functions pass arguments by value, which means that when the function is used, the actual values of arguments are copied into the dummy parameters.
- In call by reference, the & symbol specifies that the actual parameter and dummy the argument correspond to the same value.
- Inline functions are like macros. Inline functions will avoid the time complexity that arises due to context switch over head in a function call.
- In C++ it is possible to call functions without passing all its arguments. This can be made possible by specifying the default arguments
- In C++ two or more functions can share the same name as long as either the type of their arguments differs or the number of their arguments differs - or both. When two or more functions share the same name, they are said overloaded.

2.8Exercises

Short Questions

1. How does functions reduce complexity of a program
2. What do you mean by function overloading? Explain its advantages.
3. List some advantages of Inline functions
4. In call by reference you use _____ symbol
5. Inline functions in C++ are similar to _____ of C.

Long Questions

1. Explain the concept of call by value, call by reference and return by reference in detail.
2. Mention the rules associated with passing of default arguments. Give example
3. Explain function overloading with examples.

Programming Exercises

1. Create a function to compute simple interest. Supply default arguments to the function
2. Create an overloaded function called “myabs()” to find absolute value for a number.(absolute value for a number is the positive value of that number)

Answers to Have you Understood Questions

Section 2.3

1. The advantages of writing functions are (1) Functions provides Reusability (2) Functions give modular structure to a program (3) Writing functions make programming easy
2. The syntax to write a function is.

```
<return type> function name(argument list)
{
        //BODY OF THE FUNTION.
}
```

3. In certain situations, it is more effcient or convenient to be able to modify the argument(s). To do that, the & symbol specifies that the actual parameter and dummy the argument correspond to the same value. This is called passing an argument by reference.
4. By referencing the & symbol before the variable declaration.

Section 2.4

1. Inline functions are like macros. Inline functions will avoid the time complexity that arises due to context switch over head in a function call.
2. By prefixing the function declaration with inline keyword

Section 2.5

1. Right to left

Section 2.6

1. In C++ two or more functions can share the same name as long as either the type of their arguments differs or the number of their arguments differs - or both. When two or more functions share the same name, they are said overloaded.
2. No

Chapter 3

Classes and Objects in C++

Structure of the Unit

- Classes and Objects
- Defining Member Functions
- Creating Objects
- Accessing Class Members
- Member Functions with Parameters
- Inline Functions Outside the Class
- Constructors
- Constructors that take parameters
- Constructor Overloading
- Copy Constructors
- Destructors
- Passing Objects to Functions
- Returning Objects from Functions
- Static Data Members
- Static Member Functions
- Const Member Functions
- Friend Functions
- Friend Functions for two Classes

Learning Objectives

- To introduce the Classes and Objects
- To discuss member function inside and outside class definition
- To present the concept of constructors, Copy Constructors and Constructor Overloading
- To Present the concept of Destructor
- To show how to pass and return objects from functions
- To discuss static data member and static functions
- To introduce constant member functions
- To introduce friend functions and friend classes

3.1Introduction

In the previous chapters, you have followed the modular approach to C++ programming. The power of C++ lies in writing object oriented programs. In object oriented programming the data members and the functions which act upon those data are combined into a single entity called Object. Object provides a systematic way of accessing the data members. This chapter introduces you to write object oriented C++ programs using classes and objects. This chapter then discusses details regarding Constructers and Destructors. Further this chapter introduces static data members, static functions and const functions. At the end this chapter provides necessary details regarding friend functions and classes.

3.2Classes and Objects

Class is another user defined data type in C++. A class is the definition of a data structure and the associated operations that can be done on it .Classes in C++ are also called as Abstract Data Type (ADT) .Classes bind the data with its associated code. In C++, a class is declared using the class keyword. The syntax of a class declaration is similar to that of a structure. Its general form is,

```
class class-name
{
        private:
                // private functions and variables
        public:
                // public functions and variables
} object-list;
```

In a class declaration the object-list and private keyword is optional. We can create objects of the class in any function or class. If we skip to write private keyword, by default the members written before public keyword are considered as private. Functions and variables declared inside the class declaration are said to be members of the class. This means that they are accessible only by the members of that class. To declare public class members, the public keyword is used, followed by a colon. All functions and variables declared after the public specifier are accessible both by other members of the class and by any part of the program that contains the class. For Example consider the class given below


```
class number
{
        int a,b;
        public:
                void get_ab();
                void put_ab()
};
```

This class has two private variable, called a and b, and two public functions get_ab( ) and put_ab( ). Notice that the functions are declared within a class using their prototype forms. The functions that are declared to be part of a class are called member functions.

3.2.1 Defining Member Functions

Member functions can be declared either inside/outside the class. If the member function is declared inside the class it is said to be inline functions. Inline functions are those whose body is expanded at that place where the function was called while in simple functions the control transfer to the called function when call is made and after executing function's body the control moves to the calling function. When the function is declared outside the class it should follow the syntax given below.

```
<return type> classname :: functionname(argument list)
{
        //body of the function.
}
```

For example the function get_ab(),put_ab() of the number class can be written as.

```
void number :: get_ab()
{
        cin>>a;
        cin>>b;
}
void number :: put_ab()
{
        cout<<a;
        cout<<b;
}
```

3.2.2 Creating Objects

The declaration of object in C++ is similar to declaration of variables .To create an object, use the class name as type specifier. The Syntax is <class name> objectname . For example,

```
void main( )
{
        number ob1, ob2;//these are object of type number
        // ... program code
}
```


Remember that an object declaration creates a physical entity of that type. That is, an object occupies memory space, but a type definition does not.

3.2.3 Accessing Class Members

Once an object of a class has been created, the program can reference its public members by using the dot operator in much the same way that structure members are accessed. The syntax for accessing the object is given below.

<object name>.functionname(arguments);

The statement given below invokes the get_ab() and put_ab() member functions of the number class.

```
ob1.get_ab();
ob1.put_ab();
```

Example 1:

The following code creates a class called number with member functions get_ab(),add() and mul().

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

```
        private:
                int a,b;
        public:
                void get_ab()
                {
                        cout<<"Enter values for a and b \n";
                        cin>>a;
                        cin>>b;
        }
        void add();
        void mul();
};
void number :: add()
{
        int c;
        c=a+b;
        cout<<"THE SUM IS "<<c;
}
void number :: mul()
{
        int d;
        d=a*b;
        cout<<"THE PRODUCT IS "<<d;
}

void main()
{
        number ob1;
        ob1.get_ab();
        ob1.add();
        ob1.mul();
}
```


Output of the program:

Enter Values for a and b

5 6

THE SUM IS 11

THE PRODUCT IS 30

This program creates a class called number with two private data members and three public member functions. The function get_ab() is an inline function. The functions add() and mul() are written outside the class. The

Program creates one object called "ob1" through which the member functions are accessed.

3.2.4 Member Functions with Parameters

It is possible to pass one or more arguments to a member function. Simply add the appropriate parameters to the member function's declaration and definition. Then, when you declare an object, specify the arguments. The program given below makes use of parameterized member functions.

Example 2:

This program is the modified version of example 1,the function get_ab() is modified by adding two parameters.

```
#include<iostream.h>
#include<iomanip.h>
class number
{
        private:
                int a,b;
        public:
                void get_ab(int x ,int y) // member function with parameters
                {
                        a=x;
                        b=y;

                }

        void add()
        {
                int c;
                c=a+b;
                cout<<"THE SUM IS "<<c<<endl;
        }
        void mul()
        {
                int c;
                c=a*b;
                cout<<"THE PRODUCT IS "<<c;
        }
};
void main()
{
        number ob1;
        int d1,d2;
        cout<<"Enter two values"<<endl;
        cin>>d1>>d2;
        ob1.get_ab(d1,d2); // d1 value is copied to x and d2 value is copied to y
```

```
        ob1.add();
        ob1.mul();
}
```

Output of the program:

Enter two values

10 20

THE SUM IS 30

THE PRODUCT IS 200

3.2.5 Inline Functions outside Classe

A member function that is declared outside can be made inline just by prefixing it with the keyword inline.

Example 3:

The following is an example for an inline function written outside the class.

```
#include<iostream.h>
class point
{
        private:
                int x,y;
        public:
                void getpoints();
                void putpoints();
};
inline void point :: getpoints()
{
        cout<<"Enter 2 points\n"
        cin>>x>>y;
}
inline void point :: putpoints()
{
        cout<<"The points are\n";
        cout<<x<<"\t"<<y;
}
void main()
{
        point p;
        p.getpoints();
        p.putpoints();
}
```

Output of the program:

Enter 2 Points

11 20

The Points are

11 20

Have you Understood Questions?

1. What is a class and what is an object?
2. How will you make function written outside the class as inline.
3. Write the syntax to create an object for the class

3.3Constructors

Generally every object we create will require some sort of initialization. C++ allows a constructor function to be included in a class declaration. A class's constructor is called each time an object of that class is created. Thus, any initialization to be performed on an object can be done automatically by the constructor function. A constructor function has the following characteristics.

- It should have the same name as that of the class.
- It should not have any return type even void however they can take arguments.
- Constructers can take default arguments
- Constructers can be dynamically initialized.
- Constructer function cannot be made virtual
- It should be always declared in the public section of the class.

Example 4:

The following program makes use of a constructor.

```
#include <iostream.h >
class number
{
        int a;
        public:
                number( ); //constructor
                void show( );
};
number::number( )
{
        cout << "In constructor\n";
        a=100;
}
void number::show( )
```

```
{
        cout << a;
}
void main( )
{
        number ob; // automatic call to constructor
        ob.show( );
}
```

Output of the Program
In Constructor
100
In this simple example the constructor is called when the object is created, and the constructor initializes the private variable a to 100 .For a global object, its constructor is called once, when the program first begins execution. For local objects, the constructor is called each time the declaration statement is executed.

3.3.1 Constructors That Take Parameters

It is possible to pass one or more arguments to a constructor function. Simply add the appropriate parameters to the constructor function's declaration and definition. Then, when you declare an object, specify the arguments.

Example 5:

The following program makes use of a parameterized constructor.


```
# include < iostream.h >
class number
{
        int a;
        public:
                number(int x); //constructor
                void show( );
};
number::number(int x)
{
        cout << "In constructor\n";
        a=x;
}
void number::show( )
{
        cout <<"Value of a is "<< a<<"\n";
}
void main( )
{
        number ob(4);
        ob.show( );
```

```
}
```

Output of the Program

```
In Constructor
Value of a is 4
```

In this example the object ob is parameterized. The value 4, specified in the parentheses following ob, is the argument that is passed to number( )'s parameter x that is used to initialise a. Actually, the syntax is shorthand for this longer form:

```
number ob = number(4);
```

Although the previous example has used a constant value, you can pass an object's constructor any valid expression, including variables.

3.3.2 Constructer Overloading

Similar to the function overloading concept in C++, you can also overload constructers. Constructer overloading helps us to keep multiple constructers in the class, such that each constructer can perform different methods of initialization. Here is an example for constructer overloading.

Example 6:

The following program illustrates the constructor overloading concept. The constructor function number () is overloaded with two, one and no parameters

```
# include < iostream.h >
class number
{
        int a,b;
        public:
                number()
                {
                        cout<<"Default constructer";
                        x=y=0;
                }
                number(int x);
                number(int x,int y);
                void show( );
};
number::number(int x)
{
        cout << "In one parameter constructor\n";
        a=x;
        b=0;
}
number::number(int x,int y)
{
        cout << "In two parameter constructor\n";
        a=x;
```

```
        b=y;
}
void number::show( )
{
        cout << a<<"\n"<<b;
}
int main( )
{
        number ob(); //invokes default constructer
        ob.show( );
        number ob1(5); //invokes constructer with one parameter
        ob1.show( );
        number ob2(5,10); //invokes constructer with two parameters
        ob2.show( );
        return 0;
}
```

Output of the Program

```
Default Constructor
0
0
In one parameter constructor
5
0
In two parameter constructor
5
10
```


3.3.3 Copy Constructers

Copy constructer is used to copy the value of one object to another. The copy constructor takes a single argument of the same type as the class. It creates a new object, which is an (exact) copy of the passed object. This is done by copying all member variables from the passed object into the new object. An example for copy constructer is given below.

Example 7:

The following program makes use of copy constructors

```
# include < iostream.h >
class number
{
        int a;
        public:
                number(int x); //constructor
                number(number & x) //copy constructer
                void show( );
};
```

```
number::number(int x)
{
        cout << "In constructor\n";
        a=x;
}
number :: number(number& x)
{
        cout<<"Copy Constructor\n";
        a=x.a
}
void number::show( )
{
        cout <<"Value of a is "<< a<<"\n";
}
int main( )
{
        number ob(4);
        ob.show( ); //displays 4
        number ob1(ob); //invokes copy constructer
        ob1.show(); //displays 4
        return 0;
}
```

Output of the Program

In Constructor
Value of a is 4
Copy Constructor
Value of a is 4

Always the Copy constructer takes reference variable as a parameter. In this example the copy constructer copies the value of the object ob to object ob1.

Have you Understood Questions?

1. What is a constructer?
2. Is it possible to overload constructer?
3. What is purpose of copy constructer?

3.4Destructors

The complement of a constructor is the destructor. This function is called when an object is destroyed. For example, an object that allocates memory when it is created will want to free that memory when it is destroyed. The name of a destructor is the name of its class preceded by a ~. The destructor function has the following characteristics.

- A destructor function never takes any arguments.

- A destructor function will not return any values.
- Destructors cannot be overloaded
- Destructors can be made virtual.

The following is the example of a destructor.

Example 8:

The following program illustrates the use of destructors.

```
# include <iostream.h >
class number
{
        int a;
        public:
                number( ); //constructor
                ~number( ); //destructor
                void show( );
};
number::number( )
{
        cout << "In constructor\n";
        a=100;
}
number::~number( )
{
        cout << " In Destructor..,\n";
}
void number :: show()
{
        cout<<"a is "<<a<<"\n";
}
void main()
{
        number ob;
        ob.show();
}
```


Output of the Program

```
In Constructor
A is 100
In Destructor…
```

A class's destructor is called when an object is destroyed. Local objects are destroyed when they go out of scope. Global objects are destroyed when the program ends.

Have you Understood Questions?

1. What is a destructor?
2. Is it possible to overload destructor?

3.5MORE ABOUT CLASSES

3.5.1 Passing Objects to Functions

Objects can be passed to functions as arguments in just the same way that other types of data are passed. Simply declare the function's parameter as a class type and then use an object of that class as an argument when calling the function. As with other types of data, by default all objects are passed by value to a function.

Example 9:

In the following program an object is passed as parameter to the member function product.

```
class number
{
        int i;
        public:
                number(int n) { i = n; }
                int product(number o1,number o2);
};
int number :: product(number o1,number o2)
{
        return o1.i( )* o2.i( );
}
int main( )
{
        number a(5), b(2);
        cout << a.product(a,b)<< "\n"; //OUTPUT 10
        cout << b.product(a,b) << "\n"; //OUTPUT 10
        return 0;
}
```


Output of the Program

```
10
10
```

The default method of passing of parameters in C++ is pass by value (including objects).In this example the function product takes two objects as arguments and return their product.

3.5.2 Returning Objects from Functions

Functions can return objects. First, declare the function as returning a class type. Second, return an object of that type using the normal return statement. Remember that when an object is returned by a function, a temporary object is automatically created which holds the return value. It is this object that is actually returned by the function. After the value is returned, this object is destroyed.

Example 10:

The following program illustrates the concept of returning an object from a function. This example add objects of the class money.

```
#include<iostream.h>
class money
{
        int rs;
        int ps;
        money()
        {
                rs=0;
                ps=0;
        }
        money(int r,int p)
        {
                rs=r;
                ps=p;
        }
        money add(money m1,money m2);
        void display();
};
money money :: add(money m1,money m2)
{
        money temp;
        temp.ps=m1.ps+m2.ps;
        if(temp.ps>99)
        {
                temp.rs++;
                temp.ps-=99
        }
        temp.rs+=m1.rs+m2.rs;
        return temp;
}
void money :: display()
{
        cout<<"Total Rupees "<<rs;
        cout<<"\n Total Paise "<<ps;
}
void main()
{
        money m1(10,55);
        money m2(11,55);
        money m3;
        m3.add(m1,m2);
```

```
        m3.display();
}
```

Output of the Program

Total Rupees22
Total Paise10

Have you Understood Questions?

1. Objects are passed in call by _____ method

3.6 Static Data Members

A variable can be declared as static by prefixing it with a key word 'static'. A static variable in C++ will work in the same way as the 'C' language static variable. The other name for static variable is class variable. A static member is part of a class, but not part of an object of a class, so there's only one instance of them however many objects of that class are created. This is useful if, for instance, you want to keep a count of how many objects are created. A static data member must be defined outside the class definition. Static variable will have the following properties:

- It is always initialized to a default value.
- A static variable is common to the entire class, that is, only one copy of the static variable exists.
- All the objects will share the static variable.

Example 11:

The example given below will count the number of objects created for the class one.

```
#include<iostream.h>
class one
{
        private:
        static int count;
        public:
                void incr()
                {
                        count ++;
                }
                void display( )
                {
                        cout <<count<<" objects";
                }
};
int one :: count ; //Static data member definition
void main()
{
        one 01, 02, 03;
```

```
    01. incr ();
    02. incr ();
    03. incr ();
    03. display ();
}
```

Output of the Program
3 Objects

In this example a variable count is declared as a static data member. It will be initialized to zero. Every time when the function incr( ) is called the variable is incremented by one irrespective of the object.

3.6.1 Static Member Functions

A member function can be declared as static by prefixing it with the keyword "static". A static function will have the following rules.

- A static function can access only static variables.
- A static function will be called using the class name.
- A static function cannot access non-static variables.
- A static function cannot call non static functions.
- A static function is common to the entire class.

Example 12:
The following program illustrates the concept of static member functions.

```
#include<iostream.h>

#include<conio.h>

class one

{

    private:

        static int count;

    public:

        void set()

        {

            count=5;

        }

        static void incr()
```

```
        {
            count++;
            cout<<"value of count via static member:"<<count;
        }

};
int one::count;
void main()
{
    one obj1, obj2, obj3;
    obj1.set ();
    obj2.set ();
    obj3.set ();
    one::incr(); // calling a static function.
    getch();
}
```


Output of the Program

Value of count via static member:6

Have you Understood Questions?

1. What is the default value for static integer value?

2. How will you make a variable static?

3. A static function can access only static variables (TRUE/FALSE)

4. Assume that c1 is a class and fn() is a static function then how will you call the function f1()

3.7 Const Members Functions

Class member functions may be declared as const. When this is done, that method cannot modify the object that invokes it. Also, a const function/method may not invoke a non-const member function of the same object. However a const method might call non-const methods of other classes or of other objects of the same class as itself. Further, a const class can be called from either const or non-const objects. To specify a member function as const, use the form shown in the following example:

```
class ex1
{
        private:
                int x;
        public:
                int fun() const; // const member function
};
```

The purpose of declaring a member functions as const is to prevent it from modifying the object that invokes it.

Have you Understood Questions?

1. A constant object is required to invoke a constant class (TRUE/FALSE)

3.8Friend Functions

There will be time when you want a function to have access to the private members of a class without that function actually being a member of that class. C++ supports friend functions to achieve this purpose. A friend function is not a member of a class but still has access to its private elements. Friend functions are useful with operator overloading and the creation of certain types of I/O functions. Friend Functions possess the following features.

- The function that is declared as friend will not fall in the scope of the class it was declared.
- A friend declaration can be placed in either the private or the public part of a class declaration
- Like a member function, a friend function is explicitly declared in the declaration of the class of which it is a friend
- A friend function can access private data members of that class.
- A friend function will be called without the object.
- Usually friend function takes object as an argument.

The Syntax for creating a friend function is given below.

```
class sample
{
        private:
                ----
                ----
        public:

                ----
                ----
                friend void callme( );
};
```

In the main program the friend function can be called as a ordinary function without any objects.

```
void main()
{
        sample s;
                -----
                -----
                callme( );
}
```

Example 13:

The following is an example for friend function

```
#include<iostream.h>
#include<conio.h>
class one
{
        private:
                int a,b;
        public:
                one()
                {
                        a=20;
                        b=30;
                }
                friend float avg(one obj);
};
float avg(one obj)
{
        float r;
        r=(obj.a+obj.b)/2;
        return r;
}
void main()
{
```

```
        one e;
        float m;
        m=avg(e);
        cout<<" The Average is: "<<m;
        getch();
}
```

Output of the Program

The Average is:25

In this example the function avg( ) is declared as friend to the class one. As mentioned earlier this friend function takes object as a parameter of type the class “one”. Note that the definition of the function avg( ) is written like a ordinary function this is because the friend function will not fall in the scope of the class where it is defined. Finally the function is invoked without the help of the object.

3.8.1 Friendly Function for Two Classes

In C++ it is possible to declare member function of one class as a friend function of other class in this case we say that the two classes are friendly to each other and this concept is called as friend class.

To create friendly classes you need to create a function that is friend to both the classes. Further you need to do forward declaration of the class that you are going to define later.

Forward declaration is a concept in which we declare a class before defining it. Forward declaration intimates the compiler that the class declared is defined in the later part of the program. Here is an example for forward declaration.


```
class test2; //Forward Declaration
class test1
{
        ----
        -----
        -----
};
class test2 //Definition of the class that has the forward declaration.
{
        ----
        ----
        ----
}
```

Example 14:

The following example creates two friendly classes.

```
#include<iostream.h>
#include<conio.h>
class c2; //Forward Declaration
class c1
{
        private:
                int x;
        public:
                c1()
                {
                        cout<<"Enter the value of first number:";
                        cin>>x;
                }
                friend float avg(c1 o1,c2 o2);
};
class c2
{
        private:
                int y;
        public:
                c2()
                {
                        cout<<"Enter the value of second number:";
                        cin>>y;
                }
                friend float avg(c1 o1,c2 o2);
};
float avg (c1 o1,c2 o2)
{
        int ans;
        ans= (o1.x +o2.y) / 2;
        return ans;
}
void main()
{
        clrscr();
        c1 a;
        c2 b;
        float xyz;
        xyz= avg(a,b);
        cout<<xyz;
        getch();
```

```
}
```

Output of the Program

```
Enter the value of first number 20
Enter the value of second number:10
15
```

In this example the function avg( ) is declared as friend to the classes c1 and c2. Since the class c2 is written after class c1 we have to forward declaration for c1. The friend function avg( ) computes the average of the variables x and y declared in classes c1 and c2 respectively. Since avg( ) is a friend function it is called without the help of the object.

Have you Understood Questions?

1. In which situation you will declare a function as friend ?
2. How will you call a friend function ?
3. What is forward declaration?

3.9Summary

- Class is an user defined data type in C++. Classes in C++ are also called as Abstract Data Type (ADT) .Classes bind the data with its associated code.
- Member functions can be declared either inside/outside the class. If the member function is declared inside the class it is said to be inline functions.
- In C++ class variables are called as objects. Objects are run time entities of an object oriented system.
- A constructer is a special member function that has the same name as the class. A class's constructor is called each time an object of that class is created.
- Constructers will not have return type but can take parameters, hence constructers can be overloaded.
- Copy constructer is used to copy the value of one object to another.
- The complement of a constructor is the destructor. This function is called when an object is destroyed.
- A static member is part of a class, but not part of an object of a class, so there's only one instance of them however many objects of that class are created.
- A static member function can access only static variables and they are called using the class name.
- Class member functions may be declared as const. When this is done, that method cannot modify the object that invokes it.
- A friend function is not a member of a class but still has access to its private elements.
- Forward declaration is a concept in which we declare a class before defining it. Forward declaration intimates the compiler that the class declared is defined in the later part of the program

3.10 Exercises

Short Questions

1. When we create a new class a new built in data type is created (TRUE/FALSE)
2. Mention the properties of constructers and destructors.
3. Destructors can be made virtual TRUE/FALSE.
4. Explain Copy constructers with example
5. A constructer can take parameter but it cannot have_____
6. List out any three properties of static member functions.
7. A static member function can access only_____ data members
8. In which situation you will declare a member function as constant member function.
9. Is friend function violating OOPS principle of data hiding? Discuss.
10. By default friend functions will have this pointer (TRUE/FALSE)

Long Questions

1. Explain how objects can be passed and returned from a function with an example
2. Explain the concept of constructer overloaing
3. Explain the concept of Friend functions with example.
4. Explain briefly regarding static data members and static functions.

Programming Exercises

1. Create a class called calculator and implement all basic functions of the calculator as member functions.
2. Create a class called Product with prod_name, prod_id, cost, quantity as data members create necessary members functions constructers and destructors for the class
3. Write a program to add two weights (Weights are represented in Kilograms and Grams) use passing and returning objects concept.
4. Write a friend function to add objects of two complex numbers.

Answers to Have you Understood Questions

Section 3.3

1. Class is another user defined data type in C++. Classes in C++ are also called as Abstract Data Type (ADT) .Classes binds the data with its associated code. Objects are run time entities of a class
2. Just by prefixing the keyword inline before the function prototype.
3. <class name> object; Eg. number obj

Section 3.4

1. Constructers are special member functions that have the same as the class. They are called when we create an object for a class.
2. Yes
3. Copy constructer is used to copy the value of one object to another.

Section 3.5

1. The complement of a constructor is the destructor. This function is called when an object is destroyed.
2. No.

Section 3.6

1. Value

Section 3.7

1. 0
2. Just by prefixing with the keyword static before variable declaration
3. True.
4. c1::fn()

Section 3.8

1. False

Section 3.9

1. When we want a function to have access to the private members of a class without that function actually being a member of that class.
2. Friend functions are called like an ordinary member function without the help of the object.
3. Forward declaration is a concept in which we declare a class before defining it. Forward declaration intimates the compiler that the class declared is defined in the later part of the program.

Chapter 4

Operator Overloading and Type Conversions

Structure of the Unit

- Operator Overloading
- Defining Operator Overloading
- Overloading Binary Operator
- Overloading Unary Operator
- Overloading Prefix/Postfix Increment/ Decrement Operator
- Operator Overloading using Friend Functions
- Type conversion
- Implicit Type Conversion
- Explicit Type Conversion
- Basic to Class Type
- Class to Basic Type
- One class type to another class type

Learning Objectives

- To introduce the concept of operator overloading
- To define operator overloading function
- To understand the steps involved in overloading binary operator
- To understand the steps involved in overloading unary operators
- To show how to overload unary prefix and postfix operator
- To discuss overloading using friend functions
- To present the concept of type casting
- To discuss implicit and explicit type casting

4.1Introduction

Operator overloading refers giving special meaning to an existing operator. Operator overloading is similar to function overloading. In fact, operator overloading is really just a type of function overloading. However, some additional rules apply. This chapter introduces you to operator overloading that deals with overloading of both unary and binary operators. The second part of the chapter provides necessary details regarding type conversions or type casting. Type conversion or typecasting refers to changing an entity of one data type into another. This chapter focuses on implicit and explicit type conversions.

4.2Operator Overloading

As mentioned earlier operator overloading refers giving special meaning to an existing operator. When an operator is overloaded, that operator loses none of its original meaning. Instead, it gains additional meaning relative to the class for which it is defined. An operator is always overloaded relatively to a user defined type, such as a class. To overload an operator, you create an operator function. Most often an operator function is a member or a friend of the class for which it is defined.

Almost all operators in C++ can be overloaded, except the following few operators.

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

4.2.1 Defining Operator Overloading

The general form of a member operator function is shown here:

```
return-type class-name::operator#(arg-list)
{
    // operation to be performed
}
```

- The return type of an operator function is often the class for which it is defined (however, operator function is free to return any type).
- The operator being overloaded is substituted for #. For example, if the operator + is being overloaded, the operator function name would be operator +.
- The contents of arg-list vary depending upon how the operator function is implemented and the type of operator being overloaded.

There are two important restrictions to remember when you are overloading an operator:

- The precedence of the operator cannot be change.
- The number of operands that an operator takes cannot be altered.

4.2.2 Overloading Binary Operator

When a member operator function overloads a binary operator, the function will have only one parameter. This parameter will receive the object that is on the right side of the operator. The object on the left side is the object that generates the call to the operator function and is passed implicitly by **this pointer**.

Example 1:

The following program overloads the "+" operator such that it adds two objects of the type complex.

```
#include <iostream.h>
#include<conio.h>
class complex
{
        int real;
        float imag;
        public:
                complex()
                {
                        real=imag=0;
                }
                complex (int r,float i)
                {
                        real = r; imag=i;
                }
                //overload + operator for complex class
                complex operator +(complex);
                void display()
                {
                        cout<<"The Result is ";
                        cout<<real <<" +j " <<imag ;
                }
};
complex complex :: operator + (complex c1)
{
        complex temp;
        temp.real=real +c1.real;
        temp.imag=imag + c1.imag;
        return temp;
}
void main()
{
        clrscr();
        complex c1,c2,c3;
        int a; float b;
        cout<<"Enter Real and Imaginary Part of First ComplexNumber:"<<endl;
        cin>>a>>b;
        c1=complex(a,b);
        cout<<"Enter Real and Imaginary Part of Second Complex
Number:"<<endl;
        cin>>a>>b;
```

```
        c2=complex(a,b);
        //adds two complex objects invokes operator + (complex) function
        c3=c1 + c2 ;
        c3.display();
        getch();
}
```

Output of the Program

```
Enter Real and Imaginary Part of First Complex Number 2 3
Enter Real and Imaginary Part of Second Complex Number 3 4
The result is:5+7j
```

Example 2:

The program given below is the modified version of the example 10 program in Chapter3. The member function add() present in that program is replaced by the operator overloading function.

```
#include<iostream.h>
#include<conio.h>
class money
{
        int rs;
        int ps;
        public:
                money()
                {
                        rs=0;
                        ps=0;
                }
                money(int r,int p)
                {
                        rs=r;
                        ps=p;
                }
                money operator +(money);
                void display();
};
money money :: operator +(money m1)
{
        money temp;
        temp.ps=m1.ps+ps;
        if(temp.ps>99)
        {
                temp.rs++;
                temp.ps-=99;
        }
        temp.rs+=m1.rs+rs;
```

```
        return temp;
}
void money :: display()
{
        cout<<"Total Rupees: "<<rs;
        cout<<"\n Total Paise "<<ps;
}
void main()
{
        clrscr();
        money m1(10,55);
        money m2(11,55);
        money m3;
        m3=m1+m2;
        m3.display();
        getch();
}
```

Output of the Program

Total Rupees :22
Total Peise:11

4.2.3 Overloading Unary Operator

Overloading a unary operator is similar to overloading a binary operator except that there is one operand to deal with. When you overload a unary operator using a member function, the function has no parameters. Since, there is only one operand, it is this operand that generates the call to the operator function. There is no need for another parameter.

Example 3:

The following program overloads the unary – operator that negates the object of point class

```
#include <iostream.h>
#include<conio.h>
class point
{
        int x, y;
        public:
                point()
                {
                        x = 0;
                        y = 0;
                }
                point(int i, int j)
                {
                        x = i;
```

```
                y = j;
        }
        void operator-( );
        void display()
        {
                cout<<"value of x is:"<<x <<"\tValue of y is:"<<y;
        }
};
// Overload - operator for point class
void point::operator-( )
{
        x=-x;
        y=-y;
}
void main( )
{
        clrscr();
        point o1(10, 10);
        int x, y;
        -o1; //Negate the object
        o1.display();
        getch();
}
```

Output of the Program

Value of x is:-10 Value of y is:-10

4.2.4 Overloading Prefix and Postfix Increment/Decrement Operators

When overloading the prefix ++ operator you write the member function as

void operator ++( )

Similarly when overloading the postfix ++ operator we will write the member function as

void operator ++( )

Hence, there is an ambiguity because both the member function takes the same form the same problem exist for the pre and post fix decrement operators. To resolve this you have to pass a dummy integer argument for the postfix operator.

```
void operator ++( ); //UNARY PREFIX INCREMENT
void operator -- ( ) ; //UNARY PREFIX DECREMENT
void operator ++(int); //UNARY POSTFIX INCREMENT
void operator -- ( int ) ; //UNARY POSTFIX DECREMENT
```

Example 4:

The program given below overloads unary prefix ++ operator to increment the object of type point class

```
#include <iostream.h>
#include<conio.h>
class point
{
        int x, y;
        public:
                point()
                {
                        x = 0;
                        y = 0;
                }
                point(int i, int j)
                {
                        x = i;
                        y = j;
                }
                void operator++( );
                void display()
                {
                        cout<<"The Object is Incremented to \n";
                        cout<<"First object:"<<x<<"\n";
                        cout<<"Second Object:"<<y;
                }
};
void point::operator++( ) // Overload prefix ++ operator for point class
{
        ++x;
        ++y;
}
void main( )
{
        clrscr();
        int a,b;
        cout<<"Enter two values:";
        cin>>a>>b;
        point o1(a,b);
        ++o1; //increment an object
        o1.display();
        getch();
}
```


Output of the Program

Enter two values: 10 20
The Object is incremented to
First Object:11

Second Object:21

Example 5:

The program given below overloads unary postfix -- operator to decrement the object of type point class

```
#include <iostream.h>
#include<conio.h>
class point
{
        int x, y;
        public:
                point( )
                {
                        x = 0;
                        y = 0;
                }
                point(int i, int j)
                {
                        x = i;
                        y = j;
                }
                void operator--();
                void display()
                {
                        cout<<"The Object is Decremented to \n";
                        cout<<x<<"\t";
                        cout<<y;
                }
};
void point::operator--() // Overload postfix -- operator for point class
{
        x--;
        y--;
}
void main( )
{
        clrscr();
        point o1(10, 10);
        --o1; //decrement an object
        o1.display();
        getch();
}
```


Output of the Program

Enter two values: 10 20

The object is Decrement to

First Object:9
Second Object:19

4.2.5 Operator Overloading Using Friend Functions

It is possible to overload an operator relative to a class by using a friend rather than a member function. A friend function does not have a this pointer. In the case of a binary operator, this means that a friend operator function is passed both operands explicitly. For unary operators, the single operand is passed. We cannot use a friend to overload the assignment operator. The assignment operator can be overloaded only by a member operator function.

Example 6:

The following program overloads + operator using friend function.

```
#include <iostream.h>
#include<conio.h>
class point
{
        int x, y;
        public:
                point()
                {
                        x = 0;
                        y = 0;
                }
                point(int i, int j)
                {
                        x = i;
                        y = j;
                }
                void display()
                {
                        cout << "X IS : " << x << ", Y IS : " << y <<"\n";
                }
                friend point operator+(point ob1, point ob2);
};
// Overload + using a friend.
point operator+(point ob1, point ob2)
{
        point temp;
        temp.x = ob1.x + ob2.x;
        temp.y = ob1.y + ob2.y;
        return temp;
}
int main( )
{
```

```
        clrscr();
        point o1(10, 10), o2(5, 3), o3;
        o3 = o1 + o2; //add to objects
        // this calls operator+( )
        o3.display();
        getch();
        return 0;
}
```

Output of the Program

X is:15
Y is:13

Note that the left operand is passed to the first parameter and the right operand is passed to the second parameter.

Have you Understood Questions?

1. When we overload a operator will its meaning change?
2. Give the general form of operator function
3. Write examples for unary and binary operators
4. Which operator cannot overloaded using friend function?

4.3TYPE CONVERSIONS

Type conversion or typecasting refers to changing an entity of one data type into another. This is done to take advantage of certain features of type hierarchies. For instance, values from a more limited set, such as integers, can be stored in a more compact format and later converted to a different format enabling operations not previously possible, such as division with several decimal places worth of accuracy. Type conversion allows programs to treat objects of one type as one of their ancestor types to simplify interacting with them.

4.4 Implicit Type Conversion

Implicit type conversion is an automatic type conversion by the compiler. The type conversions are automatic as long as the data types involved are built-in types.

Example

```
int y;
float x=123.45;
y = x;
```

In this example the float variable x is automatically gets converted to int. Thus the fractional part of y gets truncated.

4.5 Explicit Type Conversion

Automatic type conversion for user defined data types is not supported by the compiler hence the conversion routines are ought to be specified explicitly.

Three types of situations might arise in the data conversion between incompatible types.

- Conversion from basic type to class type.
- Conversion from class type to basic type
- Conversion from one class type to another class type.

4.5.1 Basic to Class Type

This is done by using constructors in the respective classes. In these types the '=' operator is overloaded.

Example 7:

The following program converts integer to the class type "length"

```
#include<iostream.h>
#include<conio.h>
class length
{
        int cm,m;
        public:
                length(int n) //Constructor
                {
                        m=n/100;
                        cm=m%100;
                }
};
void main()
{
        clrscr();
        int n;
        cin >> n;
        length obj = n; //Integer to class type
        getch();
    }
```

The constructor used for type conversions take a single argument whose type is to be converted.

4.5.2 Class to Basic Type

Overloaded casting operator is used to convert data from a class type to a basic data type.

Syntax:

```
operator typename()
{
    statements.....
}
```

This function converts a class type to specified type name. For example operator int() converts the class object to integer data type.
Conditions for a casting operator function

- It must be a class member.
- It must not specify a return type.
- It must not have any arguments.

Example 8:
The following example cast the object of type "time" to basic data types (int and float)

```
#include<iostream.h>
class time
{
        int min,sec;
        public:
                time(int n)
                {
                        min = n/60;
                        sec=n%60;
                }
                operator int() //Casting Operator
                {
                        int x;
                        x= min * 60 + sec;
                        return sec;
                }
                operator float() //Casting Operator
                {
                        float y;
                        y = min + sec/60;
                        return y;
                }
};
void main()
{
        time t1(160);
        int n = t1; //Conversion
        float x = t1; //Conversion
}
```


4.5.3 One Class to another Class Type

The Constructors helped us to define the conversion from a basic data type to class type and the overloaded casting operator helped to define the conversion from a class type to basic data type. Now to convert from one class type to another class type these both help us to do.

Example 9:

The following program converts the class type length1 to type length2

```
class length2; //forward declaration
class length1
{
        int m,cm;
        public:
                length1(int n)
                {
                        m=n/100;
                        cm=n%100;
                }
                operator length2() //from length1 type to length2 type
                {
                        int x= m*100 + cm;
                        length2 tempobj(x);
                        return tempobj;
                }
}
class length2
{
        int cm;
        public:
        length2(int n)
        {
                cm=n;
        }
        operator length1() //from length2 type to length1 type
        {
                        length1 tempobj(cm);
                        return tempobj;
                }
};
void main()
{
        int x= 125;
        length2 obj1(x);
        length1 obj2= obj1;
}
```


Have you Understood Questions?

1. What is type casting?
2. Which operator you have to overload to convert a basic data type to user defined data type?

4.6 *Summary*

- Operator overloading refers giving special meaning to an existing operator. When an operator is overloaded, that operator loses none of its original meaning. Instead, it gains additional meaning relative to the class for which it is defined.
- We cannot overload Class member access operators, Scope Resolution operator, Size operator(sizeof) , Conditional operator (? :).
- When a member operator function overloads a binary operator, the function will have only one parameter.
- When you overload a unary operator using a member function, the function has no parameters.
- To overload unary postfix operators a dummy integer is passed as a parameter to the operator function
- It is possible to overload an operator relative to a class by using a friend rather than a member function
- We cannot use a friend to overload the assignment operator.
- Type conversion or typecasting refers to changing an entity of one data type into another
- Implicit type conversion is an automatic type conversion by the compiler. The type conversions are automatic as long as the data types involved are built-in types.
- Explicit type conversion is necessary for user defined data types.

4.7Exercises

Short Questions

1. List out the operators that cannot be overloaded in C++>
2. Write the various rules for overloading operators.
3. Write the advantage of overloading operators using friend functions.
4. What do you mean by implicit type conversion?

Long Questions

1. Explain how prefix and postfix operators are overloaded
2. Explain Explicit type conversion methods with example.

Programming Exercises

1. Write a program to overload binary operator + such that it is capable of adding two objects of “distance” class .The distance is expressed as foot and inches. .
2. Write a program to overload ++ operator to increment time object. Time is expressed in hour, minute and second.

Answers to Have you Understood Questions

Section 4.3

1. No
2. return-type class-name::operator#(arg-list)

```
{
        // operation to be performed
}
```

3. Unary operator Eg !. Binary operator Eg. +
4. =(assignment operator)

Section 4.4

1. Type conversion or typecasting refers to changing an entity of one data type into another.
2. =(assignment operator)

Chapter 5

Inheritance

Structure of the Unit

- Overview of Inheritance
- Single Inheritance
- Access Control
- Inheritance and Access Control
- Inheritance in Private Mode
- Using Protected Members
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance
- Constructors and Destructors in Derived Class
- Parameterized Constructor in Derived Class

Learning Objectives

- To introduce inheritance concept
- To present various access control mechanism in C++
- To discuss single inheritance and its implementation
- To discuss multiple inheritance and its implementation
- To discuss multilevel inheritance and its implementation
- To discuss hierarchical inheritance and its implementation
- To discuss hybrid inheritance and its implementation
- To show how constructor and destructors gets executed in inheritance

5.1 Introduction

This chapter explores one of the most powerful features of C++ language, the inheritance. C++ strongly supports the concept of reusability with the help of inheritance. Inheritance is one of the primary features of any object oriented language. In this chapter you will go through the various forms of inheritance and you will learn how to implement them in C++ programs further this chapter will introduce you the access specifiers and you will learn how to apply them in your programs.

5.2 Overview Of Inheritance

Inheritance is a feature by which new classes are derived from the existing classes. Inheritance allows re-using code without having to rewrite from scratch. The parent class(Original existing class) is said to be the base class the new made class from it is called a derived class A class that is derived from parent

class will possess all the properties of the base class apart from that it also has its own unique properties. However the derived class will not inherit constructers, destructors, friend functions and assignment operators further derived classes cannot remove any data member present in the base class.C++ supports various types of inheritance scheme. They are

- Single inheritance
- Multiple inheritance
- Multilevel inheritance
- Hierarchical inheritance
- Hybrid inheritance

Have you Understood Questions?

1. What is Inheritance?
2. Mention the various inheritance schemes supported in C++.

5.3Single Inheritance

When a class is derived from a single base class it is said to be single inheritance. It can be thought of as a specialization of an existing class.

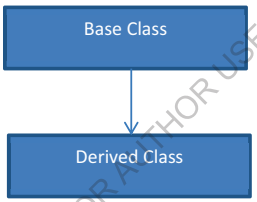


Fig 5.1 Single Inheritance

The syntax for implementing inheritance is

class derivedclassname : access mode baseclassname

The access mode may be public, protected or private.

Example

```
class base
{
        ----
        ----
};
class derived : public base
{
        -----
        -----
};
```

In this example the class derived inherits the properties of the class base. The inheritance is made in the public mode.

Example 1

The following code is a simple example for single inheritance. In this example the derived class (number2) inherits the base class (number1) in public mode

```
#include<iostream.h>
#include<conio.h>
class number1
{
        int a;
        public:
                number1( )
                {
                        a=0;
                }
                void get_a()
                {
                        cout<<"Enter the value for a \n";
                        cin>>a;
                }
                void show_a()
                {
                        cout<<"a is "<<a;
                }
};
class number2: public number1
{
        private:
                int b;
        public:
                number2( )
                {
                        b=0;
                }
                void get_b()
                {
                        cout<<"Enter the value for b \n";
                        cin>>b;
                }
                void show_b()
                {
                        cout<<"b is "<<b;
                }
```

```
};
void main ( )
{
        clrscr();
        number2 obj;
        obj. get_a ( );
        obj. get_b ( );
        obj.show_a( );
        obj.show_b( );
        getch();
}
```

Output of the Program

```
Enter the value for a:5
Enter the value for b: 10
a is 5
b is 10
```

Here the inheritance is made in the public visibility mode. In the main program we create the object for derived class and access all public functions present in the base class. The derived class number2 will have the following features.

- **From the class number2**

Public Functions	Private Variables
number2 //constructor void get_b() void show_b()	int b

- **From the class number1**

Public Functions	Private Variables
void get_a() void show_a()	None.

Have you Understood Questions?

1. Draw the scheme for single inheritance
2. Write the syntax for implementing single inheritance.

5.4Access Control

An access specifier in C++ determines which data member, which member function can be used by other classes. The three access specifiers supported in C++ are

- The private access specifier
- The public access specifier
- The protected access specifier

Only objects of the same class can have access rights to a member function or data member declared as private.

A data member or a method declared as public can be accessed by all classes and their objects.

Data members, methods that are declared protected are accessible by that class and the inherited subclasses

The program given below uses all C++ access specifiers

```
#include<iostream.h>
#include<conio.h>
class Base
{
	private:
		int privateInt ;
	protected:
		int protectedInt;
	public:
		int publicInt;
		Base()
		{
			privateInt =10;
			protectedInt=20;
			publicInt=30;
		}
		void display()
		{
			cout<<"Private Variable:"<<privateInt;
			cout<<"\tProtected Variable:"<<protectedInt;
			cout<<"\tPublic Variable:"<<publicInt;
		}
};
class Derived :public Base
{
	public:
		void change()
		{
			//privetInt++ Wrong! Private variables cannot be accessed
			protectedInt++; //OK
			publicInt++; //OK


		}
};
void main()
{
	clrscr();
	Derived obj;
```

```
        //obj. privetInt=50 ; Wrong! Private variables cannot be accessed
        //obj. protectedInt=50;  Wrong! Protected variables cannot be accessed
        obj.publicInt=50; //OK
        obj.display();
        getch();
}
```

Output of the Program

Private Variable:10 Protected Variable:20 Public Variable:50

5.4.1 Inheritance and Access Control

The protection keyword after the ':' operator in the derived class determines the access by derived class to base class members. For example consider the following scheme of inheritance.

1. class derived : private base

All public members of the base class become private to the derived class,i.e. only members of the derived class can access them. All private members of the base class are private to the base class, i.e. nothing can access them, not even members of the derived class. All protected members become also private to the derived class.

2. class derived : protected base

All public members of the base class become protected, i.e. only the derived class (and further derived classes) can access them. All private base class members are private to base class, i.e. nothing can access them. All protected members of the base class stay protected, i.e. only the derived class (and further derived classes) can access them.

3. class derived : public base

All public members of the base class are public in the derived class, i.e.anything can access them. All private members of the base are private to the base class, i.e. nothing can access them. All protected members of the base class stay protected, i.e. only the derived class (and further derived classes) can access them

5.4.2 Inheritance in Private Mode

We know that when the access specifier is private public members of the base become private members of the derived class, but these are still accessible by member functions of the derived class.

Example 2:

The program given below is an example for inheritance in private mode.

```
#include <iostream.h>
#include<conio.h>
class base
{
```

```
        int x;
        public:
                void setx(int n)
                {
                        x = n;
                }
                void showx( )
                {
                        cout <<"Value of x:"<<x << "\n";
                }
};
// Inherit base as private
class derived : private base
{
        int y;
        public:
        void sety(int n)
        {
                y = n;
        }
        void showy( )
        {
                cout <<"Value of y:"<<y << "\n";
        }
};
int main( )
{
        clrscr();
        derived ob;
        //ob.setx(10); ERROR now private to derived class
        ob.sety(20); // access member of derived class - OK
        //ob.showx( ); ERROR now private to derived class
        ob.showy( ); // access member of derived class - OK
        getch();
        return 0;
}
```


Output of the Program

Value of y:20

As the comments in this program illustrates, both showx( ) and setx() become private to derived and are not accessible outside it. Note that showx( ) and setx( ) are still public within base, no matter how they are inherited by some derived class. This means that an object of type base could access these functions anywhere.

5.4.3 Using Protected Members

As you know from the preceding section, a derived class does not have access to the private members of the base class. However, there will be times when you want to keep a member of a base class private but still allow a derived class access to it. To accomplish this, C++ includes the protected access specifier. The protected access specifier is equivalent to the private specifier with the sole exception that protected members of a base class are accessible to members of any class derived from that base. Outside the base or derived classes, protected members are not accessible.

The protected access specifier can occur any where in the class declaration, although typically it occurs after the (default) private members are declared and before the public members. The full general form of a class declaration is shown here:

```
class class-name
{
        // private members
        protected: //optional
                // protected members
        public:
                //public members
};
```

Example 3:

The following program illustrates a single inheritance concept with protected variables

```
#include<iostream.h>
#include<conio.h>
class base
{
        protected:
                int a; //private to base but accessible from derived
        public:
                void get_a ( )
                {
                        cout<<"Enter the Value for a \n";
                        cin>>a;
                }
};
class derived : public base
{
        int b;
        public:
                void get_b()
                {
                        cout<<"Enter the Value for b \n";
```

```
                cin>>b;
        }
        void product()
        {
                int c;
                c= a * b; // variable a is protected member of class base
                cout<<"The Product is "<<c;
        }
};
void main()
{
        clrscr();
        derived obj;
        obj.get_a();
        obj.get_b();
        obj.product();
        getch();
}
```

Output of the Program

```
Enter the value for a
10
Enter the value for b
20
The Product is 200
```


Have you Understood Questions?

1. If a base class is inherited in private mode, how its public members appears for the derived class
2. If a base class is inherited in public mode, all the members of the class appears as public members to the derived class (TRUE/FALSE)
3. What is the special access specifier used for inheritance?
4. List the access specifiers available in C++.

5.5Multiple Inheritance

C++ allows a class to get derived from multiple base classes. A derived class can directly inherit more than one base class. In this situation, two or more base class is combined to create the derived class. Multiple inheritance is never necessary (some object oriented languages, e.g. java, don't even support multiple inheritance) and should be avoided. However, sometimes it makes programming easier. The multiple inheritance diagram is depicted below.

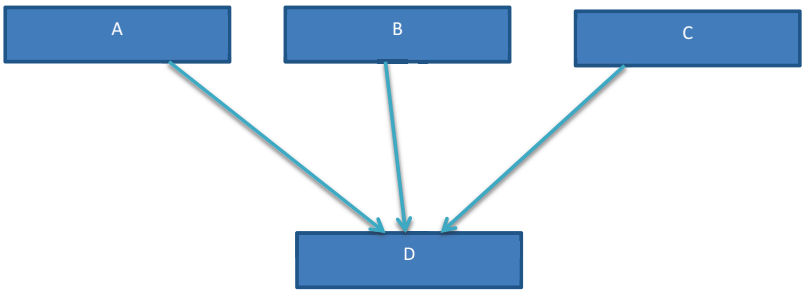


Fig 5.2: Multiple Inheritance

Here the class D inherits the properties of classes A, B and C. The syntax for creating a derived class with multiple inheritance is

class derived classname : access mode base1, access mode base2,…..access mode base n

{

//body of the derived class

}

Example 4:

The following code contains the C++ classes for the multiple inheritance diagram given in fig 5.3

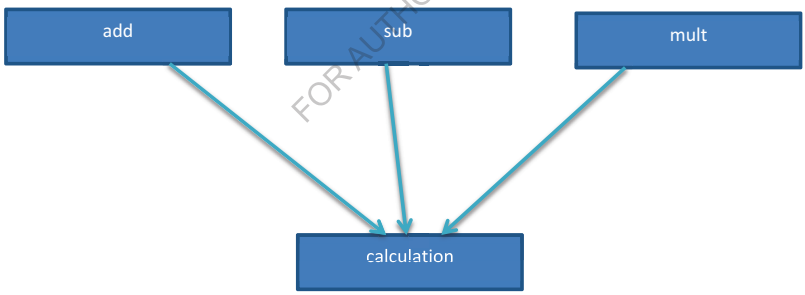


Fig 5.3: Multiple Inheritance(Promotion Details)

```
#include<iomanip.h>
#include<conio.h>
class add
{
        protected:
                int c;
        public:
                void caladd(int a,int b)
```

```
                {
                   c=a+b;
                }
};
class sub
{
        protected:
                int d;
        public:
                void calsub(int a,int b)
                {
                        d=a-b;
                }
};
class mult
{
        protected:
                int e;
        public:
                void calmult(int a,int b)
                {
                        e=a*b;
                }
};
class calculation : public add ,public sub, public mult
{
        public:
                void result( )
                {
                        cout<<"Result:";
                        cout<<"\nAddition is:"<<c;
                        cout<<"\nSubtration is:"<<d;
                        cout<<"\nMultiplication is:"<<e;
                }
};
void main( )
{
        clrscr();
        int a,b;
        calculation obj;
        cout<<"Enter two nummber:";
        cin>>a>>b;
        obj.caladd(a,b);
        obj . calsub(a,b);
```

```
        obj .calmult(a,b);
        obj . result();
        getch();
}
```

Output of the Program

Enter two number: 10 20
Result:
Addition is:30
Subtraction is:-10
Multiplication is:200

Have you Understood Questions?

1. What is multiple inheritance?
2. Multiple inheritance is supported in _____ and not supported in_____

5.6Multilevel Inheritance

Multilevel inheritance is one form of multiple inheritance. In this scheme the derived class is made from two or more base class. Here a derived class can be used as a base class for another derived class, creating a multilevel class hierarchy. In this case, the original base class is said to be an indirect base class of the second derived class.

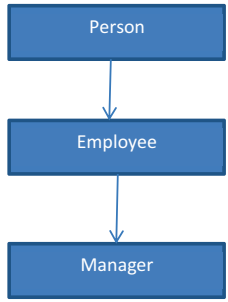


Fig 5.4 Multilevel Inheritance

Thus this scheme can be extended to any level (i.e generalization to specialization).Let us write the program for multilevel inheritance

Example 5:

The following code contains the C++ classes for the multilevel inheritance diagram given in fig 5.4

```
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
#include<string.h>
class person
{
        protected:
                int age;
                char *name;
        public:
                void get1( );
};
class emp : public person
{
        protected:
                int basic,hra;
        public:
                void get2();
};
class manager : public emp
{
        private:
                int deptcode;
        public:
                void get3( );
```

```
                void display( );
};
void person :: get1( )
{
        cout<<"Enter your Age"<<endl;
        cin >> age;
        cout<<"Enter your name"<<endl;
        cin>> name;
}
void emp :: get2( )
{
        cout<<"Enter your Basic and HRA"<<endl;
        cin>>basic>>hra;
}
void manager :: get3()
{
        cout<<"Enter your Dept Code"<<endl;
        cin >> deptcode;
}
void manager :: display( )
{
        cout<<"Name is "<<name;
        cout<<"\nAge is"<<age;
        cout<<"\nBasic and HRA "<<basic<<"\t"<<hra;
        cout<<"\nDept Code "<<deptcode;
}
void main( )
{
        clrscr();
        manager obj;
        obj.get1( );
        obj.get2( );
        obj.get3( );
        obj.display( );
        getch();
}
```


Output of the Program

```
Enter your age 20
Enter your name Girish
Enter your basic and HRA
200
150
Enter your Dept Code
1235
```

Name is Girish
Age is 20
Basic and HRA 200 150
Dept Code 1235

The class manager is inherited from the class emp which in turn inherited from class person .Thus the class emp and manager will possess the following features.

- Class emp

Public Functions	Protected Variables
void get1()(From Person Class) void get2()	name, age (From Person Class) basic,hra.

- Class manager

Public Functions	Protected Variables
void get1()(From Person Class) void get2() void get3() void display()	name, age (From Person Class) basic,hra(From emp class) deptcode(Private variable)

Have you Understood Questions?

1. Multilevel inheritance is special form of Multiple inheritance (TRUE/FALSE)

5.7Hierarchical Inheritance

The hierarchical inheritance structure is given below .This type is helpful when we have class hierarchies. In this scheme the base class will include all the features that are common to the subclasses.

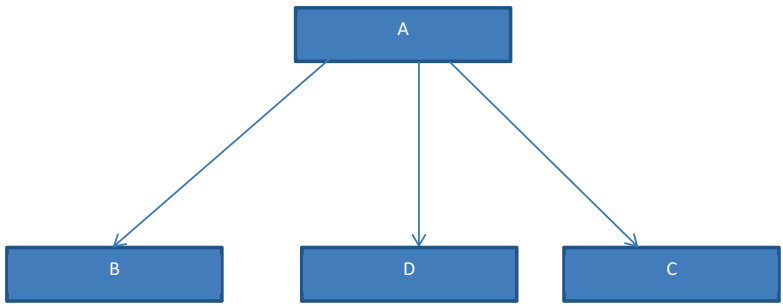


Fig 5.5: Hierarchical Inheritance

In hierarchical inheritance it is necessary to create object for all the derived classes at the lower level because each derived class will have its own unique feature.

Example 6:

A simple program is written to implement the hierarchical inheritance structure given in the above figure.

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class A
{
        protected:
                int x, y;
        public:
                void get ( )
                {
                        cout<<"Enter two values"<<endl;
                        cin>> x>>y;
                }
};
class B : public A
{
        private:
                int m;
        public:
                void add( )
                {
                        m= x + y;
                        cout<<"\nThe Sum is "<<m;
                }
};
class C : public A
{
        private:
                int n;
        public:
                void mul( )
                {
                        n= x * y;
                        cout << "\nThe Product is "<<n;
                }
};
class D : public A
{
```

```
        private:
                float l;
        public:
                void division( )
                {
                        l = x / y;
                        cout <<"\nThe Quotient is"<< l;
                }
};
void main( )
{
        clrscr();
        B obj1;
        C obj2;
        D obj3;
        cout<<"Addition:\n";
        obj1.get( );
        obj1.add( );
        cout<<"\nMultiplication:\n";
        obj2.get();
        obj2.mul( );
        cout<<"\nDivision:\n";
        obj3.get();
        obj3.division( );
        getch();
}
```


Output of the Program

```
Addition:
Enter two values:
10 20
The sum is:30
Enter two values
20 30
The Product is:600
Division:
Enter two values
40 50
The Quotient is:0
```

Here we can note that objects were created for all the subclasses to access their member functions.

Have you Understood Questions?

1. Hierarchical inheritance is useful when we have _____ of classes

5.8 Hybrid Inheritance

A hybrid inheritance scheme is the combination of all the inheritance schemes discussed earlier. Consider for a example , calculating net pay for an employee, apart from the normal pay he may also get additional pay by working overtime, thus the net pay will be the sum of normal pay plus overtime pay. This example can be modeled as hybrid inheritance scheme in C++.

Example 7:

The C++ classes for the fig 5.6 are given below.

```
#include<iostream.h>
#include<conio.h>
class emp_det
{
        protected:
                int emp_id;
        public:
                void get_id()
                {
                        cout<<"ENTER EMPID"<<endl;
                        cin>>emp_id;
                }
                void disp_id()
                {
                        cout<<" EMPID IS "<<emp_id<<endl;
                }
};
class norm_pay :public emp_det
{
        protected:
                float bpay;
        public:
                void get_bpay()
                {
                        cout<<"ENTER BASIC PAY"<<endl;
                        cin>>bpay;
                }
                void disp_bpay()
                {
                        cout<<" BASIC PAY IS = "<<bpay<<endl;
                }
};
class add_pay
{
        protected:
                int hrs;
```

```
                int rp;
                int ap;
        public:
                void get_addpay()
                {
                        cout<<"ENTER OVERTIME HOURS"<<endl;
                        cin>>hrs;
                        cout<<"ENTER HOW MUCH RUPEES PER
HOUR"<<endl;
                        cin>>rp;
                }
                void disp_addpay()
                {
                        ap=hrs * rp;
                        cout<<"TOTAL OVERTIME HOURS"<<hrs<<endl;
                        cout<<"ADDITIONAL PAY = "<< ap <<endl;
                }
};
class net_pay :public norm_pay,public add_pay
{
        int netpay;
        public:
                void display()
                {
                        netpay=ap+bpay;
                        cout<<"NET PAY IS "<<netpay;
                }
};
void main()
{
        clrscr();
        net_pay obj;
        obj.get_id();
        obj.get_bpay();
        obj.get_addpay();
        obj.disp_id();
        obj.disp_bpay();
        obj.disp_addpay();
        obj.display();
        getch();
}
```


Output of the Program

```
ENTER EMPID
1
```

```
ENTER BASIC PAY
200
ENTER OVERTIME HOURS
2
ENTER HOW MUCH RUPEES PER HOUR
123
EMP ID IS 1
BASIC PAY IS=200
TOTAL OVERTIME HOURS 2
ADDITIONAL PAY=246
NET PAY IS 446
```

Have you Understood Questions?

1. Hybrid inheritance scheme is the combination of
   (a) Multiple (b) Multilevel (c) both a and b

5.9Constructers and Destructors in Derived Classes

It is possible for the base class, the derived class, or both to have constructor and/or destructor functions. When a base class and a derived class both have constructor and destructor functions, the constructor functions are executed in order of derivation. The destructor functions are executed in reverse order. That is, the base constructor is executed before the constructor in the derived class. The reverse is true for destructor functions: the destructor in the derived class is executed before the base class destructor.

Example 8:
This example illustrates the order in which the constructors and destructors get executed in an inheritance program

```
#include<iostream.h>
#include<conio.h>
class A
{
        public:
                A ( )
                {
                        cout<<"INSIDE A";
                }
                ~A ( )
                {
                        cout<<"\nEXIT A";
                }
};
class B : public A
{
        public:
                B()
                {
                        cout<<"\nINSIDE B";
                }
                ~B()
                {
                        cout<<"\nEXIT B";
                }
};
class C : public B
{
        public:
                C( )
                {
                        cout<<"\nINSIDE C";
                }
                ~ C( )
                {
                        cout<<"\nEXIT C";
                }
};
void main()
{
     C obj;
        getch();
}
```


Output of the Program

Inside A
Inside B
Inside C
Exit C
Exit B
Exit A

5.9.1 Parameterized Constructers in Derived Classes

C++ supports a unique way of initializing class objects when a hierarchy of classes are created using inheritance. Constructers in an inheritance program can also be written using the syntax given below.

constructer (arglist): initialization-section
{
//body of the constructer
}

Note that we have used a colon ":" immediately after the constructer name to assign initial values. An example is given below.

Example 9:

The following is the example of parameterized constructors in an inheritance program


```
#include <iostream.h>
#include<conio.h>
class A
{
        int a;
        public:
                A (int x )
                {
                        a=x;
                        cout<<"INSIDE A";
                        cout<<" \na is "<<a<<endl;
                }
                ~A ( )
                {
                        cout<<"\nEXIT A";
                }
};
class B
{
        int b;
        public:
                B(int y) : b(y)
                {
                        cout<<"\nINSIDE B";
```

```
                cout<<"\n b is "<<b<<endl;
        }
        ~B()
        {
                cout<<"\nEXIT B";
        }
};
class C : public A,public B
{
        int m,n;
        public:
                C(int p,int q,int r,int s ) :A(p),B(q*2),m(r),n(s)
                {
                        cout<<"\nINSIDE C";
                        cout<<" \nm is "<<m<<" n is "<<n<<endl;
                }
                ~ C( )
                {
                        cout<<"\nEXIT C";
                }
};
void main()
{
        clrscr();
        int a,b,c,d;
        cout<<"Enter four values:";
        cin>>a>>b>>c>>d;
        C obj(a,b,c,d);
        getch();
}
```


Output of the Program

```
Enter four values:10 20 30 40
Inside A
a is 10
Inside B
b is 20
Inside C
M is 30 and n is 40
Exit C
Exit B
Exit A
```

In this program the object for the class C is created, in the constructer of class C we have initialized the values for the member variables m and n using the statements m(r) and n(s) which is given in the initialization section of the

constructer C. Similarly we have also initialized the values of the base class constructers. The statement A(p) will initialize the value of 'a' in the base class A and the statement B(q*2) will initialize the value of b to q multiplied by 2 in the base class B.

Have you Understood Questions?

1. Write the order in which constructers are executed in inheritance
2. Write the order in which destructors are executed in inheritance

5.10 Summary

- Inheritance is a feature by which new classes are derived from the existing classes.
- The parent class(Original existing class) is said to be the base class the new made class from it is called a derived class
- When a class is derived from a single base class it is said to be single inheritance. It can be thought of as a specialization of an existing class.
- A derived class can directly inherit more than one base class this concept is said o be multiple inheritance
- Multilevel inheritance is one form of multiple inheritance. In this scheme the derived class is made from two or more base class
- When the properties of one class are inherited by two or more classes then that inheritance scheme is said to be hierarchical inheritance.
- Hybrid inheritance scheme is the combination many simple inheritance schemes.
- A derived class can inherit a base class in public or protected or private mode.
- The protected access specifier is equivalent to the private specifier with the sole exception that protected members of a base class are accessible to members of any class derived from that base.
- It is possible for the base class, the derived class, or both to have constructor and/or destructor functions.
- When a base class and a derived class both have constructor and destructor functions, the constructor functions are executed in order of derivation. The destructor functions are executed in reverse order.

5.11 Exercises

Short Questions

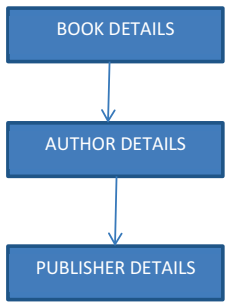
1. Mention the different scheme of inheritance supported by C++
2. Write the syntax of creating a derived class.
3. The derived class is the _____ of the generalized base class
4. All the public members of the base class will become _____ for the derived class if we inherit in protected mode
5. Write the syntax for creating multiple inheritance
6. _____ inheritance scheme is involved ,if a derived class D inherits properties of the base classes A and B.
7. List out some advantages of multilevel inheritance
8. What do you mean by hybrid inheritance?
9. Constructers are not inherited in derived classes (TRUE/FALSE)
10. Destructors in inheritance are called in _____ order of the inheritance

Long Questions

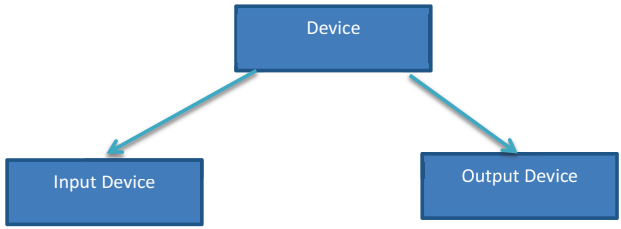
1. Discuss the various types of inheritance available in C++
2. Give a brief note on hybrid inheritance with an example
3. Explain Constructers and destructors in inheritance.
4. Explain how parameters are passed to constructors in inherited program.

Programming Exercises

1. Create a base class called vehicle and add relevant data members and functions to it. Create derived classes that directly inherit the base class called two wheeler and four wheeler add relevant data members and functions to them. Write a main program and create objects for the derived class and access the member functions
2. Implement the inheritance scheme given below.



3. Implement the inheritance scheme given below with a C++ program



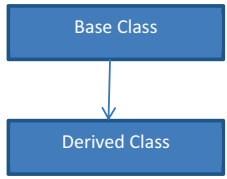
Answers to Have you Understood Questions

Section 5.3

1. Inheritance is a feature by which new classes are derived from the existing classes.
2. Single inheritance, Multiple inheritance, Multilevel inheritance, Hierarchical inheritance ,Hybrid inheritance

Section 5.4

1.



2. class derived classname : access mode base classname

Section 5.5

1. private
2. FALSE
3. Protected
4. private, protected and public

Section 5.6

1. If a derived class directly inherits from more than one base class.then this scheme of inheritance is said to be multiple inheritance.
2. C++,Java

Section 5.7

1. TRUE

Section 5.8

1. Hierarchy

Section 5.9

1. C

Section 5.10

1. The constructor functions are executed in order of derivation.
2. The destructors are executed from derived to base.

Chapter 6

Polymorphism

Structure of the Unit

- Binding
- Virtual Functions
- Virtual Destructors
- Virtual Base Classes
- Pure Virtual Functions
- Abstract Classes

Learning Objectives

- To introduce the concept of binding
- To present the concept of virtual functions
- To present the concept of virtual destructors
- To introduce pure virtual functions
- To discuss details regarding virtual base classes
- To introduce the concept of abstract classes

6.1Introduction

Polymorphism is another important feature of object oriented programming. The word "Poly" means many, and "Morphism" means structure or forms, thus in C++ polymorphism refers to an object that possess many form. Polymorphism can be classified into compile time polymorphism and runtime polymorphism. You have learned about compile time polymorphism in Chapter 2 and Chapter 4. Compile time polymorphism in C++ can achieved using function and operator overloading. In addition to compile time polymorphism (function and operator overloading) C++ also provides run time polymorphism with virtual functions and abstract classes. These can be used to generate common interfaces to different classes. In this chapter we will discuss details regarding run time polymorphism. This chapter covers all the fundamental aspects of run time polymorphism.

6.2 Binding

The process of associating or binding a particular function/method code to an object is called binding, i.e. binding specifies exactly which code is called, when a function is called. Usually, functions are called statically, i.e. the compiler knows at compile time exactly, which code is associated with an object. This is called static binding or early binding.

However, sometimes the compiler cannot know at compile time, which function is meant. In this case, the binding has to happen at run time. This is called dynamic binding or late binding.

Early binding achieves compile time polymorphism and late biding achieves run time polymorphism.

In C++ dynamic binding can be achieved by creating object pointers to the base class. When you create object pointers to base class we get the following advantages

- It makes the program flexible
- We can create one array to store objects of both base and derived classes

Pointers to base classes can be assigned addresses of derived classes. When such a pointer is used to access member methods, the compiler does not know, which method to call. It can be either the function from the base class or potentially an overloaded function in the derived class.

Example 1:

The following program creates a object pointer to the base class and access functions of both base and derived class.

```
#include<iostream.h>
#include<conio.h>
class base
{
        public:
                void display()
                {
                        cout<<"\nI AM IN BASE";
                }
};
class derived : public base
{
        public:
                void display()
                {
                        cout<<"\nI AM IN DERIVED";
                }
};
void main()
{
        clrscr();
        base b,*ptr;
        derived d;
        //To call display function in class base
        ptr= &b;
        ptr->display();
```

```
        //To call display function in class derived
        ptr=&d;
        ptr->display(); //This will invoke the display function of base not derived
        getch();
}
```

Output of the Program

I AM IN BASE
I AM IN BASE

To overcome the problem encountered earlier virtual functions are used.

6.2.1 Virtual Functions.

A virtual function is simply declared by putting the keyword virtual before the function declaration. Overloading virtual functions is called as overriding. The argument list and return-type of the function in derived classes must be the same as in the base class otherwise it is only overloaded, not overridden.

Any pointer to a class that contains virtual functions is provided with additional information about what it points to. When a virtual function is called through such a pointer, then the pointer knows what object it points to. This is dynamic binding.

Example 2:

The following program is the modified version of example 1, here the function display() is declared as virtual.

```
#include<iostream.h>
#include<conio.h>
class base
{
        public:
                virtual void display()
                {
                        cout<<"I AM IN BASE";
                }
};
class derived : public base
{
        public:
                void display()
                {
                        cout<<"\nI AM IN DERIVED";
                }
};
void main()
{
```

```
    clrscr();
    base b,*ptr;
    derived d;
    ptr= &b; //To call display function in class base
    ptr->display();//To call display function in class derived
    ptr=&d;
    ptr->display(); //This will invoke the display function of derived
    getch();
}
```

Output of the Program

I AM IN BASE
I AM IN DERIVED

A function declared virtual in base class is always virtual. The keyword virtual is not required in the derived class. Some important points about virtual functions:

- Overridden virtual functions must have the same argument list and return type.
- The different behavior of a virtual function and a non-virtual function occurs only, if the object is accessed through a pointer with type of a base class.
- Even if the virtual function is overridden, the derived class has still access to the virtual function. It can call it by using the scope-operator (::).
- The prototype of the virtual function in the base class and the derived class must be the same.
- A virtual function must be a member function of the class. It should not be a friend function.
- Virtual functions cannot be made static.
- Constructors cannot be made virtual; however destructors can be made virtual.
- A virtual function will be usually called with the help of a pointer.
- A virtual function in the base class must be declared even when no definitions can be made possible.

Each object with virtual function(s) is provided with a Function Lookup Table. This table contains information about the virtual functions and is used to determine, which function is to be called. However, there is only one table for each class and each object of this class gets a pointer to this table. If a virtual function is overridden, the pointer in the table points to the derived class function. If it is not overridden, it points to the base class version.

Have you Understood Questions?

1. Virtual functions achieves _____ form of polymorphism
2. When binding happens at run time we call it as _____
3. List the advantages of creating object pointers to base classes.

6.3Virtual Destructors

Constructers and destructors are not inherited. Constructors always create one particular type and hence it makes no sense to inherit them. Therefore they cannot be made virtual. Destructors are used when an object is deleted. If the derived class allocates some resources that outside the base class methods, it needs its own destructor. However, as explained before, a pointer to a base class can be used to point to a derived class. To ensure that the correct destructor is called, the destructor should be virtual otherwise it might happen that some resource is not deallocated. For this reason, destructors should always be declared virtual.

Have you Understood Questions?

1. Constrcuters are inherited(TRUE /FALSE)
2. Destructors are not inherited (TRUE/FALSE)

6.4Virtual Base Classes

Suppose you have two derived classes B and C that have a common base class A, and you also have another class D that inherits from B and C. You can declare the base class A as virtual to ensure that B and C share the same sub object of A.

In the following example, an object of class D has two distinct sub objects of class A, one through class B1 and another through class B2. You can use the keyword virtual in front of the base class specifies in the base lists of classes B1 and B2 to indicate that only one sub object of type A, shared by class B1 and class B2, exists.

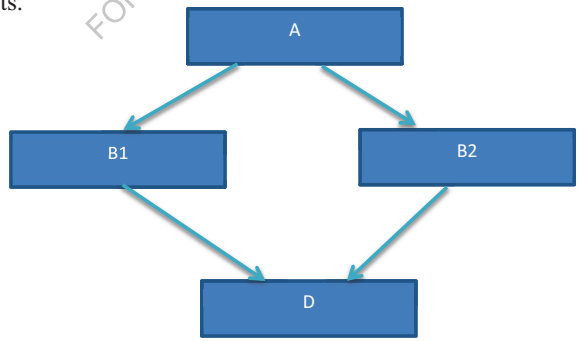


Fig. 6.1 Virtual Base Classes

The C++ classes for the fig 6.1 were

```
class A
{
        ----
        ----
};
class B1 :virtual public A
{
        ----
        ----
};
class B2: virtual public A
{
        ----
        ----
};
class D:public B1,public B2
{
        ----
        ----
};
```


Note:
The keywords virtual and public can be used in either order. For example virtual public A is equivalent to public virtual A.

Have you Understood Questions?

1. What happens if base class is declared virtual?
2. In which order the keywords virtual and public must appear in the derived class?

6.5Pure Virtual Function

Sometimes when a virtual function is declared in the base class there is no meaningful operation for it to perform. This situation is common because often a base class does not define a complete class by itself. Instead, it simply supplies a core set of member functions and variables to which the derived class supplies the remainder. When there is no meaningful action for a base class virtual function to perform, the implication is that any derived class must override this function. To ensure that this will occur, C++ supports pure virtual functions.

A pure virtual function has no definition relative to the base class. Only the function prototype is included. To make a pure virtual function, use this general form:

```
virtual type func-name(parameter-list) = 0;
```

For Example

virtual void show()=0;

The key part of this declaration is the setting of the function equal to 0. This tells the compiler that no body exists for this function relative to the base class. When a virtual function is made pure, it forces any derived class to override it. If a derived class does not, a compile-time error results. Thus, making a virtual function pure is a way to guaranty that a derived class will provide its own redefinition.

Have you Understood Questions?

1. How will you declare a pure virtual function?

6.6Abstract Classes

An abstract class is a class that is designed to be specifically used as a base class. For an abstract class no objects can be created. Objects can only be generated from derived classes. An abstract class contains atleast one pure virtual function. As mentioned earlier a pure virtual function is declared by adding "= 0" to the function declaration.

Any (non-virtual) class that is derived from an abstract class must implement all pure virtual functions of the base class. Otherwise it is a virtual class itself. The following is the example of an abstract class.

```
class shape
{
        public:
                virtual void area()=0;
};
```


Once the abstract class is created the class inheriting it should override all the pure virtual functions. For example

```
class square :public shape
{
        public:
                void area()
                {
                        cin>>side;
                        int s=2 * side;
                        cout<<"AREA IS "<<s;
                }
};
```

Have you Understood Questions?

1. Objects cannot be created for abstract class (TRUE/FALSE)

6.7Summary

- The process of associating or binding a particular function/method code to a name is called binding.
- Early binding achieves compile time polymorphism and late biding achieves run time polymorphism.
- Virtual functions are used to implement run time polymorphism.
- A virtual function is simply declared by putting the keyword virtual before the function declaration. Overloading virtual functions is called as overriding.
- Each object with virtual function(s) is provided with a Function Lookup Table. This table contains information about the virtual functions and is used to determine, which function is to be called.
- You can declare the base class as virtual to ensure that the subclasses share the same sub object of the base class
- A pure virtual function has no definition relative to the base class. Only the function prototype is included always a pure virtual function is equated to zero.
- An abstract class is a class that is designed to be specifically used as a base class.

6.8Exercises

Short Questions

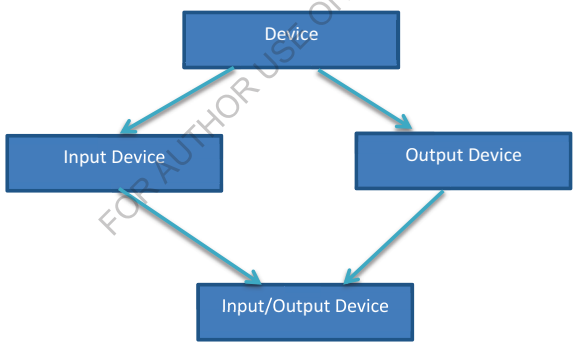
1. Differentiate early and late binding concepts
2. The binding that is associated with function overloading is _____
3. What is the purpose of function look up table ?
4. Write short notes on Virtual destructors
5. Why we need pure virtual functions? How it is declared?
6. Overloading of virtual functions is called as _____

Long Questions

1. Explain virtual functions with a example. Write the rules associated with virtual functions.
2. Write a not on Virtual Destructors
3. Explain Abstract classes and its features with example

Programming Exercises

1. Implement the inheritance scheme given below with a C++ program



2. Create a virtual base class called shape with a virtual function area (). Create derived classes rect, square and circle and override the function area() and compute the area of the objects.

Answers to Have you Understood Questions

Section 6.3

1. Run time
2. Dynamic binding
3. (a)It makes the program flexible
   (b) We can create one array to store objects of both base and derived classes

Section 6.4

1. False
2. True

Section 6.5

1. If we declare the base class as virtual to ensure that all the sub classes share the same object of the base class
2. They can appear in either order.

Section 6.6

1. By equating the function prototype to zero.

Section 6.7

1. TRUE.

Chapter 7

Files

Structure of the Unit

- File Stream Operations
- File I/O
- Sequential Input and Output Operations
- Random Access Files
- Input and Output Operations with Binary Files
- Reading and writing objects in a binary file
- Error Handling in File Operations

Learning Objectives

- To introduce file stream operations
- To introduce file I/O
- To discuss sequential input and output operations
- To present the concept of random access files
- To show how input output operations are performed on binary files
- To introduce error handling mechanisms in files

7.1Introduction

We all know C++ I/O operates on streams hence it is very important to know about streams supported by C++. Regarding streams this chapter starts the discussions from knowing the hierarchy of C++ I/O classes. You will be learning handling sequential files, random access files and binary files in detail. This chapter introduces you to write programs in C++ to handle the above mentioned files. Finally the chapter also discusses the error handling mechanisms supported by c++ when working with files.

7.2 File Stream Operations

C++ I/O system operates on streams. A stream is a common, logical interface to various devices of a computer system. A stream either produces or consumes information, and is linked to a physical device by the C++ I/O system. All streams behave in the same manner.

There are two types of streams: text and binary. A text stream is used with characters. When a text stream is being used, some character translations may take place (e.g. newline to carriage-return/linefeed combination). A binary

stream uses binary format for representing data in the memory. A binary stream can be used with any type of data. No character translation will occur. Thus always the binary file contains exact data sent by the stream.
C++ contains several predefined streams that are automatically opened. These are cin, cout, cerr and clog. By default, cin is linked to the keyboard, while the others are linked to the screen. The difference between cout, cerr and clog is, that cout is used for normal" output, cerr and clog are used for error messages.

The I/O system of C++ has many classes and file handling functions. All these classes are derived from the base class ios.The hierarchy of C++ I/O classes are given below.

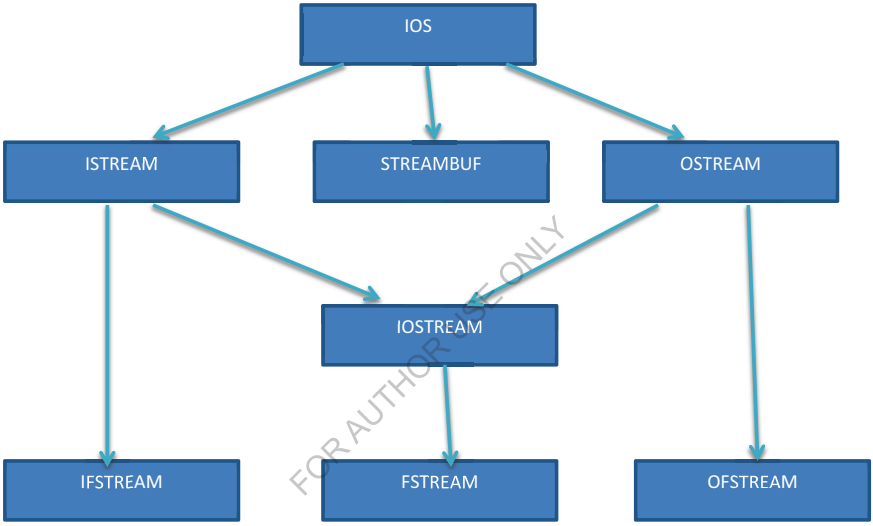


Fig. 7.1 Hierarchy of C++ I/O Classes

Have you Understood Questions?

1. What is a stream ?
2. What are the two types of streams?

7.3File I/O

File I/O and console I/O are closely related. To perform file I/O, you must include <fstream> in your program. It defines several classes, including ifstream, ofstream and fstream. These classes are derived from ios, so ifstream, ofstream and fstream have access to all operations defined by ios. In C++, a file is opened by linking it to a stream. There are three types of streams: input, output and input/output. Before you can open a file, you must first obtain a stream.

- To create an input stream, declare an object of type ifstream.
- To create an output stream, declare an object of type ofstream.
- To create an input/output stream, declare an object of type fstream.

Examples

ofstream out("emp.dat"); //Opens emp.dat file in output mode
ifstream in("emp.dat"); //Opens emp.dat file in input mode
fstream fin("emp.dat"); //Opens emp.dat file in input/output mode
The prototype of each member function is given below.

- void ifstream::open(const char * filename,openmode)
- void ifstream::open(const char * filename,openmode)
- void ifstream::open(const char * filename,openmode)

Here filename is the name of the file to be processed, the filename also includes the path specifier. The value of the mode specifies how the file has to be opened.C++ supports the following file opening modes

- ios::app
- ios::ate
- ios::binary
- ios::in
- ios::out
- ios::trunk

Note that the file opening modes can be combined using | symbol. For example
ofstream out("emp.dat" ,ios::out | ios::trunk)

- ios::app: causes all output to that file to be appended to the end. Only with files capable of output.
- ios::ate: causes a seek to the end of the file to occur when the file is opened.
- ios::out: specify that the file is capable of output.
- ios::in: specify that the file is capable of input.
- ios::binary: causes the file to be opened in binary mode. By default, all files are opened in text mode. In text mode, various character translations might take place, such as carriage return/linefeed sequences being converted into newlines. However, when a file is opened in binary mode, no such character translations will occur
- ios::trunc: causes the contents of a pre-existing file by the same name to be destroyed and the file to be truncated to zero length. When you create an output ios::trunc: causes the contents of a pre-existing file by the same name to be destroyed and the file to be truncated to zero length. When you create an output stream using ofstream, any pre-existing file is automatically truncated.

Once the file is opened we can read from or write into the file based on the mode you have opened. After processing the file it has to be closed. To close a file we have to use the member function called close, for example
in.close() //this will close file pointed by the stream in.
The close function will not take any parameters and will not return any value.

Example 1:

The following program writes name and rollno into a file called student.

```
#include<iostream.h>

#include<conio.h>

#include<fstream.h>

void main()

{

        clrscr();

        ofstream out("Student.txt"); //open the file student in output mode

        char name[30];

        int rollno;

        cout<<"ENTER YOUR NAME"<<endl;

        cin>>name;

        cout<<"ENTER YOUR ROLL NO"<<endl;

        cin>>rollno;

        out<<name <<"\t" ; // write name to the file student

        out<<rollno; // write rollno to the file student

        out.close();

        getch();

}
```


Output of the Program

```
ENTER YOUR NAME

Girish

ENTER YOUR ROLL NO

2020
```

To open file go to tc/bin folder and open student.txt file (Contents of File)

Girish 2020

In this example we have opened the file student in output mode. Since the openmode parameter to open( ) defaults to a value appropriate to the type of stream being opened, there is no need to specify its value in the preceding example.

Example 2:

The following example reads the content of the file student.

```
#include<iostream.h>

#include<conio.h>

#include<fstream.h>

void main()

{

    clrscr();

    ifstream in("Student.txt"); //open the file student in input mode

    char name[30];

    int rollno;

    in>>name; // read name from the file student

    in>>rollno; // read rollno from the file student

    cout<<"NAME IS "<<name<<endl;

    cout<<"ROLL NO IS "<<rollno ;

    in.close();

    getch();

}
```


Output of the Program

NAME IS Girish

ROLL NO IS 2020

In the preceding examples we are working with one file at a time. Now lets learn now let us learn how to work with multiple files.

Example 3:

This example works with more than one file.

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
        ofstream out;
        out.open("book.txt");
        out<<"C++";
        out<<"JAVA";
        out.close();
        out.open("author.txt");
        out<<"Girish";
        out<<"Rydhm";
        out.close();
        out.open("Publisher.txt");
        out<<"Lambert";
        out<<"TATA";
        out.close();
        ifstream in;
        char name[100],author[100],publisher[100];
        in.open("book.txt");
        cout<<"BOOK DETAILS "<<endl;
        in>>name;
        cout<<name;
        cout<<"\nAUTHOR DETAILS"<<endl;
        in.open("author.txt");
        in>>author;
        cout<<author;
        in.close();
        cout<<"PUBLISHER DETAILS"<<endl;
        in.open("publisher.txt");
        in>>publisher;
        cout<<publisher;
        in.close();
        getch();
}
```


Output of the Program

BOOK DETAILS
C++ JAVA
AUTHOR DETAILS
Girish Rydhm
PUBLISHER DETAILS
Lambert Tata

Here three files book, author, publisher are opened in output mode and the contents are written and then same files are opened in input mode, the contents are read from the file and displayed.

In this example we have used the getline function, which is used to read a line from the file pointed by the stream and store in a buffer .The second parameter in the getline function indicates buffer name.

Have you Understood Questions?

1. List some classes available in fstream header file?
2. if we want have a input and output operations on a file which type of object we create?
3. Write any two file opening modes?

7.4Sequential Input And Output Operations

Sequential input and output operations are performed on sequential files. A sequential file is one in which every record is accessed serially, in the order in which they are written hence to process any ith record all the previous i-1 records has to be processed.

In C++ every sequential file (including Random access file) will be associated with two file pointers. They are

- get() pointer
- put() pointer

The get pointer is an input pointer; it is used to read the content of the file. The get pointer will point content of the file to be read. When the file is opened in input mode the get pointer will point the first location of the file.

The put pointer is an output pointer; it is used to write content to the file. The put pointer will point to location where the content has to be written. When the file is opened in output mode the put pointer will point the first location of the file. However when the file is opened in append mode (ios::app) the put pointer will point the last location of the file.

Example 4:

This example program uses put pointer to write content to the sequential file.

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
        clrscr();
        char ch;
        fstream out("test.txt",ios::out); //opens “test” file in output mode
        cout<<"Enter a string at end press #"<<endl;
        do
```

```
        {
                cin.get(ch);
                out.put(ch); //write a character to the file
        }while(ch!= '#');
        out.close();
        getch();
}
```

Output of the Program

```
Enter a string at end press #
Girish#
Open file from tc\bin\test.txt
Girish#
```

Example 5:

This is an example program that uses get pointer to read content from the sequential file.

```
#include<conio.h>
#include<iostream.h>
#include<fstream.h>
void main()
{
        clrscr();
        fstream in ("test.txt",ios::in); //opens "test" file in input mode
        char ch;
        cout<<"READING CONTENT FROM THE FILE"<<endl;
        while (in)
        {
                in.get(ch); //get a character to the file
                cout<<ch;
        }
        in.close();
        getch();
}
```

Output of the Program

```
READING CONTENT FROM THE FILE
Girish#
```


Have you Understood Questions?

1. What is sequential file ?
2. write the 2 pointers associated with sequential file ?

7.5Random Access Files

Random access file is one that allows accessing records randomly in any order hence any ith record can be accessed directly. In order to perform random

access to a file C++ supports the following functions in addition to get() and put() pointers

- seekg() Moves get pointer to specified position
- seekp() Moves put pointer to specified position
- tellg() Returns the position of get pointer
- tellp() Returns the position of get pointer

The tell pointer is use to give the current position of the file. The functions tellg() and tellp() have the following prototype

- position tellg()
- position tellp()

Here position is the integer value that is capable of holding the largest value that defines the file position.
The functions seekg() and seekp() have the following prototype

- istream &seekg(offset,seekdirection);
- ostream &seekp(offset,seekdirection);

From the prototype we can note that both the functions have the same form. The first parameter offset specifies the number of bytes the file pointer has to move from the specified position. The second parameter seekdirection may take any one of the following values.

- ios::beg seek from the beginning
- ios::cur seek from the current position
- ios::end seek from end.

Consider the following examples

SEEK	POSITION OF THE FILE POINTER
in.seekg(0,ios::beg)	Moves the get pointer to the beginning of the file pointed by the stream in
in.seekg(0,ios::end)	Moves the get pointer to the end of the file pointed by the stream in
in.seekg(10,ios::cur)	Moves the get pointer 10 bytes ahead from the current position of the file pointed by stream in
in.seekg(-k,ios::end)	Moves the get pointer m bytes before from the end of the file pointed by stream in
out.seekp(k,ios::beg)	Moves the put pointer k bytes ahead from the beginning of the file pointed by stream out
out.seekp(-k,ios:;end)	Moves the put pointer k bytes before the end of the file pointed by stream out

Example 6:
The following program opens a random access file in input and output mode and it will allow accessing the specified character in the file.

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
	clrscr();
	fstream in("test.txt",ios::in | ios :: out); //opens “test” file in input/output mode
	char ch;
	int pos;
	cout<<"ENTER A STRING AT END PRESS #"<<endl;
	do
	{
		cin.get(ch);
		in.put(ch); //write a character to the file
	} while (ch!='#');
	in.seekg(0) ; // Goto the beginning of the file;
	cout<<"READING CONTENT FROM THE FILE"<<endl;
	while (in)
	{
		in.get(ch); //get a character to the file
		cout<<ch;
	}
	cout<<"ENTER THE POSITION OF THE FILE TO READ";
	cin>>pos;
	in.seekg(6,ios::beg); //MOVES THE GET POINTER TO THE POSITION
	while (in)
	{
		in.get(ch); //get a character to the file
		cout<<ch;
	}
	in.close();
	getch();
}
```

Output of the Program

```
ENTER THE STRING AT THE END PRESS #
Rydhm#
READING CONTENT FROM THE FILE
Rydhm#
```

ENTER THE POSITION OF THE FILE TO READ1
ydhm

Have you Understood Questions?

1. What is a random access fle?
2. Write the two seek pointers available in random access files.
3. what do you mean by out.seekp(k,ios::beg) ?

7.6Input and Output Operations with Binary Files

As mentioned earlier a binary file stores the content in binary format and a binary file can be used to represent any kind of data. Similar to the get and put functions of the text file we have read and write functions in a binary file. The syntax of read and write functions is given below.

istream &read(char *buf, streamsize num);
ostream &write(const char *buf, streamsize num);

The read( ) function reads num bytes from the stream and puts them in the buffer pointed to by buf. The write( ) function writes num bytes to the associated stream from the buffer pointed by buf. The streamsize type is some form of integer. An object of type streamsize is capable of holding the largest number of bytes that will be transferred in any I/O operation.

Example 7:

This program writes an integer number into a binary file.

```
#include<iostream.h>
#include<fstream.h>
#include<conio.h>
void main()
{
        clrscr();
        ofstream out;
        out.open("number",ios::binary); //opens a binary file
        int k=55;
        out.write((char *)&k,sizeof(k)); //writes integer to the file
        out.close();
        getch();
}
```

In this example a binary file stream is created by specifying ios::binary in the open statement. To write an integer to the file we have used write function, the first parameter to the function is the address of the variable k and the second is the length in bytes.

Note: The address of the variable must be casted to the type char*

Example 8:

This program reads an integer number from the binary file.

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

```
int main()
{
        clrscr();
        ifstream in;
        in.open("number",ios::binary); //opens a binary file
        int k;
        in.read((char *) &k,sizeof(k));
        cout<<k;
        in.close();
        getch();
        return 0;
}
```

Output of the Program

55

7.6.1 Reading and Writing Objects in a Binary File

We can write a object to a binary file as we do with the primitive data type like int, float etc. Similarly we can also read an object from the binary file. We will still use the read() and write() functions to perform read and write operations. The program given below illustrates reading and writing an object from the binary file.

Example 9:

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
class book
{
        char bname[40];
        float cost;
        char aname[40];
        int pubid;
        public:
                void getdetails();
                void printdetails();
};
void book :: getdetails()
{
        cout<<"ENTER BOOK NAME"<<endl;
        cin>>bname;
        cout<<"ENTER AUTHOR NAME"<<endl;
        cin>>aname;
        cout<<"ENTER COST OF THE BOOK"<<endl;
        cin>>cost;
        cout<<"ENTER PUBLISHER ID"<<endl;
        cin>>pubid;
```

```
}
void book :: printdetails()
{
        cout<<"BOOK NAME IS "<<bname<<endl;
        cout<<"AUTHOR NAME IS"<<aname<<endl;
        cout<<"COST OF THE BOOK IS "<<cost<<endl;
        cout<<"PUBLISHER ID IS "<<pubid<<endl;
}
int main()
{
        clrscr();
        fstream fin;
        fin.open("book",ios::in||ios::out);
        int i;
        book ptr[2];
        for(i=0;i<2;i++)
        {
                ptr[i].getdetails();
                fin.write((char *) &ptr[i],sizeof(ptr[i]));
        }
        fin.seekg(0);
        for(i=0;i<2;i++)
        {
                fin.read((char *) &ptr[i],sizeof(ptr[i]));
                ptr[i].printdetails();
        }
        fin.close();
        getch();
        return 0;
}
```


Output of the Program

```
Enter Book Name
C++
Enter Author Name
Girish
Enter Cost of the Book
500
Enter Publisher ID
123
Enter Book Name
Java
Enter Author Name
Rydhm
Enter Cost of the Book
```

```
560
Enter Publisher ID
124
BOOK NAME IS C++
AUTHOR NAME IS Girish
COST OF THE BOOK IS 500
PUBLISHER ID IS 123
BOOK NAME IS Java
AUTHOR NAME IS Rydhm
COST OF THE BOOK IS 560
PUBLISHER ID IS 124
```

This program creates an array of objects for the class book. The first for loop in the main program reads the details of 2 books. Note that we still use the write() function with the same syntax to write the object to the file. The second for loop reads the content from the file and displays the details.

Have you Understood Questions?

1. How will you perform read/write operations in binary files?

7.7ERROR HANDLING IN FILE OPERATIONS

When we are working with files a number of errors may occur. The most common errors that may occur while working with files are listed below

- Attempting to perform read or write operation on a file that does not exist.
- Attempting to process the file even after the last byte file of the file is reached.
- Attempting to write information to a file when opened in read mode.
- Attempting to store information in file, when there is no disk space for storing more data.
- Attempting to create a new file with a file name that already exits.

To overcome from these errors The C++ I/O system maintains status information about the outcome of each I/O operation. The current I/O status of an I/O stream is described in an object of type iostate, which is an enumeration defined by ios that includes the members.

NAME	MEMBERS
goodbit	No error occurred
failbit	A non fatal I/O error has occurred
eofbit	End of file has encountered

There are two ways in which you can obtain the I/O status information. First, we call the rdstate( ) function, which is a member of ios. It has this prototype:

```
iostate rdstate( );
```

It returns the current status of the error flags. rdstate( ) returns goodbit when no error has occurred. Otherwise, an error flag is returned. The other way you can determine whether an error has occurred is by using one of these ios member functions:

```
bool bad( );
bool eof( );
bool fail( );
bool good( );
```

The eof( ) function returns true if end of file is reached. The bad( ) function returns true if badbit is set. The fail( ) function returns true if failbit is set. The good( ) function returns true if there are no errors. Otherwise, they return false.

We can use all these functions in our file handling program to minimize the errors.

Example 10:

This program given below contains all error handling mechanisms.

```
#include<conio.h>
#include<fstream.h>
#include<iostream.h>
void main()
{
        clrscr();
        ifstream in;
        in.open("text");
        char ch;
        if(!in)
        {
                cout<<"CANNOT OPEN FILE"<<endl;
        }
        if(in.bad())
        {
                cout<<"FATAL ERROR IN FILE"<<endl;
        }
        cout<<"READING CONTENT FROM THE FILE"<<endl;
        {
                in.get(ch); //get a character to the file
                cout<<ch;
        }
        in.close();
        getch();}
```

This program initially checks whether the file pointed by the stream in exits, if so it will check whether there is any fatal error by using the faction bad().If there is no error the entire content of the file is read and displayed.

Have you Understood Questions?

1. Which function we use to detect end of file?
2. When will the fail bit set?

7.8Summary

- A stream is a common, logical interface to various devices of a computer system.
- A text stream is used with characters. A binary stream can be used with any type of data. No character translation will occur.
- File I/O in C++ has many classes that include ifstream, ofstream and fstream all these classes are derived from ios.
- There are many file opening modes that include input,output,appending etc.,
- A sequential file is one in which every record is accessed serially.
- In C++ every sequential file (including Random access file) will be associated with two file pointers namely get() and put().
- The get pointer is an input pointer; it is used to read the content of the file.
- The put pointer is an output pointer; it is used to write content to the file.
- Random access file is one that allows accessing records randomly in any order
- The tell function is use to give the current position of the file.
- The seek function is used to move the file pointer to the specified position.
- The read and write functions are used perform I/O operations in a binary file.
- The C++ I/O system maintains status information about the outcome of each I/O operation.
- The eof( ) function returns true if end of file is reached.
- The bad( ) function returns true if badbit is set.
- The fail( ) function returns true if failbit is set.
- The good( ) function returns true if there are no errors.

7.9Exercises

Short Questions

1. Draw the hierarchy of C++ IO streams
2. Mention the advantage of using binary files
3. List out the various file opening modes
4. ifstream in("emp.dat"); opens the file in _____ mode
5. _____ symbol is used to combine file opening modes
6. Mention the use of get() and put() pointers
7. Write the syntax of seekg and seep function.
8. Mention the members of the iostate object
9. To close a file we will call _____ function.
10. ios :: end is used to position the file pointer at the _____ of the file.

Long Questions

1. Explain random access files also explain how the file pointer is manipulated
2. Write short notes on reading and writing objects in binary files.
3. Explain in detail about error handling functions in files
4. Write short motes on sequential file organization

Programming Exercises

1. Write a program to create a file called "emp.dat" and write employee details into that file
2. Create a file called "number.txt" and write some numbers to the file. Create files called "odd.txt" and "even.txt" and write all the odd numbers to file "odd.txt" and even numbers to the file "even.txt".
3. Create a class called sales with your own data members and functions. Create object for the class and write the object to the binary file called "sales.dat".
4. Modify program 1 by including all error handling facilities.

Answers to Have you Understood Questions

Section 7.3

1. A stream is a common, logical interface to various devices of a computer system
2. (1) Binary stream (2) Text stream

Section 7.4

1. ifstream, ofstream and fstream.
2. To create an input/output stream we create an object of type fstream.
3. ios::in , ios::out, ios::trunk

Section 7.5

1. A sequential file is one in which every record is accessed serially
2. (1) get (2) put pointers

Section 7.6

1. Random access file is one that allows accessing records randomly in any order
2. seekg and seep
3. Moves the put pointer k bytes ahead from the beginning of the file pointed by stream out

Section 7.7

1. Using read() and write() functions

Section 7.8

1. By using eof() function.
2. When a non fatal I/O error has occurred

Chapter 8

Templates

Structure of the Unit

- Function Templates
- Function Templates with multiple parameters
- C++ function templates overloading
- Class Templates
- Class Templates with multiple parameters
- Member function

Learning Objectives

- To introduce the concept of function templates
- To show how function templates can be used with multiple parameters
- To discuss function template overloading
- To introduce the concept of class template
- To discuss class templates with multiple parameters
- To present the concept of member function templates

8.1Introduction

Generic programming is a approach where generic types are used as parameters. In C++ generic programming is done with the help of templates. Template is one of the important features available in C++. Using templates it is possible to create generic classes and generic functions hence templates provides support for generic programming.

C++ templates are those which can handle different data types without separate code for each of them. In generic functions or generic classes, the type of data that is operated upon is specified as a parameter. This allows us to use one function or class with several different types of data without having to explicitly recode a specific version for each type of data type. Thus, templates allow you to create reusable code.

8.2 Function Templates

In many situations, we want to write the same functions for different data types. For an example consider multiplication of two variables. The variable can be integer, float or double. The requirement will be to return the corresponding

return type based on the input type. Instead of writing an overloaded function to solve the above problem we can create a C++ function template. When we use C++ function templates, only one function signature needs to be created. The C++ compiler will automatically generate the required functions for handling the individual data types. Thus function templates make programming easy.

A function template or generic function defines a general set of operations that will be applied to various types of data. A function template has the type of data that it will operate upon passed to it as a parameter. Using this mechanism, the same general procedure can be applied to a wide range of data; hence templates are also called as parameterized functions or classes. A function template is created by using the keyword template. The general form of a template function definition is as

```
template <class T>
ret-type-name(parameter list)
{
        // body of function
}
```

Here the type T acts as a placeholder name for the data type used by the function. The compiler will automatically replace this placeholder with an actual data type when it creates a specific version of the function.

Example 1:

The following example declares add() function that will add two values of any given data type.


```
//FUNCTION TEMPLATES
#include<conio.h>
#include <iostream.h>
template <class T>
void add(T &a,T &b)
{
        T c;
        c= a+b;
        cout<<"THE SUM IS: "<<c<<endl;
}
void main()
{
        clrscr();
        int a=10,b=20;
        add(a,b);
        float c=12.5,d=13.8;
        add(c,d);
        getch();
}
```

Output of the Program

THE SUM IS:30
THE SUM IS:26.299999

In this example the keyword template is used to create template function add(). Here the variable T is used as a placeholder name for the data type .The function add() performs the addition of two numbers. In the main program the function add is called with two different set of data types i.e. int and float. Since add() is a template function the complier creates two version of add() function one to perform addition of integers and another to perform addition of two floats.

Example 2:

This example of function templates takes input parameters and return values.

```
//FUNCTION TEMPLATES WITH RETURN VALUES
#include<conio.h>
#include <iostream.h>
template <class T>
T small(T &a,T &b)
{
        if(a>b)
                return b;
        else
                return a;
}
int main()
{
        clrscr();
        int a,b;
        cout<<"ENTER TWO INTEGERS"<<endl;
        cin>>a>>b;
        cout<<"SMALLEST INTEGER IS "<<small(a,b);
        double m,n;
        cout<<"ENTER TWO REAL NUMBERS"<<endl;
        cin>>m>>n;
        cout<<"SMALLEST REAL NUMBER IS "<<small(m,n);
        getch();
        return 0;
}
```


Output of the Program

ENTER TWO INTEGERS
20 50
SMALLEST INTEGER IS 20 ENTER TWO REAL NUMBERS
15 12
SMALLEST REAL NUMBER IS 12

8.2.1 Function Templates with Multiple Parameters

A template function can take multiple parameters that may vary in their data types. The list of parameters that are passed to the template function are separated using commas. The general form of the template function with multiple parameters is given below.

```
template <class T1,class T2,….>
ret-type-name(parameter list of types T1,T2……)
{
// body of function
}
```

Example 3:

The following code snippet gives details regarding the function template with multiple parameters.

```
//FUNCTION TEMPLATES WITH MULTIPLE PARAMETERS
#include<iostream.h>
#include<conio.h>
template <class T1,class T2>
void power(T1 &a,T2 &b)
{
        T1 p=1;
        int i;
        for(i=1;i<=b;i++)
        {
                p=p*a;
        }
        cout<<a<<" RAISED TO THE POWER "<<b<<" IS = "<<p<<"\n";
}
void main()
{
        clrscr();
        power(3.5,5);
        power(5,5);
        getch();
}
```


Output of the Program

```
3.5RAISED TO THE POWER 5 IS=525.21875
5 RAISED TO THE POWER 5 IS=3125
```

In this example the generic function power() raise a base ‘a’ to the power ‘b’. Here we can note that this function takes two parameters. In the main program the first line power (3.5.5) will raise the float value 3.5 to the integral power 5 i.e. it computes 3.55. The second line power(5,5)will raise the integer value 5 to the power 5 i.e. it computes 55.Thus we can see that it is possible to have a template function with multiple parameters.

Example 4:

This example creates a generic function with multiple parameters that performs recursive binary search on a linear array.

```
//Generic functions to perform Binary Search
#include<conio.h>
#include<iostream.h>
template <class T1,class T2>
bsearch( T1 *a, T2 &e, int start, int end)
{
        int mid;
        if(start>end)
                return -1;
        mid=(start+end)/2;
        if(e==a[mid])
                return mid;
        else if(e<a[mid])
                return bsearch(a,e,start,mid-1);
        else
                return bsearch(a,e,mid+1,end);
}
void main()
{
        clrscr();
        int n,pos;
        int a[100],element;
        cout<<"Enter the array limit\n";
        cin>>n;
        cout<<"Enter the array elements";
        for(int i=0;i<n;i++)
                cin>>a[i];
        cout<<"Enter the element to be searched";
        cin>>element;
        pos=bsearch(a,element,0,n);
        if(pos==-1)
                cout<<"Element not found";
        else
                cout<<"Element is found in position"<<pos+1;
        getch();
}
```


Output of the Program

```
Enter the array limit
3
Enter the array elements 1 2 4
Enter the element to be searched 2
```

Element is found in position 2

8.2.2 C++ Function Templates Overloading

You can overload a template function as you do with the normal C++ function. If you call the name of an overloaded function template, the compiler will try to deduce its template arguments and check it's explicitly declared template arguments. If successful, it will instantiate a function template specialization. Errors will be generated if no match is found.

Example 5:

This example performs function templates overloading.

```
//TEMPLATE FUNCTION OVERLOADING
#include<conio.h>
#include<iostream.h>
template <class T1,class T2>
T1 add(T1 a,T2 b)
{
        T1 c;
        c=a+b;
        return c;
}
template <class T1,class T2,class T3>
T1 add(T1 &a,T2 &b,T3 &c)
{
        T1 d;
        d=a+b+c;
        return d;
}
void main()
{
        clrscr();
        int x;
        x=add(12,13);
        float y;
        y=add(12.5,10.2,15.8);
        cout<<"SUM OF TWO INTEGERS"<<x;
        cout<<"\nSUM OF THREE FLOATS"<<y;
        getch();
}
```


Output of the Program

```
SUM OF TWO INTEGERS 25
SUM OF THREE FLOATS 38.5
```

Two overloaded set of the template function add() exists in this program. If the add() function is invoked with two parameters then the template function having two parameters is invoked, if the add() function is called with three parameters then the template function having three parameters is invoked.

You can also overload a template function with a non template function. Always a non template function takes precedence over template functions. Consider the example given below.

Example 6:

This example performs overloading of template function with a non template function.

```
//TEMPLATE FUNCTION OVERLOADING
#include<iostream.h>
#include<conio.h>
template <class T>
void square(T &x)
{
        cout<<"TEMPLATE FUNCTION";
        cout<<"\nSQUARE VALUE IS "<<x * x;
}
void square(int x)
{
        cout<<"\nNON TEMPLATE FUNCTION";
        cout<<"\nSQUARE VALUE IS "<<x * x;
}
void main()
{
        clrscr();
        int a=10;
        square(a); //INVOKES NON TEMPLATE FUNCTION
        float b=12.5;
        square(b); //INVOKES TEMPLATE FUNCTION
        getch();
}
```


Output of the Program

```
NON TEMPLATE FUNCTION
SQUARE VALUE IS 100
TEMPLATE FUNCTION
SQUARE VALUE IS 156.25
```

Thus we can note that when the function square is called with an integer parameter it invokes the non template function whereas when the same function is invoked with a float parameter it calls the template function.

Have you understood?

1. What do you mean by Generic programming?
2. What do you mean by function templates?
3. Write the syntax for creating function templates.
4. How overloaded function template is called?

8.3Class Templates

C++ Class Templates are used where we have multiple copies of code for different data types with the same logic. For example we create a template for an queue class that would enable us to create queue to hold data items of different data types like int, float, double etc.,

Templates can be used to define generic classes. A class template definition looks like a regular class definition, except it is prefixed by the keyword template. A parameter should be included inside angular brackets. The parameter inside the angular brackets, can be either the keyword class or typename. This is followed by the class body declaration with the member data and member functions. The syntax to declare the class template is given below

```
//C++ class template
template <class T>
class classname
{
        //member variable declarations of type T and other types
        //member function declarations
};
```

Example 7:

The following is a simple example of class templates

```
//SIMPLE C++ CLASS TEMPLATE
#include<iostream.h>
#include<conio.h>
template <class T>
class calc
{
        T a,b;
        public:
                calc(T x,T y)
                {
                        a=x;
                        b=y;
                }
                void add()
                {
                        cout<<"SUM IS "<< a+b;
                }
                T mul()
                {
                        return a*b;
                }
};
void main()
{
```

```
        clrscr();
        calc<int> obj(10,10);
        obj.add();
        int c;
        c=obj.mul();
        cout<<"\nPRODUCT IS "<<c;
        getch();
}
```

Output of the Program

SUM IS 20
PRODUCT IS 100

This class template is designed to perform simple addition and multiplication for all data types. You can note that this template class resembles ordinary c++ class except the template declaration. Just look at the line that creates the object

calc<int> obj(10,10);

The int parameter within the angular brackets specifies that object obj will be working on integer parameters. You can note that the member functions are called in the same way as you do with the ordinary classes.

Example 8:

The program given below creates a generic stack and performs all basic stack operations.


```
//GENERIC STACK
#include<iostream.h>
#include<conio.h>
#include<iomanip.h>
#define SIZE 10
#include<stdlib.h>
template <class T>
class template_stack
{
        T stack[SIZE];
        int top;
        int status,data,choice;
        public:
        template_stack()
        {
                top=-1;
        }
        int push(int data)
        {
                if(isFull())
                        return -1;
                top++;
```

```
                stack[top]=data;
                return 1;
        }
        int pop()
        {
                int d;
                if(isEmpty())
                        return -1;
                d=stack[top];
                top--;
                return d;
        }
        int peek()
        {
                if(isEmpty())
                        return -1;
                return stack[top];
        }
        int isEmpty()
        {
                if(top==-1)
                        return 1;
                else
                        return 0;
        }
        int isFull()
        {
                if(top==SIZE-1)
                        return 1;
                else
                        return 0;
        }
        int size()
        {
                return top+1;
        }
};
void main()
{
        clrscr();
        template_stack<int> stk;
        int ch;
        int data;
        int status;
```

```
        do
        {
                cout<<"\n1->PUSH";
                cout<<"\n2->POP";
                cout<<"\n3->PEEK";
                cout<<"\n4->SIZE";
                cout<<"\n5->EXIT";
                cout<<"\nEnter Ur Choice\n";
                cin>>ch;
                switch(ch)
                {
                        case 1:
                                cout<<"\nEnter the data to insert";
                                cin>>data;
                                status=stk.push(data);
                                if(status==-1)
                                        cout<<"\nFull Stack"<<endl;
                                else
                                        cout<<data<<"\n Successfully Inserted";
                        break;
                        case 2:
                                status=stk.pop();
                                if(status==-1)
                                        cout<<"\nEmpty Stack"<<endl;
                                else
                                        cout<<"\n Data Popped is "<<status;
                        break;
                        case 3:
                                status=stk.peek();
                                if(status==-1)
                                        cout<<"\nEmpty Stack"<<endl;
                                else
                                        cout<<"\nThe Top Element in the Stack
is"<<status;
                        break;
                        case 4:
                                cout<<"\nThe Size of the Stack is "<<stk.size();
                        break;
                        case 5:
                                cout<<"\nBye";
                                exit(0);
                        default:
                                cout<<"\nInvalid Choice"<<endl;
                }
```

```
        }while(1);
}
```

Output of the Program

```
1->PUSH
2->POP
3->PEEK
4->SIZE
5->EXIT
Enter Ur Choice
1
Enter the data to insert 23
23 successfully inserted
1->PUSH
2->POP
3->PEEK
4->SIZE
5->EXIT
Enter Ur Choice
5
Bye
```

8.3.1 Class Templates with Multiple Parameters

Like function templates a class template can also take multiple parameters. Thus it is possible to use more than one generic data type in a class template. The list of parameters that are passed to the class template are separated using commas. The general form of the is given below

```
template <class T1,class T2,….>
class classname
{
        ……
        //body of the class
        ……
}
```

Example 9:

The following code is an example for class templates with multiple parameters

```
#include<conio.h>
#include<iostream.h>
#include<iomanip.h>
template <class T1,class T2>
class disp
{
        T1 x;
        T2 y;
        public:
                disp(T1 a,T2 b)
```

```
        {
                x=a;
                y=b;
        }
        void print()
        {
                for(int i=0;i<x;i++)
                cout<<y<<"\t";
        }


};
void main()
{
        clrscr();
        disp<int,char> obj(5,'a');
        obj.print();
        cout<<endl;
        disp<int,float> obj1(5,1.1);
        obj1.print();
        getch();
}
```

Output of the Program

a	a	a	a	a
1.1	1.1	1.1	1.1	1.1

8.3.2 Member Function Templates

So far in our class templates discussions we have used only inline functions in the class but it is still possible to write a member function definition outside the template class. The syntax is

```
template <class T>
returntype classname<T> :: functionname(parameter list)
{
//body of the function
}
```

Example 10:

The program given in the previous example is modified using member function templates.

```
#include<iostream.h>
#include<iomanip.h>
template <class T1,class T2>
class disp
```

```
{
        T1 x;
        T2 y;
        public:
                disp(T1 a, T2 b);
                void print();
};
template <class T1,class T2>
disp<T> :: disp(T1 a,T2 b)
{
        x=a;
        y=b;
}
template <class T1,class T2>
void disp<T> :: print()
{
        for(int i=0;i<x;i++)
                cout<<y<<endl;
}
void main()
{
        disp<int,char> obj(5,'a');
        obj.print();
        disp<int,float> obj(5,1.1);
        obj.print();
}
```


Output of the Program

a	a	a	a	a
1.1	1.1	1.1	1.1	1.1

Have you understood?

1. When you will create a class template?
2. Write the syntax for creating class templates.

8.4Summary

- Generic programming is an approach where generic types are used as parameters. In C++ generic programming is done with the help of templates.
- A function template or generic function defines a general set of operations that will be applied to various types of data.
- A template function can take multiple parameters that may vary in their data types.
- You can overload a template function as you do with the normal C++ function.
- C++ Class Templates are used where we have multiple copies of code for different data types with the same logic
- Templates can be used to define generic classes.
- Like function templates a class template can also take multiple parameters. Thus it is possible to use more than one generic data type in a class template.

8.5Exercises

Short Questions

1. How templates provide reusability of code?
2. How templates function differs from function overloading?
3. Function templates can return values (True/False)
4. Write the syntax for a template function taking multiple parameters
5. Explain Generic classes
6. Is it possible to use more than one generic data type in a class template?

Long Questions

1. Explain overloading template function with example.
2. Explain class templates with an example

Programming Exercises

1. Create a function template to perform bubble sort for generic type
2. Create a overloaded function template to compute area of different objects
3. Create a class template to implement Queue ADT

Answers to Have you Understood Questions

Section 8.3

1. Generic programming is an approach where generic types are used as parameters. In C++ generic programming is done with the help of templates.
2. A function template or generic function defines a general set of operations that will be applied to various types of data.
3. The syntax is

```
template <class T>
        ret-type-name(parameter list)
        {
                // body of function
        }
```

4. The function template is overloaded, the compiler will try to deduce its template arguments and check it's explicitly declared template arguments. If successful, it will instantiate a function template specialization. Errors will be generated if no match is found.

Section 8.4

1. C++ Class Templates are used where we have multiple copies of code for different data types with the same logic
2. The syntax is

```
template <class T>
class classname
{
    //member variable declarations of type T and other types
    //member function declarations
};
```

Chapter 9

Exception Handling

Structure of the Unit

- Exception Types
- Exception Handling Mechanism
- Functions Generating Exceptions
- Throwing Mechanism
- Catching Mechanism
- Multiple Catch Statements
- Catching All Exceptions
- Specifying Exceptions
- Rethrowing Exceptions

Learning Objectives

- To discuss different types of exceptions
- To introduce exception handling mechanisms in C++
- To discuss how to handle exceptions generated by a function
- To present throwing mechanism
- To present catching mechanism
- To discuss creating catches for a single try
- To show how to catch all exceptions
- To discuss about restricting exceptions
- To discuss about rethrowing exceptions

9.1 Introduction

Generally when we write program we may commit some errors. Errors cane be classified in two categories. They are Compile time errors and Run time errors. Compile time errors usually comprise of syntax errors such as a missing semicolon, unbalanced parenthesis etc., they can be easily detected when we compile our program but Runtime errors are relatively difficult to detect and rectify it. To solve this C++ provides a build-in error handling mechanism that is called exception handling. Using the C++ exception handling mechanism we can easily identify and respond to run time errors. This chapter briefly explores the various exception handling mechanisms available in C++.

9.2Exception Types

Exceptions can be classified into two categories. They are

- Synchronous Exception
- Asynchronous Exception.

Exceptions that are within the control of the program are called synchronous exceptions for example referring to an index of array, out of the range index. Exceptions that are beyond the control of program is said to be asynchronous exceptions for example memory overflow interrupt.

C++ exception handling mechanism works only for synchronous exception. To handle synchronous exceptions we have to follow the following steps

- Identify the problem(error)
- Report the error
- Receive the error
- Rectify the error.

The first step to handle exceptions is to identify the block of code that may cause error. After identifying it report the error to the error handling routine by "throwing" it. The error handling routine will receive and process the error.

Have you understood?

1. Give some examples for compile time errors.
2. What are the two categories of exceptions?
3. Mention the steps to handle synchronous exceptions.

9.3Exception Handling Mechanism

C++ exception handling mechanism is built upon three important keywords. They are

- try
- catch
- throw

The program statements that we suspect that they may cause runtime are put in the guarded section called "try" block. If the exception occurs in the try block it is thrown to the catch block. The catch block catches the thrown exception and handles it appropriately. The general form of the try catch structure is

```
try
{
        //try block
} catch(type1 arg)
{
        //catch block
}
```

When any statement present in the try block causes an exception (i.e error) the program control leaves the try block without executing further statements and switches to the catch block. The catch block will have the necessary program statements to handle the error.

In C++, exceptions are treated as objects. Hence when an exception is created and thrown the corresponding catch block (the catch that has the type which matches with the object) will catch the exception and handle it otherwise, the exception will be left unhandled and an abnormal program termination may occur.

The general form of a throw statement is

```
throw exception;
```

throw must be executed either from within the try block or from any function that the code within the block calls (directly or indirectly).

Example 1:

The following is an simple example with try catch block.

```
#include<iostream.h>
int main()
{
    int i,j;
    cout << "Starts here\n";
    i=10;
    j=0;
    try
    {
        // start a try block
        cout << "Inside try block\n";
        if(j==0)
        {
            throw j; // throw an error
            cout << "This will not be printed \n";
        }
        cout<<"RESULT IS "<<i/j;
    }
    catch( int a)
    { // catch an error
        cout << "The Number Caught is : ";
        cout << a << "\n";
    }
    cout << "Ends here";
    return 0;
}
```

Output of the Program

Starts Here
Inside try block
The Number caught is 0
Ends here

When the program enters the try block it is said to be in the guarded section. In this program when the value of j is zero an exception is created and thrown note that the statements after throw statement in try block is not executed. Once the exception is thrown the catch block catches the value (here zero) and handles it. After that the program continues it normal execution. Thus we can see that in C++ the exceptions are handled using the following steps.

- Control reaches the try statement by normal sequential execution. The guarded section within the try block is executed
- If no exception is thrown during execution of the guarded section, the catch clauses that follow the try block are not executed.
- If an exception is thrown during execution of the guarded section or in any routine the guarded section calls (either directly or indirectly), an exception object is created from the object created by the throw operand. The catch handler that matches the exception object is selected and it is executed.
- When there is no matching catch handler the exception will be left unhandled and an abnormal program termination may occur.

Have you understood?

1. Give the general form of try catch block.
2. When will the catch block gets executed?
3. Give the general form of throw statement

9.4Functions Generating Exceptions

C++ allows functions to generate exceptions. However these functions cannot be called as an ordinary function. To call a function generating exception, you have to enclose the function call with a try catch block.

Example 2:

The function compute() given in this example generates and throws exceptions.

```
#include<iostream.h>
#include<iomanip.h>
void compute(int a,int b)
{
    int c;
    if(b==0)
        throw b;
        c=a/b;
        cout<<"RESULT OF THE DIVISION"<<c;
}
```

```
void main()
{
    int x,y;
    cout<<"ENTER TWO NUMBERS"<<endl;
    cin>>x>y;
    try
    {
        compute(x/y)
    }
    catch(int k)
    {
        cout<<"DIVIDE BY ZERO EXCEPTION"<<endl;
    }
}
```

Output of the Program

```
ENTER TWO NUMBERS
5       0
DIVIDE BY ZERO EXCEPTION
```

Here we can note that the call to the function "compute()" is enclosed within the try catch block. If the value of the dividend "b" is zero the function generates an exception, which is handled in the catch block kept in the main program.

Have you understood?

1. How will you call a function that generates exception?

9.5 Throwing Mechanisms

An exception is thrown by using the throw keyword from inside the try block. Exception handlers are declared with the keyword catch, which must be placed immediately after the try block. A throw expression accepts one parameter, which is passed as an argument to the exception handler. The throw statement will have the following form

- throw (exception)
- throw exception
- throw

If an exception is thrown during execution of the guarded section or in any routine the guarded section calls (either directly or indirectly), an exception object is created from the object created by the throw operand. Now, the compiler looks for a catch clause that can handle an exception of the type thrown or a catch handler that can handle any type of exception. The catch handlers are examined in order of their appearance following the try block. If no appropriate handler is found, the next enclosing try block is examined. This process continues until the outermost enclosing try block is examined.

Have you understood?

1. Mention the different form of throw statement
2. How catch handlers are examined in a try block.

9.6Catching Mechanisms

The general form of the catch statement is

```
catch(type arg)
{
        //catch block
}
```

The catch statement used is determined by the type of the exception. That is, if the data type specified by a catch, matches that of the exception, that catch statement is executed (all other are bypassed). When an exception is caught, arg will receive its value. If you don't need access to the exception itself, specify only type in the catch clause (arg is optional). Any type of data can be caught, including classes that you create

Have you understood?

1. Give the general form of catch block.

9.7Multiple Catch Statements

A try can have multiple catches, if there are multiple catches for a try, only one of the matching catch is selected and that corresponding catch block is executed. The syntax for try with a multiple catch is given below.

```
try
{
        any statements
        if (some condition) throw value1;
        else if (some other condition) throw value2;
        ...
        ...
        ...
        else if (some last condition) throw valueN;
}catch (type1 name)
{
        any statements
}
catch (type2 name)
{
        any statements
}
```

```
...
...
...
catch (typeN name)
{
        any statements
}
```

Throw generates the exception specified by the corresponding value. This value can be of any type and multiple throw statements might occur. When an exception is thrown, it is caught by the corresponding catch statement, which then processes the exception.

If there are more than one catch statement associated with a try. The type of the exception determines which catch statement is used. When an exception is caught, name (the argument of the catch statement) will receive the thrown value. This thrown value determines which catch block will be executed. Sometimes the arguments of several catch statements match the type of an exception, in that case the first matching catch handler block is selected and executed. A simple example for try with multiple catches is given below.

Example 3:

The following example has a single try with multiple catches.

```
#include<iostream.h>
#include<iomanip.h>
void multiple_catch(int value)
{
        try
        {
                if (value==0) //throw an int value
                        throw1;
                else if (value==1) //throw a char
                        throw 'a';
                else //throw float
                        throw 1.1;
        }
        catch(char c)
        {
                cout<<"Character value is thrown" <<c;
        }
        catch(int i)
        {
                cout<<"Integer value is thrown"<<i;
        }
        catch(float f)
        {
                cout<<"Float value is thrown"<<f;
```

```
        }
}
void main()
{
        cout<<"Main Program"<<endl;
        multiple_catch(0);
        multiple_catch(1);
        multiple_catch(5);
}
```

Output of the Program

```
Main Program
Integer Value is thrown 1
Character Value is thrown a
Float value is thrown 1.1
```

Have you understood?

1. A try can many catch (True/False)
2. If a try has multiple catches, how many catch block gets executed.

9.8Catching All Exceptions

In some circumstances, it might be useful to catch all exceptions, instead of just a certain type. This situation may occur when we to write a program to handle all possible exceptions. To achieve this C++ supports a unique catch statement (catch with three dots) that can catch all exceptions. The syntax for the catch statement is


```
try
{
        //statements
}
catch (...)
{
        do something
}
```

Example 3:

The following code snippet contains the catch statement that can catch all exceptions.

```
#include<iostream.h>
#include<iomanip.h>
void multiple_catch(int value)
{
        try
        {
                if (value==0) //throw an int value
                        throw 1;
```

```
                else if (value==1) //throw a char
                        throw 'a';
                else //throw float
                        throw 1.1;
        }
        catch( …)
        {
                cout<<"Exception is Caught!!!" <<endl;
        }
}
void main()
{
        cout<<"Main Program"<<endl;
        multiple_catch(0);
        multiple_catch(1);
        multiple_catch(5);
}
```

Output of the Program

```
Main Program
Exception is Caught!!
Exception is Caught!!
Exception is Caught!!
```


Have you understood?

1. Write the syntax of the catch handler that can catch all exceptions

9.9 Specifying (Restricting) Exceptions

The type of exceptions that a function can throw can be restricted to certain types. To accomplish these restrictions, a throw clause must be added to the function definition. The throw clause specifies the possible list of types a function can throw. The syntax is given below.

```
ret.type func-name(arg-list) throw(type-list)
{
        //body of the function
}
```

Only the types in the comma separated type-list can be thrown by the function. Throwing any other type will cause abnormal program termination.

Example 4:

This example program restricts exceptions for the function multiple_catch().

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

```
#include<iomanip.h>
void multiple_catch(int value) throw (int,char,double)
{
        try
        {
                if (value==0) //throw an int value
                        throw1;
                else if (value==1) //throw a char
                        throw 'a';
                else //throw float
                        throw 1.1;
}
        catch(char c)
        {
                cout<<"Character value is thrown" <<c;
        }
        catch(int i)
        {
                cout<<"Integer value is thrown"<<i;
        }
        catch(float f)
        {
                cout<<"Float value is thrown"<<f;
        }
}
void main()
{
        cout<<"Main Program"<<endl;
        multiple_catch(0);
        multiple_catch(1);
        multiple_catch(5);
}
```


Output of the Program

Main Program
Integer value is thrown 1
Character value is thrown a
Float value is thrown 1.1

Have you understood?

1. How will you restrict functions that generates exceptions?

9.10 Rethrowing Exceptions

Sometimes an exception handler may not wish to process the exception instead it wishes to throw the exception. If an exception needs to be rethrown from within an exception handler, this can be done by calling throw statement without any parameter. The syntax is

```
try
{
        ….
        ….
}
catch (char c)
{
        throw; //rethrow same exception
}
```

Exceptions can be rethrown only within the catch block. When we rethrow an exception the same catch statement will not catch it again.

Have you understood?

1. Write the syntax of rethrowing exceptions.
2. Is it possible to rethrow an exception outside catch block ?

9.11 Summary

- Errors cane be classified in two categories they are Compile time errors and Run time errors.
- Exceptions can be classified as Synchronous and Asynchronous exceptions.
- C++ exception handling mechanism is built upon three important keywords. They are try, catch and throw.
- When any statement present in the try block causes an exception (i.e error) the program control leaves the try block without executing further statements and switches to the catch block.
- In C++, exceptions are treated as objects
- Throw statement must be executed either from within the try block or from any function that the code within the block calls (directly or indirectly).
- C++ allows functions to generate exceptions
- If an exception is thrown during execution of the guarded section or in any routine the guarded section calls (either directly or indirectly), an exception object is created from the object created by the throw operand.
- The catch handlers are examined in order of their appearance following the try block.
- The catch statement used is determined by the type of the exception
- A try can have multiple catches, if there are multiple catches for a try, only one of the matching catch is selected and that corresponding catch block is executed.
- C++ supports a unique catch statement (catch with three dots) that can catch all exceptions.
- The type of exceptions that a function can throw can be restricted to certain types. To accomplish these restrictions, a throw clause must be added to the function definition. The throw clause specifies the possible list of types a function can throw.
- If an exception needs to be rethrown from within an exception handler, this can be done by calling throw statement without any parameter.

9.12 Exercises

Short Questions

1. How does compile time errors differs form run time errors
2. Memory overflow is an example for _____
3. What do you mean by asynchronous exceptions
4. Write the three keywords used to handle exceptions
5. Explain guarded section
6. The number of try and catch statements in a c++ program should be the same (True/False)
7. How will you catch all exceptions?
8. Is it possible to restrict exceptions generated from a function?

Long Questions

1. Explain the exception handling statements in detail
2. Explain the different form of throw statement.
3. Discuss restricting exception generated by a function

Programming Exercises

1. Write a program to read array from a keyboard. Accept the index from the user and retrieve the element present in that index. Include exception handling facilities.
2. Create your own program to handle divide by zero exceptions

Answers to Have you Understood Questions

Section 9.3

1. Syntax errors such as a missing semicolon, unbalanced parenthesis etc.,
2. Exceptions can be classified as Synchronous and Asynchronous exceptions.
3. To handle synchronous exceptions the following steps are involved
   - Identify the problem(error)
   - Report the error
   - Receive the error
   - Rectify the error.

Section 9.4

1. The general form of the try catch structure is

```
try
{
        //try block
}
 catch(type1 arg)
{
        //catch block
}
```

2. When any statement present in the try block causes an exception (i.e error) the program control leaves the try block without executing further statements and switches to the catch block.
3. throw exception.

Section 9.5

1. By enclosing the function call in try catch block.

Section 9.6

1. The different form are (1) throw (exception) (2) throw exception (3) throw
2. The catch handlers are examined in order of their appearance following the try block. If no appropriate handler is found, the next enclosing try block is examined. This process continues until the outermost enclosing try block is examined.

Section 9.7

1. The general form of the catch statement is
   catch(type arg)

```
{

        //catch block
}
```

Section 9.8

1. True
2. one

Section 9.9

1.
```
try
{
        //statements
}
catch (...)
{
        do something
}
```

Section 9.10

1. The type of exceptions that a function can throw can be restricted to certain types. To accomplish these restrictions, a throw clause must be added to the function definition. The throw clause specifies the possible list of types a function can throw.

Section 9.11

1. The syntax is
```
try
{
        ….
        ….
}
catch (char c)
{
        throw; //rethrow same exception
}
```
2. No

Glossary

abstract class - a class that can only be used as a base class for some other class. A class is abstract if it has at least one pure virtual function.
access control - a C++ mechanism for prohibiting or granting access to individual members of a class.
access declaration - a way of controlling access to a specified member of a base class when it is used in a derived class.
access specifier - a way of labeling members of a class to specify what access is permitted.
Aggregate- an array or object of a class with no constructors, no private or protected members, no base classes, and no virtual functions.
allocation - the process of giving memory space to an object.
ANSI - acronym for American National Standards Institute, a standards body currently standardizing C++.
argument - when calling a function, refers to the actual values passed to the function.
argument matching - the process of determining which of a set of functions of a specified name matches given arguments in a function call.
ARM - acronym for the book The C++ Annotated Reference Manual, a C++ reference book by Ellis and Stroustrup.
array - an ordered and index-able sequence of values. C++ supports arrays of a single dimension (a vector) or of multiple dimensions.
asm - C++ keyword used to specify assembly language in the middle of C++ code.
assignment - the process of giving a value to a preexisting object.
assignment operator - an operator for doing assignment.
auto - a C++ keyword used to declare a stack-based local variable in a function. This is the default and is normally not needed.
base class - a class that serves as a base for a derived class to inherit members from.
bit field - a member of a class that represents small integral values.
bitwise copy - to copy an object without regard to its structure or members.
bool - C++ keyword used to declare a Boolean data type.
break - C++ keyword used to specify a statement that is used to break out of a for or while loop or out of a switch statement.

Glossary

browser - a software development tool used for viewing class declarations and the class hierarchy.
built-in type - see fundamental type.
C - a programming language in widespread use. C++ is based on C.
C-style string - refers to a char* and to the contents of any dynamic storage it may point at. C++ does not have true strings as part of the language proper, though a standard string class library is envisioned as part of the ANSI standardization effort.
call by reference - passing a pointer to an argument to a function. The function can then change the argument value.
call by value - passing a copy of an argument to a function. The function cannot then change the argument value. C and C++ use call by value argument passing.
calling conventions - refers to the system-specific details of just how the arguments to a function are passed. For example, the order in which they are passed on the stack or placed in machine registers.
case - a C++ keyword used to denote an individual element of a switch statement.
cast - a way of doing explicit type conversion via a cast operator.
catch - a C++ keyword used to declare an exception handler.
cerr - in C++ stream I/O, the standard error stream.
cfront - a C++ front end that translates C++ source code to C code, which is then compiled via a C compiler. Originally developed by AT&T Bell Labs in the mid-1980s.
char - a C++ keyword used to declare an object of character type. Often considered the same as a byte, though it is possible to have multi-byte characters.
cin - in C++ stream I/O, the standard input stream.
class - a C++ keyword used to declare the fundamental building block of C++ programs. A class has a tag, members, access control mechanisms, and so on.
class hierarchy - see base class, derived class.
class layout - the way in which data class members are arranged in a class object.
class library - a set of related classes declared in header files and defined in object files.

Glossary

class member - a constituent member of a class, such as a data declaration, a function, or a nested class.
class template - a template used for generating class types.
comments - C++ has C-style comments delimited with /* and */, and new C++-style line-oriented comments starting with //.
compilation unit - see translation unit.
compiler - a software tool that converts a language such as C++ into a different form, typically assembly language.
const - a C++ keyword used to declare an object as constant or used to declare a constant parameter.
constant - a literal or variable declared as const.
constant expression - a C++ expression that can be evaluated by the compiler. Used to declare bounds for an array among other things.
constructor - a function called when a class object comes into scope. The constructor is used to initialize the object.
const_cast - a C++ keyword used as a style of cast for explicitly casting away const.
container class - a type of class or template that is used to hold objects of other types. Lists and stacks would be examples of container classes.
continue - C++ keyword used with for and while statements to continue the iteration at the top of the loop.
conversion - to convert from one data type to another.
copy constructor - a special type of constructor that is called when an object is copied.
cout - in C++ stream I/O, the standard output stream.
data abstraction - the idea of defining a data representation (for example, to represent a calendar date), and a set of operations to manipulate that representation, with no public access to the representation except via the operations.
deallocation - the processing of freeing memory space previously used by an object.
debugger - a tool for stepping through the execution of a program, examining variables, setting breakpoints, and so on.
declaration - a C++ entity that introduces one or more names into a program.

Glossary

declaration statement - a declaration in the form of a statement that may be used in C++ where statements would normally be used.
declarator - a part of a declaration that actually declares an identifier name. A declarator appears after a sequence of type and storage class specifiers.
default argument - an optional argument to a function. A value specified in the function declaration is used if the argument is not given.
delete operator - C++ keyword and operator used to delete dynamic storage.
delete[] operator - See delete operator. Used to delete array objects.
demotion - converting a fundamental type to another fundamental type, with possible loss of precision. For example, a demotion would occur in converting a long to a char.
deprecate - to make obsolete (a language feature).
derived class - a class that inherits members from a base class. See inheritance.
destructor - a function called when a class object goes out of scope. It cleans up the object, freeing resources like dynamic storage. See constructor and deallocation.
dialect - refers to a variant of a programming language, used by a subset of the software community. Can also refer to a particular style of programming.
do - see while.
dominance - refers to the case where one name is used in preference to another.See multiple inheritance.
double - C++ keyword used to declare a floating point type.
dynamic storage - refers to memory allocated and deallocated during program execution using the new operator and delete operator.
dynamic_cast - a C++ keyword that specifies a style of cast used with run-time type information. Using dynamic_cast one can obtain a pointer to an object of a derived class given a pointer of a base class type. If the object pointed to is not of the specified derived class, dynamic cast will return 0.
else - C++ keyword, part of the if statement.
embedded system - a low-level software program that executes without much in the way of run-time services, such as those provided by an operating system.
encapsulation - a term meaning to wrap up or contain within. Used in relation to the members of a class. See access control.
enum - C++ keyword used to declare an enumeration.

Glossary

enumeration - a set of discrete named integral values. See enum.
enumerator - a member of an enumeration.
exception - a value of some type that is thrown. See exception handling.
exception handler - a piece of code that catches an exception. See catch and try block.
exception handling - the process of signalling that an exceptional condition (such as divide by zero) has occurred. An exception is thrown and then caught by an exception handler, after stack unwinding has occurred.
explicit - a C++ keyword used in the declaration of constructors to indicate that conversion of an initializer should not take place.
expression - a combination of constants, variables, and operators used to produce a value of some type.
expression statement - a statement that is an expression, such as a function call or assignment.
extern - a C++ keyword used to declare an external name.
external name - a name available to other translation units in a program. See linker and global variable.
false - C++ keyword used to specify a value for the bool type.
finalization - to declare that an object or resource is no longer needed, and initiate cleanup of that object. See initialization.
float - a C++ keyword used to declare a floating point type.
floating point - non-integral arithmetic. A floating-point number is typically represented as a base-two fraction part and an exponent.
for - a C++ keyword used to specify an iteration or looping statement.
forward class - a class for which only the tag has been declared. Such a class can be used where the size of the class is not needed, for example in pointer declarations.
free store - see dynamic storage.
friend - a type of declaration used within a class to grant other classes or functions access to that class. See access control.
front end - often refers to the early stages of C++ compilation, such as parsing and semantic analysis.
function - a C++ entity that is a sequence of statements. It has its own scope, accepts a set of argument values, and returns a value on completion.
function template - a template used for generating function types.

Glossary

fundamental type - a type built in to the C++ language. Examples would be integral types like int and pointer types such as void*.
garbage collection - a way of automatically managing dynamic storage such that explicit cleanup of storage is not required. C++ does not have garbage collection. See new operator and delete operator.
generic programming - see template.
global name - a name declared at global scope.
global namespace - the implicit namespace where global variables reside.
global scope - see global namespace.
global variable - a variable that is accessible throughout the whole program, whose lifetime is that of the program.
goto - C++ keyword, used to transfer control within a C++ function. See label.
grammar - a way of expressing the syntax of a programming language, to describe exactly what usage is valid and invalid.
header - see header file.
header file - a file containing class declarations, preprocessor directives, and so on, and included in a translation unit. It is expanded by the preprocessor.
heap storage - see dynamic storage.
helper class - a class defined as part of implementing the details of another class.
hiding - see encapsulation.
if - C++ keyword used in conditional statements.
implementation-dependent behavior - not every aspect of a programming language like C++ is specified in a language standard. This term refers to behavior that may vary from implementation to implementation.
implicit conversion - a conversion done as part of another operation, for example converting a pointer type to bool in an if statement.
inheritance - the process whereby a derived class inherits members from a base class. A derived class will also add its own members to those of the base class.
initialization - to give an initial value to an object. See constructor and assignment.
initialize - the process of initialization.
initializer - a value or expression used to initialize an object during initialication.
inline - C++ keyword used to declare an inline function.

Glossary

inline function - a function that can be expanded by a compiler at the point of call, thereby saving the overhead time required to call the function.
instantiation - see template instantiation.
int - a C++ keyword and fundamental type, used to declare an integral type.
integral conversion - the process by which an integer is converted to signed or unsigned.
integral promotion - the process by which a bool, char, short, enumerator, or bit field are converted to int for use in expressions, argument passing, and so on.
keyword - a reserved identifier in C++, used to denote data types, statements of the language, and so on.
label - a name that is the target of a goto statement.
layout - refers to the way that objects are arranged in memory.
library - a set of object files grouped together. A linker will search them repeatedly and use whatever object files are needed. See class library.
lifetime - refers to the duration of the existence of an object. Some objects last for the whole execution of a program, while other objects have a shorter lifetime.
linkage - refers to whether a name is visible only inside or also outside its translation unit.
linker - a program that combines object files and library code to produce an executable program.
literal - a constant like 1234.
local - typically refers to the scope and lifetime of names used in a function.
local class - a class declared local to a function.
local variable - a variable declared local to a function.
long - C++ keyword used to declare a long integer data type.
long double - a floating point type in C++.
lvalue - an expression referring to an object. See rvalue.
macro - a preprocessor feature that supports parameter substitution and expansion of commonly-used code sequences. See inline function.
mangling - see name mangling.
member - see class member and namespace member.
member function - a function that is an element of a class and that operates on objects of that class via the this pointer to the object.

Glossary

memberwise copy - to copy an object a member at a time, taking into account a copy constructor for the member. See bitwise copy.
method - see member function.
mixed-mode arithmetic - mixing of integral and floating point arithmetic.
module - see translation unit.
multiple inheritance - a derived class with multiple base classes. See inheritance.
mutable - C++ keyword declaring a member non-constant even if it is a member of a const object.
name - an identifier that denotes an object, function, a set of overloaded functions, a type, an enumerator, a member, a template, a namespace, or a label.
name lookup - refers to taking a name and determining what it refers to, or its value, based on the scope and other rules of C++.
name mangling - a way of encoding an external name representing a function so as to be able to distinguish the types of its parameters. See overload.
name space - a grouping of names.
namespace - a C++ keyword used to declare a namespace, which is a collection of names such as function declarations, classes, and so on.
namespace alias - an alias for a namespace, that can be used to refer to the namespace.
namespace member - an element of a namespace, such as a function, typedef, or class declaration.
nested class - a class declaration nested within another class.
new handler - a function established by calling set_new_handler. It is called when the new operator cannot obtain dynamic storage.
new operator - C++ keyword and operator used to allocate dynamic storage.
new-style cast - a cast written in functional notation.
new[] operator - see new operator. Used to allocate dynamic storage for array objects.
NULL - a special constant value that represents a null pointer.
null pointer - a pointer value that evaluates to zero.

Glossary

Glossary

object - has several meanings. In C++, often refers to an instance of a class. Also more loosely refers to any named declaration of a variable or other entity that involves storage.
object file - in C or C++, typically the output of a compiler. An object file consists of machine language plus an external name list that is resolved by a linker.
object layout - refers to the ordering of data members within a class.
object-oriented - this term has various definitions, usually including the notions of derived classes and virtual functions. See data abstraction.
old-style cast - a cast written in C style, with the type in parentheses before the value being casted.
OOA / OOD - acronym for object-oriented analysis and object-oriented design, processes of analyzing and designing object-oriented software.
OOP - acronym for object-oriented programming.
operator - a builtin operation of the C++ language, like addition, or an overloaded operator corresponding to a member function of a class. See function and operator overloading.
operator overloading - to treat a C++ operator like << as a function and overload it for particular parameter types.
overload - to specify more than one function of the same name, but with varying numbers and types of parameters. See argument matching.
overload resolution - see argument matching.
parameter - refers to the variables passed into a function. See also argument.
parameterized type - see template.
parser - see parsing.
parsing - the process by which a program written in some programming language is broken down into its syntactic elements.
placement - the ability to define a variant of the new operator to take an additional argument that specifies what storage is to be used.
pointer - an address of an object.
pointer to data member - a pointer that points at a data member of a class.
pointer to function - an address of a function or a member function.
pointer to member - see pointer to data member, pointer to function.
polymorphism - the ability to call a variety of member functions for a given class object using an identical interface in each case. See virtual function.

Glossary

postfix - refers to operators that appear after their operand. See prefix.
pragma - a preprocessor directive used to affect compiler behavior in an implementation-defined way.
prefix - refers to operators that appear before their operand. See postfix.
preprocessing - a stage of compilation processing that occurs before the compiler proper is invoked. Preprocessing handles macro expansion among other things. In C++ use of const and inline functions makes preprocessing
less important.
preprocessor - see preprocessing.
private - a C++ keyword used to specify that a class member can only be accessed from member functions and friends of the class. See access control, protected, and public.
programming environment - a set of integrated tools used in developing software, including a compiler, linker, debugger, and browser.
promotion - see integral promotion.
protected - a C++ keyword used to specify that a class member can only be accessed by member functions and friends of its own class and by member functions and friends of classes derived from this class. See private, public, and access control.
PT - see parameterized type.
public - a C++ keyword used to specify that class members are accessible from any (non-member) function. See access control, protected, and private.
pure virtual function - a virtual function with a "= 0" initializer. See abstract class.
qualification - to prefix a name with the name of a class or namespace.
recursive descent parser - see parsing. This is a type of parsing used in C++ compilers. It is more flexible than the older Yacc approach often used in C compilers.
reference - another name for an object. Access to an object via a reference is like manipulating the object itself. References are typically implemented as pointers in the underlying generated code.
register - C++ keyword used as a hint to the compiler that a particular local variable should be placed in a machine register.

Glossary

reinterpret_cast - a C++ keyword used as a style of cast for performing unsafe and implementation dependent casts.
repository - a location where an instantiated template class can be stored. See template instantiation.
resolution - see overload resolution.
resumption - a style of exception handling where program execution continues from the point where an exception is thrown. C++ uses the termination style.
return - C++ keyword used for returning values from a function.
return value - the value returned from a function.
RTTI - acronym for run-time type information.
run-time - refers to actions that occur during program execution.
run-time efficiency - refers to the issue of whether basic C++ operations will cause a performance penalty when the program is run.
run-time type information - a system for determining at run-time what the type of an object is.
rvalue - a value that may appear on the right-hand side of an assignment.
scope - the region of a program where a name has visibility.
semantic analysis - a stage that a compiler goes through after parsing. In this stage the meaning of the program is analyzed.
semantics - the meaning of a program, as opposed to its syntax.
separate compilation - refers to the process by which each translation unit of a program is compiled separately to produce an object file. The object files are then combined by a linker.
set_new_handler - a function used to establish a new handler.
short - a C++ fundamental type used to declare small integers.
signed - C++ keyword used to indicate a signed data type.
sizeof - C++ keyword for taking the size of an object or type.
smart pointer - an object that acts like a pointer but also does some processing whenever an object is accessed through them. The C++ operator -> can be overloaded to achieve this effect.
specialization - a special case of a template defined for particular template argument types.
stack frame - refers to a region of storage on the hardware stack, used to store information such as local variables for each invocation of a function.

Glossary

stack unwinding - see exception handling. When an exception is thrown, each active stack frame must be removed from the stack until an exception handler is found. This process involves calling a destructor as appropriate for each local object in the stack frame, and so on.
standard conversion - refers to standardized conversions between types, such as integral conversion.
standard library - see library. The C++ standard library includes much of the C standard library along with new features such as strings and container class support.
statement - the parts of a program that actually do the work.
static - see static member, static object, and static storage.
static member - a class member that is part of a class for purposes of access control but does not operate on particular object instances of the class.
static object - an object that is local to a function or to a translation unit and whose lifetime is the life of the program.
static storage - storage that persists throughout the life of the program. See static object and dynamic storage.
static type checking - refers to type checking that occurs during compilation of a program rather than at run-time.
static_cast - a C++ keyword specifying a style of cast meant to replace old-style
C casts.
storage class - see auto and static.
stream - an object used to represent an input or output channel. See stream I/O.
stream I/O - a C++ I/O library using overloaded operators << and >>. It has more type safety than C-style I/O.
string - see C-style string.
struct - a C++ class in which all the class members are by default public.
switch - C++ keyword denoting a statement type, used to dispatch to one of several sequences of statements based on the value of an expression.
symbol table - a compiler structure used to record type information about program names. The symbol table is used to generate compiler output.
syntax - the rules that govern how C++ expressions, statements, declarations, and programs are constructed. See grammar and semantics.

Glossary

Glossary

systems programming - refers to low-level programming, for example writing I/O drivers or operating systems. C and C++ are suitable languages for this type of programming.
tag - a name given to a class, struct, or union.
template - a parameterized type. A template can accept type parameters that are used to customize the resulting type.
template argument - an actual value or type given to a template to form a template class. See argument.
template class - a combination of a template with a template argument list via the process of template instantiation.
template declaration - a declaration of a template with its associated template parameter list.
template definition - an actual definition of a template or one of its members.
template instantiation - the process of combining template arguments with a template to form a template class.
template parameter - a value or type declared to be passed in to a template. See parameter.
temporary - an unnamed object used during the evaluation of an expression to store intermediate values.
termination - a style of exception handling where control does not return to the point where an exception is thrown. C++ uses this style of exception handling.
this - C++ keyword used in a member function to point at the object currently being operated on.
throw - C++ keyword used to throw (initiate) an exception. See exception handling.
translation limit - a limit on the size of a source program that a compiler will accept.
translation unit - a source file presented to a compiler with an object file produced as a result.
trigraph - a sequence of characters used to represent another character, for example to represent a character not normally found in the character set.
true - C++ keyword used to specify a value for the bool type.
try - C++ keyword used to delimit a try block.

Glossary

try block - a statement that sets up a context for exception handling. A subsequent throw from a function called from within the try block will be caught by the exception handler associated with the try block or by a handler further out in the chain of handlers.
type - a property of a name that determines how it can be used. For example, an object of a class type cannot be assigned to an integer variable.
type checking - see type system.
type conversion - converting a value from one type to another, for example via a constructor.
type safety - see type system.
type system - a system of types and operations on objects of those types. Type checking is done to ensure that the operations for given types are appropriate, for example that a function is called with arguments of the appropriate types.
type-safe linkage - refers to the process of encoding parameter type information in external names so that the linker will reject mismatches between the use and definition of functions. See name mangling.
typedef - a C++ keyword used to declare an alias for a type.
typeid - an operator that returns an object describing the type of the operand. See run-time type information.
union - a structure somewhat like a class or struct, except that individual union members share the same memory. See class layout.
unsigned - a C++ keyword used to declare an integral unsigned fundamental type.
unwinding - see stack unwinding.
user-defined conversion - a member function that supports conversion from an object of class type to any target type.
user-defined type - a class or typedef.
using declaration - a declaration making a class or namespace name available in another scope.
using directive - a way of making available to a program the members of a namespace.
using namespace - see using directive.
variable - an object that can be assigned to.

Glossary

vector - a one-dimensional array.
virtual base class - a base class where a single subobject of the base class is shared by every derived class that declared the base class as virtual.
virtual function - a member function whose interpretation when called depends on the type of the object for which it is called; a function for an object of a derived class will override a function of its base class.
virtual table - a lookup table used for dispatching virtual function calls. A class object for a class containing virtual functions will contain a pointer to a virtual table.
visibility - refers to the processing of doing name lookup without regard to whether a name is accessible. Once a name is found, then type checking and access control are applied.
void - a C++ keyword used to declare no type. It has special uses in C++, for example to declare that a function has no parameter list. See also void*.
void* - a pointer to a void type. Often used as the lowest common denominator type of pointer in C and C++.
volatile - a type qualifier used to indicate that an object may unpredictably change value (for example if it is mapped to a machine register) and thus should not have accesses to it optimized.
wchar_t - C++ keyword to declare a fundamental type used for handling wide characters.
while - C++ keyword used to declare an iteration statement.