C++ Programming

Basic Data Types:

Data types in C++ can be classified under various categories. Both C and C++ compilers support all the built-in (also known as basic-or fundamental) data types. With the exception of void, the basic data types may have several modifiers preceding them to serve the needs of various situations. The modifiers signed, unsigned, long, and short may be applied to character and integer basic data types. However, the modifier long may also be applied to double.

Table showing Size and range of C++ basic data types

Type	Bytes	Range
char	1	-128 to 127
unsigned char	1	0 to 255
signed char	1	-128 to 127
int	2	-32768 to 32767
unsigned int	2	0 to 65535
signed int	2	-32768 to 32767
short int	2	-32768 to 32767
unsigned short int	2	0 to 65535
signed short int	2	-32768 to 32767
long int	4	-2147483648 to 2147483647
signed long int	4	-2147483648 to 2147483647
unsigned long int	4	0 to 4294967295
float	4	3.4E -38 to 3.4E + 38
double	8	1.7E -308 to 1 .7E + 308
long double	10	3.4E -4932 to 1.1 E + 4932

User-Defined Data Types:

Structures and Classes: We have used user-defined data types such as struct and union in C. While these data types are legal in C++, some more features have been added to make them suitable for object-oriented programming. C++ also permits us to define another user-defined data type known as class, which can be used, just like any other basic data type, to declare variables. The class variables are known as objects.

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

Examples:

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

The enumerated types differ slightly in C++ when compared with those in ANSI C. In C++, the tag name shape, colour, and position become new type names. That means we can declare new variables using these tag names.

Examples:

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

ANSI C defines the types of enums to be int's. In C++, each enumerated data type retains its own separate type. This means that C++ does not permit an int value to be automatically converted to an enum value.

Examples:

colour background = blue; // allowed
colour background = 7; // Error in C++
colour background = (colour) 7; //OK

```
#include <iostream>
using namespace std;
enum week { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday };
int main()
{
        week today;
        today = Wednesday;
        cout << "Day " << today+1;
        return 0;
}
```

Derived Data Types:

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

char string[3] = "xyz";

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

char string[4] = "xyz"; II O.K. for C++

Pointers: Pointers are declared and initialized as in C.

Examples:

int * ip; // Int pointer
ip = &x; // address of x assigned to ip
***ip = 10; // 50 assigned to x through indirection**

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

Char *const ptr1="GOOD"; //constant pointer

We cannot modify the address that ptr1 is initialized to.

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

ptr2 is declared as pointer to a constant. It can point to any variable of correct type, but the contents of what it points to cannot be changed. We can also declare both the pointer and the variable as constants in the following way:

const char *const cp="xyz";

The statement declares cp as a constant pointer to the string, which has been declared as a constant. In this case, neither the address assigned to the pointer cp nor the contents it points to can be changed. Pointers are extensively used in C++ for memory management and achieving polymorphism.

Dynamic Initialization Of Variables:

One additional feature of C++ is that it permits initialisation of the variables at run time. This is referred to as dynamic initialisation. Remember that, in C, a variable must be initialised using a constant expression and the C compiler would fix the initialisation code at the time of compilation: However, in C++, a variable can be initialised at run time using expressions at the place of declaration. For example, the following are valid initialisation statements:

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

Manipulators:

Manipulators are operators used with the insertion operator << to modify -- or manipulate -- the way data is displayed. We'll look at two of the most common here: endl and setw.

The endl Manipulator: This is a manipulator that causes a linefeed to be inserted into the stream. It has the same effect as sending the single '/n' character.

The setw Manipulator: You can think of each value displayed by cout as occupying a field: an imaginary box with a certain width. The default field is just wide enough to hold the value. That is, the integer 567 will occupy a field three characters wide, and the string "pajamas" will occupy a field seven characters wide. However, in certain situations this may not lead to optimal results.

```
// demonstrates setw maniputator
#include <iostream.h>
#include <conio.h>
#include <iomanip.h>
void main()
{
        long pop1=2425785, pop2=47, pop3=9761;
        cout << setw(8) << "LOCATION" << setw(12) << "POPULATION" << endl
        << setw(8) << "Porttity" << setw(12) << pop1 << endl
        << setw(8) << "Hightown" << setw(12) << pop2 << endl
        << setw(8) << "Lowville" << setw(12) << pop3 << endl;
        getch();
}
```

The setw manipulator causes the number (or string) that follows it in the stream to be printed within a field n characters wide, where n is the argument to setw (n). The value is right justified within the field.

Type Conversion:

C++ like C, is more forgiving than some languages in the way it treats expression involving sever al different data types. As an example, consider the program below.

```
//shows mixed expression
#include <iostream.h>
#include <conio.h>
void main( )
{
        int count = 7;
        float avgWeight = 155.5;
        double totalWeight = count * avgWeight;
```

```
        cout<< "TotalWeight ="<< totalWeight << endl;
        getch();
}
```

Here a variable of type int is multiplied by a variable of type float to yield a result of type double. This program compiles without error, the compiler considers it normal that you want to multiply numbers of different types. Not all languages are this relaxed. Some don't permit mixed, expressions,and would flag as an error the line that performs the arithmetic in the above program.

Casts: the term applies to data conversions specified by the programmer, as opposed to the automatic data conversions.Sometimes a programmer needs to convert a value from one type to another in a situation where the compiler will not do it automatically, resulting in an erroneous result. In such case we might be able to solve the problem by using a cast. Here's an example:

```
// Program to illustrate the effect of casts
#include < iostream.h>
void main( )
{
        int intvar = 25000;
        intvar = (intvar * 10) / 10; // result too large
        cout << "intvar = " << intvar << endl; // wrong answer
        int intvar = 25000;
        intvar =((long)intvar) * 10 / 10; // cast to long
        cout << "intvar = " << intvar << endl; // right answer
}
```

When we multiply the variable intvar by 10, the result -250,000 is far too large to fit in a variable of type int, or unsigned int. This leads to the wrong answer, as shown in the first part of the program. We could redefine the data type of the variables to be long; this provides plenty of room, since this type holds numbers up to 2,14,74,83,647 . But suppose that for some reason, such as keeping the program small, we don't want to change the variables to type long. We can cast intvar to type long before multiplying. This is sometimes called coercion;the data is coerced into becoming another type. The expression

```
long(intVar)
```

casts intVar to type long. It generates a temporary variable of type long with the same value as intVar. It is this temporary variable that is multiplied by 10. Since it is type long, the result fits. This result is then divided by 10 and assigned to the normal int variable intVar. Here's the program's output:

```
intVar = -1214
intVar = 25000
```

The first answer, without the cast, is wrong; but in the second answer, the cast produces the correct result. You can use another syntax for casts. You can say

```
(long)intvar
```

with the parentheses around the type rather than around the variable. This is the only syntax acceptable in C, but in C++ the first approach (called "functional notation") is preferred, because it is similar to the way to functions.

Operators:

Arithmetic operators: C++ uses operators to do arithmetic. It provides operators for five basic arithmetic calculations: addition, subtraction, multiplication, division and taking the modulus. Each of these operators

uses two values (called operands) to calculate a final answer. Together, the operator and its operands constitute an expression. For example, consider the following statement:

int wheels = 4 + 2;

The values 4 & 2 are operands, the + symbol is the addition operator, and 4 + 2 is an expression whose value is 6. Some of the basic arithmetic operators are +, -, *, / and %.

Logical Operators: These operators allow you to logically combine Boolean (true/ false) values. For example, today is a weekday has a Boolean value, since it's either true or false.

Logical operators are:

Logical AND -&&
Logical OR - ||
Logical NOT - !

Relational Operators: A relational operator compares two values. The values can be any built-in C+ + data type, such as char, int or float or they can be user-defined class.

Relational operators are:

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

Decision Statements:

Programs also need to make these one-time decisions. In a program a decision causes a one-time jump to a different part of the program, depending on the value of an expression. The most important is with **if … else** statement, which chooses between two alternatives. This statement can be used without the else, as a simple **if** statement. Another decision statement, switch, creates branches for multiple alternative sections of code, depending on the value of a single variable. Finally the conditional operator is used in specialized situations.
if construct: The **if** statement is the simplest of the decision statements. The programs below illustrate this.

```
#include<iostream.h>
#include<conio.h>
void main ( )
{
        > int x;
        cout<< "Enter a number";
        cin>> x;
        if(x >100)
                cout<<"The number is greater than 100 \n";
        getch();
}
```

The **if** keyword is followed by a test expression in parentheses. The statements following the if are executed only once if the test expression is true.
if...else construct: The if statement lets you do something if a condition is true. If it isn't true, nothing happens. But suppose we want to do one thing if a condition is true, and do something else if it's false. That's where the **if ...else** statement comes in. It consists of an if statement, followed by a statement or block of statements, followed by the keyword else, followed by another statement or block of statements.

```
if( condition)
        statement; // if true
else
        statement; // if false
```

if the test expression in the **if** statement is true, the program prints one message; if it isn't, it prints the other.

else...if construct: The nested **if …else** statements can look clumsy and can be hard to interpret, especially if they are nested more deeply. However there's another approach to writing the same statements.

```
if( first condition)
        statement; // if true
else if( second condition )
        statement; // if false
else
        statement; // default
```

The compiler sees this as identical to **if… else,** but rearranged the if's so that they directly follow the else's. The program goes down the ladder of else … if's until one of the test expressions is true. It then executes the following statement and exits from the ladder.

switch construct: If you have a large decision tree, and all the decisions depend on the value of the same variable, you can probably consider a switch statement instead of a series of if ...else or else if constructions. Here's a simple example.

```
// program to demonstrates SWITCH statement
        #include <iostream.h>
        #include<conio.h>
        void main( )
        {
                int speed;
                cout << "\nEnter 33,45, or 78: "; cin >> speed;
                switch(speed) {
                        case 33:
                        cout << "Green\n"; break;
                        case 45:
                        cout << "Yellow\n"; break;
                        case 78:
                        cout << "Red\n"; break;
                }
               getch();
        }
```

This program prints one of three possible messages depending on whether the user inputs the number 33,45 or 78 The keyword switch is followed by a switch variable in parentheses

```
switch(speed)
```

Braces then delimit a number of case statements. Each case keyword is followed by a constant, which is not in parentheses but is followed by a colon

```
case 33:
```

The data type of the case constants should match that of the switch variable. Before entering the switch, the program should assign a value to the switch variable. This value will usually match a constant in one of the case statements. When this is the case, the statements immediately following the keyword case will be executed, until a break is reached.

break construct: The break keyword causes the entire switch statement to exit. Control goes to the first statement following the end of the switch.

The Conditional operator: This operator consists two symbols, which operate on three operands. It's the only such operator in C+ +; other operators operate on two or one operands. For example:

```
X = (a>b) ? a : b
```

The part of this statement to the right of the equals sign is called conditional expression. The question mark and the colon make up the conditional operator. The expression before the question mark is the test expression. If the test expression is true, then the entire conditional expression takes on the value of the operand following the question mark. If it is false, the conditional expression takes on the value of the operand following the colon.

Iteration statements:

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

```
do {
        statement 1;
        statement N;

} while(condition is true);
```

while statement: This is also a loop structure, but it an entry - controlled one.

```
while( condition is true)
{
statement 1;
statement N;
}
```

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

```
for( initial value; test condition; update expression) {
        statement 1;
        statement N;
}
```

Structures:

A structure is a collection of simple variables. The variables in a structure can be of different types; some can be int,some can be float, and so on. The data items in a structure are called the members of the structure.

Structures are one of the two important building blocks in the understanding of objects and classes. In fact,the syntax of a structure is almost identical to that of a class. A structure can be called as a collection of data, while class is a collection of both data and functions. So by learning about structures we'll be paving the way for an understanding of classes and objects. Structure is user-defined type, with a structure declaration serving to define the type's data properties. After you define the type, you can create variables of that type. Thus, creating a structure is a two - part process. First, you define a structure description. It

describes and labels the different types of data that can be stored in a structure. Then, you can create structure variables, or, more generally, structure data objects, that follows the description's plan.

Here's the structure description:

```
struct // Structure name
{
        structure element 1;
        structure element 2;
        … … …
        structure element N;
};
```

The keyword **struct** indicates that the code defines the layout for a structure. The **structuretag** is the identifier name of the structure.
For example:

```
struct data
{       char name[20];
        int age;
        char address[40];
};
```

for the above example; data is the name for the new type. Thus, you now can create variables of type data.Next, between braces, comes the list of data types to be held in the structure. Each list item is a declaration statement. This example has char, int. each individual item in the list is called a structure member.

```
// Program on a simple structure
# include <iostream.h>
#include<conio.h>
struct product {
        int pnumber;
        char type;
        float price;
};

void main( )
{
        product p1;
        p1.pnumber = 1234;
        p1.type = 'A';
        p1.price = 214.50;

        cout <<"\n Product number "<< p1.pnumber;
        cout <<"\n Product type "<< p1.type;
        cout <<"\n Product price "<< p1.price;
        getch();
}
```

Defining a Structure Variable:

The statement **product p1;** defines a variable, called p1, of type structure product. This definition reserves space in memory for p1 will hold enough space depending upon the number of elements in the structure. In some ways we can think of the product structure as the specification for a new data type. The format for defining the structure variable is the same as that for defining a built - in data type. This similarity is not accidental. One of the aims of C + + is to make the syntax and the operation of user-defined data types as similar as possible to that of built - in data types.

Accessing Structure Members:

Once a structure variable has been defined, its members can be accessed using some - thing called the dot operator. Here's how the first member is given value:

```
p1.pnumber = 1234;
```

The structure member is written in three parts: the name of the structure variable (p1); the dot operator, which consists of a period (.); and the member name (pnumber). This means the pnumber is member of p1. Remember that the first component of an expression involving the dot operator is the name of the specific structure variable (p1 in this case), not the name of the structure specifier (product). The variable name must be used to distinguish one variable from another when there is more than one, such as p2, p3, etc,. Structures members are treated just like other variables. In the statement p1.pnumber = 1234; the member is given the value 1234 using a normal assignment operator.

Function Overloading:

Function overloading allows you to create different functions that have the same name provided that they have different argument lists. Function overloading enables you to use the same name for related functions that performs the same basic task in different ways or for different types.It would be far more convenient to use the same name for all three functions, even though each value has different arguments. Here's a program, to illustrate overloading.

```
//demonstrates function overloading
#include< iostream.h>
#include< conio.h>
void repchar( );
void repchar(char);
void repchar(char,int);
void main( )
{
repchar( );
repchar('=');
repchar('*',40);
getch();
}

void repchar( )
{
for(int i=0; i<=40; i++)
cout<<"+";
}

void repchar(char c)
{
for(int i=0; i <=40; i++)
cout << c;
}

void repchar(char c, int j)
{
for(int i=0; i<=j; i++)
cout << c;
}
```

The program contains three functions with the same name. There are three declarations,three function calls,and three function definitions. What keeps the compiler from becoming hopelessly confused? It uses

the number of arguments, and their data types,to distinguish one function from another. In other words, the declaration void repchar( );

which takes no arguments, describes an entirely different function than does the declaration

void repchar(char);

which takes one argument of type c h a r, or the declaration

void repchar(char, int);

which takes one argument of type c h a r and another of type i n t . The compiler, seeing several functions with the same name but different numbers of arguments, it sets up a separate function for every such definition. Which one of these functions will be called depends on the number of arguments supplied in the call.

Object oriented concepts

Object oriented programming (OOP) is a particular conceptual approach to designing programs, and C++ has enhanced C with features that ease the way to applying that approach. The most important OOP features are these:

- Abstraction
- Encapsulation and data hiding
- Polymorphism
- Inheritance
- Reusable code

The class is the single most important C++ enhancement for implementing these features and tying together.

Procedural and Object-Oriented Programming:

Although we have explored the OOP perspective on programming ever so often, we've usually stuck pretty close to the standard procedural approach of languages such as C, Pascal, and BASIC. Let's look at an example that clarifies how the OOP outlook differs from that of procedural programming. You first concentrate on the procedures you will follow and then think about how to represent the data. You begin by thinking about the data. Furthermore, you think about the data not only in terms of how to represent it, but also in terms of how it's to be used. You concentrate on the object, as the user perceives it, thinking about the need to describe the object the object and the operations that will describe the user's interaction with the data. After you develop a description of that interface, you move on to decide how to implement the interface and data storage. Finally, you put together a program to use your new design.

Abstraction: Abstraction is the crucial step of representing information in terms of its interface with the user. That is, you abstract the essential operational features of a problem and express a solution in those terms. The interface describes how the user initializes, updates, and displays the data. From abstraction, it is a short step to the user-defined type, which in C++ is a class design that implements that interface.

Class: The class is the C++ vehicle for translating an abstraction to a user-defined type. It combines data representation and methods for manipulating that data into one neat package. An object has the same relationship to a class that a variable has to a data type. An object is said to be an instance of a class, in the same way my 1954 Chevrolet is an instance of a vehicle. A class is a way to bind the data and its associated functions together. It allows the data (and functions)to be hidden, if necessary, from external use. When defining a class we are creating a new abstract data type that can be treated like any other built-in data type.

Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members.The class function definitions describe how the class functions are implemented. The general form of a class declaration is:

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

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

The members that have been declared as private can be accessed only from within the class. On the other hand, public members can be accessed from outside the class also. The data hiding is the key feature of OOP. The use of the keyword private is optional. By default, the members are private. Such a class is completely hidden from the outside world and does not serve any purpose.

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

A Simple Class Example: A typical class declaration would look like:

```
class item
{

    int number; // variables declaration
    float cost; // private by default
    public:
    void getdata(int a, float b); // functions declaration
    void putdata(void); // using prototype
}
```

We usually give a class some meaningful name, such as item. This name now becomes a new type identifier that can be used to declare instances of that class type. The class item contains two data members and two function members. The data members are private by default while both the functions are public by declaration. The function getdata( ) can be used to assign values to the member variables number and cost, and putdata( ) for displaying their values. These functions provide the only access to the data members from outside the class. This means that the data cannot be accessed by any function that is not a member of the class item. Note that the functions are declared, not defined. Actual function definitions will appear later in the program. The data members are usual private and the member function

Creating Objects:

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

item x; // memory for x is created

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

item x, y, z;

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

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

```
class item
{
-----
-----
-----
} x. y, z;
```

would create the objects x, y and z of type item. This practice is seldom followed because we would like to declare the objects close to the place where they are used.

Accessing Class Members:

The private data of a class can be accessed only through the member functions of that class. The main( ) cannot contain statements that access number and cost directly. The following is the format for calling a member function:

object -name. function-name(actual-arguments) ;

For example, the function call statement

x.getdata(100,75.5);

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

x.putdata( );

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

getdata(100,75.5);

has no meaning. Similarly, the statement

x.number = 100;

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

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

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

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

The objects can access a variable declared as public directly. Example:

```
class xyz
{
        int x;
        int y;
        public
        int z;
        };
----
----
xyz p;
p.x = 0; // error, x is private
p.z = 10 // OK, z is public
----
----
```

Note that the use of data in this manner defeats the very idea of data hiding and therefore should be avoided.

Defining Member Functions:

Member functions can be defined in two places:

- Outside-the class definition
- Inside the class definition

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

Outside the Class Definition: Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body.

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

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

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

For instance, consider the member functions getdata( )and putdata( ) as discussed in the previous section. They may be coded as follows:

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

void item :: putdata(void)
{
        cout << "Number :" << number << "\n";
        cout << "cost ." << cost << "\n";
}
```

Since these functions do not return any value, their return::type is void.

The member functions have some special characteristics that are often used in the program development.

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

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

```
class item
{
int number;
float cost;
public:
void getdata(int a, float b) // declaration

void putdata(void) // definition
{
        cout << "Number :" << number << "\n";
        cout << "cost ." << cost << "\n";
}
};
```

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

```
#include<iostream.h>
#include<conio.h>
class part
{
int mnumber;
int partno;
float cost;
public:
```

```
        void getdata(int a, int b, float c) // declaration
        {
                mnumber=a;
                partno=b;
                cost=c;

}
void putdata(void) // definition
{
        cout << "Number :" << mnumber << "\n";
        cout << "partno :" << partno << "\n";
        cout << "cost ." << cost << "\n";
}
};

void main( )
{
        part p1; // define object of class part
        p1.getdata(234, 3455,234.35); // call member function
        p1.putdata( ); // call member function
        getch();
}
```

Inline Function:

Functions save memory space because all the calls to the function, it causes the same code to be executed; the function body need not be duplicated in memory. When the compiler sees a function call, it normally generates a jump to the function. At the end of the function it jumps back to the instruction following the calls. While this sequence of events may save memory space, it takes some extra time. There must be an instruction for the jump to the function, instructions for saving registers,instructions for pushing arguments onto the stack in the calling program and removing them from the stack in the function (if there are arguments), instructions for restoring registers, and an instruction to return to the calling program. The return value (if any) must also be dealt with.All these instructions slow down the program.

To save execution time in short functions you may elect to put the code in the function body directly in line with the code in the calling program. That is,each time there's a function call in the source file,the actual code from the function is inserted,instead of a jump to the function. The trouble with repeatedly inserting the same code is that you lose the benefits of program organization and clarity that come with using functions.The program may run faster and take less space, but the listing is longer and more complex.

The solution to this quandary is the inline function. This kind of function is written like a normal function in the source file but compiles into inline code instead of into a function.The source file remains well organized and easy to read, since the function is shown as separate entity. However, when the program is compiled, the function body is actually inserted into the program, wherever a function calls occurs. Functions that are very short, say one or two statements, are candidates to be inlined.Here's a program to demonstrate inline function.

```
        #include<iostream.h>
        #include<conio.h>
        //converts pounds to kilograms
        inline float lbstokg(float pounds)
        {
                return 0.453592 * pounds;

        }
```

```
void main( )
{
float lbs;
cout << "Enter the weight in pounds:";
cin >> lbs;
cout << "Your weight in kilograms is" << lbstokg(lbs);
getch();
}
```

The keyword **inline** in the function definition defines a inline function:

```
inline float 1gstokg(float pounds)
```

The compiler must have seen the function definition (not just the declaration) before it gets to the first function call. This is because it must insert the actual code into the program, not just instructions to call the function. In the above program we must place the definition of **lbstokg( )**before **main( ).**When you do this the function declaration is unnecessary and can be eliminated.Inline keyword is actually just a request to the compiler.Sometimes the compiler will ignore the request and compile the function as a normal It might decide the function is too long to be inline.

Class: The class is the C++ vehicle for translating an abstraction to a user-defined type. It combines data representation and methods for manipulating that data into one neat package. An object has the same relationship to a class that a variable has to a data type.An object is said to be an instance of a class, in the same way my 1954 Chevrolet is an instance of a vehicle. A class is a way to bind the data and its associated functions together.It allows the data (and functions)to be hidden,if necessary,from external use.When defining a class we are creating a new abstract data type that can be treated like any other built-in data type.

Generally, a class specification has two parts:

1. Class declaration
2. Class function definitions

The class declaration describes the type and scope of its members.The class function definitions describe how the class functions are implemented. The general form of a class declaration is:

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

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

The members that have been declared as private can be accessed only from within the class.On the other hand, public members can be accessed from outside the class also. The data hiding is the key feature of OOP.The use of the keyword private is optional.By default,the members are private. Such a class is completely hidden from the outside world and does not serve any purpose.

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

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

A Simple Class Example: A typical class declaration would look like:

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

We usually give a class some meaningful name, such as item. This name now becomes a new type identifier that can be used to declare instances of that class type. The class item contains two data members and two function members. The data members are private by default while both the functions are public by declaration. The function getdata( ) can be used to assign values to the member variables number and cost, and putdata( ) for displaying their values. These functions provide the only access to the data members from outside the class. This means that the data cannot be accessed by any function that is not a member of the class item. Note that the functions are declared,not defined.Actual function definitions will appear later in the program. The data members are usual private and the member function

Creating Objects:

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

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

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

```
item x, y, z;
```

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

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

```
class item
{
-----
-----
-----
} x. y, z;
```

would create the objects x, y and z of type item. This practice is seldom followed because we would like to declare the objects close to the place where they are used.

Accessing Class Members:

The private data of a class can be accessed only through the member functions of that class. The main( )

cannot contain statements that access number and cost directly. The following is the format for calling a member function:

object -name. function-name(actual-arguments) ;

For example, the function call statement

x.getdata(100,75.5);

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

x.putdata( );

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

getdata(100,75.5);

has no meaning. Similarly, the statement

x.number = 100;

is also illegal. Although x is an object of the type item to which number belongs, the number (declared private)can be accessed only through a member function and not by the object directly. It may be recalled that objects communicate by sending and receiving messages. This is achieved through the member functions. For example,

x.putdata( );

sends a message to the object x requesting it to display its contents.
The objects can access a variable declared as public directly. Example:

```
class xyz
{
        int x;
        int y;
        public
        int z;
        };
----
----
xyz p;
p.x = 0; // error, x is private
p.z = 10 // OK, z is public
----
----
```

Note that the use of data in this manner defeats the very idea of data hiding and therefore should be avoided.

Defining Member Functions:

Member functions can be defined in two places:

- Outside-the class definition
- Inside the class definition

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

Outside the Class Definition:Member functions that are declared inside a class have to be defined separately outside the class. Their definitions are very much like the normal functions. They should have a function header and a function body.

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

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

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

For instance,consider the member functions getdata( )and putdata( ) as discussed in the previous section. They may be coded as follows:

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

void item :: putdata(void)
{
cout << "Number :" << number << "\n";
cout << "cost ." << cost << "\n";
}
```

Since these functions do not return any value, their return::type is void.

The member functions have some special characteristics that are often used in the program development.

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

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

```
class item
{
int number;
float cost;
public:
void getdata(int a, float b) // declaration
```

```
void putdata(void) // definition
{
        cout << "Number :" << number << "\n";
        cout << "cost ." << cost << "\n";
}
};
```

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

Constructors:

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

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

A constructor is declared and defined as follows:

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

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

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

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

A constructor that accepts no parameters is called the default constructor.If no such constructor is defined, then the compiler supplies a default constructor. Therefore a statement such as

```
A a;
```

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

The constructor functions have some special characteristics:

- They should be declared in the public section.
- They are invoked automatically when the objects are created.
- They do not have return types, not even void and therefore, they cannot return values.
- They cannot be inherited, though a derived class can call the base class constructor.
- Like other C++ functions, they can have default arguments.
- Constructors cannot be virtual. We cannot refer to their addresses,
- An object with a constructor (or destructor) cannot be used as a member of a union.
- They make implicit calls to the operators new and delete when memory allocation is required,

When a constructor is declared for a class, initialisation of the class objects becomes mandatory.

Parameterized Constructors:

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

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

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

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

```
integer int1 ;
```

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

This can be done in two ways:

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

The following declaration illustrates the first method:

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

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

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

This method, sometimes called the shorthand method, is used very often as it is shorter, looks better and is easy to implement. When the constructor is parameterized, we must provide appropriate arguments for the constructor.

Multiple Constructors In A Class:

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

```
integer( ); // No arguments
integer(int, int); // Two arguments
```

In the first case, the constructor itself supplies the data values and no values are passed by the calling program. In the second case, the function call passes the appropriate values from main ( ). C++ permits us to use both these constructors in the same class. For example, we could define a class as follows:

```
class integer
{
        int m, n;
public:
        integer( ) {m = 0; n = 0;} // constructor 1 .

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

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

```
integer l1 ;
```

would automatically invoke the first constructor and set both m and n of l1 to zero.The statement

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

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

Finally, the statement

```
integer I3(l2);
```

would invoke the third constructor which copies the values of I2 into I3. That is, it sets the value of every data element of l3 to the value of the corresponding data element of 12. As mentioned earlier, such a constructor is called the copy constructor. The process of sharing the same name by two or more functions is referred to as function overloading. Similarly, when more than one constructor function is defined in a class, we say that the constructor is overloaded.

Destructors:

A destructor, as the name implies, is used to destroy the objects that have been created by a constructor. Like a constructor, the destructor is a member function whose name is the same as the class name but is preceded by a tilde. For example, the destructor for the class integer can be defined as shown below:

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

A destructor never takes any argument nor does it return any value. It will be invoked implicitly by the compiler upon exit from the program (or block or function as the case may be) to clean up storage that is no

longer accessible. It is a good practice to declare destructors in a program since it release memory space for future use.

Whenever new is used to allocate memory in the constructors, we should use delete to free that memory. A destructor has the same name as the constructor (which is the same as the class name) but proceeded by a tilde. For example

```
class Tarzan
{
private:
        int data;
public:
        Tarzan( ) { data = 0; } // constructor
        ~Tarzan( ) { } // destructor
}
```

The most common use of destructor is to deallocate memory that was allocated for the object by the constructor.

Operator Overloading:

Operator overloading is another example of C ++ polymorphism. We know that C++ enables you to define several functions having the same name as long as they have different signatures(argument lists). That was function overloading, or functional polymorphism. Its purpose is to let you use the same function name for the same basic operation even though you apply the operation to different data type.

Operator overloading extends the overloading concept to operators, letting you assign multiple meanings to C++ operators. Actually many C++ (and C) operators already are overloaded. For example, the * operator,when applied to an address, yields the value stored at that address. But applying * to two numbers yields the product of the values. C++ uses the number and type of operands to decide which action to taken.

C++ lets you extend operator overloading to user-defined types, permitting you, say, to use the + symbol to add two objects. Again, the compiler will use the number and type of operands to determine which definition of addition to use. Overloaded operators often can make code look more natural.

For instance C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types. This means that C++ has the ability to provide the operators with a special meaning for a data type. The mechanism of giving such special meanings to an operator is known as operator overloading.

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

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

The excluded operators are very few when compared to the large number of operators,which qualify for the operator overloading definition.
Although the semantics of an operator c an be extended, we cannot change its syntax, the grammatical rules that govern its use such as the number of operands, precedence and associativity. For example,the multiplication operator will enjoy higher precedence than the addition operator.Remember, when an operator is overloaded, its original meaning is not lost.

Defining Operator Overloading:

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

returntype classname :: operator op (arg-Iist)
{
    Function body //task defined
}

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

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

vector operator + (vector); //vector addition
vector operator -( ); //unary minus
friend vector operator + (vector, vector); //vector addition
friend vector operator -(vector); //unary minus
vector operator -(vector & a); //subtraction
int operator == (vector); //comparison
friend int operator == (vector, vector) //comparison

vector is a data type of class and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements (as in mathematics). The process of overloading involves the following steps:

- First, create a class that defines the data type that is to be used in the overloading operation.
- Declare the operator function operator op ( ) in the public part of the class. It may be either a member function or a friend function.
- Define the operator function to implement the required operations.

Overloaded operator functions can be invoked by expressions such as

    op x or x op

for unary operators and

    x op y

for binary operators.

    op x (OF x op) would be interpreted as operator op (x)

In case of member functions,

    operator op (x, y)

operator Keyword:

The keyword operator is used to overload the ++ operator in this declaration:

    void operator ++ ( )

The return type (void in this case) comes first, followed by the keyword operator, followed by the operator itself (++), and finally the argument list enclosed in parentheses (which are empty here).This declaration syntax tells the compiler to call this member function whenever the ++ operator is encountered,provided the

operand (the variable operated on by the ++) is of type Counter. The only way the compiler can distinguish between overloaded functions is by looking at the data type of their arguments.In the same way,it can distinguish between overloaded operators by looking at the data type of their operands. If the operand is a basic type like an int, as

intvar ++;

then the compiler will use its built-in routine to increment an int. But if the operand is a Counter variable, then the compile will know to use our user-written operator++().

Operator Return Values:

The operator++( ) function has a subtle defect. You will discover it if you use a statement like this in main ( ) :

c1 = c2++;

The compiler will give an error. Because we have defined the ++ operator to have a return type of void in the operator++ ( ) function, while in the assignment statement it is being asked to return a variable of type Counter. That is, the compiler is being asked to return whatever value c2 has after being operated on by the ++ operator, and assign this value to c1.So we can't use ++ to increment Counter objects in expression; it must always stand alone with its operand. To make it possible to use our user defined operator+ + ( ) in expressions, we must provide a way for it to return a value.

Multiple Overloading:

We can have same operator function called for different operand. Consider that you use three different programs of the + operator to add distances, to add polar coordinates, and to concatenate strings. You could put all these classes together in the same program, and C++ would still know how to interpret the + operator. It selects the correct function to carry out the "addition" based on the type of operand.

Data conversion:

Normally, when the value of one object is assigned to another, of the same type, the value, or the member data items are simply copied into the new object. The compiler, doesn't need any special instructions to use = for the assignment of user-defined objects such as Distance objects. Thus assignment between type, whether, they are basic types or user defined types, are handled by the compile, with no effort on our part provided that the same data type is used on both sides of the equals sign. But what happens when the variables on different sides of the = are of different types? The answer to this is Data conversion.

Conversions Between Basic Types:

When we write a statement like

intvar = floatvar;

where intvar is of type int and floatvar is of type float,we are assuming that the compiler will call a special routine to convert the value of floatvar, which is expressed in floating-point format, to an integer format so that it can be assigned to intvar.There are of course many such conversions:from float to double,char to float, and so on. Each such conversion has its own routine, built into the compiler and called up when the data types on different sides of the = sign so dictate. We say such conversions are implicit, because they aren't apparent in the listing. Sometimes we want to force the compiler to convert one type to another. To do this we use the cast operator.

For instance, to convert float to int we could say

intvar = int(floatvar);

Casting provides explicit conversion: It's obvious in the listing that the int( ) conversion function will convert from float to int. However, such explicit conversions use the same built-in routines as implicit conversion.

Basic to Class Type:
The conversion from basic type to class type is easy to accomplish. It may be recalled that the use of constructors was illustrated in a number of examples to initialize objects. For example, a constructor was used to build a vector object from an int type array.Similarly, we used another constructor to build a string type object from a char* type variable.

Consider the following constructor:

```
string :: string(char *a)
{
length = strlen(a);
p = new char[length+ 1];
strcpy (P, a);
}
```

This constructor builds a string type object from a char * type variable a. The variables length and p are data members of the class string. Once this constructor has been defined in the string class, it can be used for conversion from char * type to string type. Example:

```
string S1, S2;
char * name1 = "IBM PC";
char * name2 = "Apple Computers";
S1 =5tring(name1);
S2 = name2;
```

The statement

```
S1 = Strlng(name1);
```

first converts name1 from char * type to string type and then assigns the string type values to the object s1. The statement

```
S2 = name2;
```

also does the same job by invoking the constructor implicitly. Let us consider another example of converting an int type to a class type

```
class time {
int hrs;
int mins;
public :
--
--
time(int t) // constructor
{
hours = t / 60; // t in minutes
mins = t% 60;
}
};
```

The following conversion statements can be used in a function:

```
time = T1 ; I lobject T1 created
int duration = 85;
T1 = duration; Il int to class type
```

After this conversion,the hrs member of TI will contain a value of 1 and mins member a value of 25, denoting 1 hour and 25 minutes.

Note that the constructors used for the type conversion take a single argument whose type is to be converted.In both the examples, the left-hand operand of = operator is always a class object.Therefore, we can also accomplish this conversion using an overloaded = operator.

Class to Basic Type:
The constructors did a fine job in type conversion from a basic to Class type. what about the conversion from a class to basic type? The constructor functions do not support this operation. Luckily, C++ allows us to define an overloaded casting operator that could be used to convert a class type data to a basic type.The general form of an overloaded casting operator function,usually referred to as a conversion function is:

```
operator typename( )
{
----- (Function statements)
-----
-----
}
```

This function converts a class type data to typename.For example,the operator double( ) converts a class object to type double, the oðerator int( ) converts a class type object to type int, and so on. Consider the following conversion function:

```
vector :: operator double( ) {
double sum = 0;
for(int i = 0; i < size; i++)
sum = sum + v [i] * v [i];
return sqrt(sum);
}
```

This function converts a vector to the corresponding scalar magnitude. Recall that the magnitude of a vectoris given by the square root of the sum of the squares of its Components. The operator double( ) can be used as follows:

```
double length = double(V1); or double length = V1 ;
```

where V1 is an object of type vector. Both the statements have exactly the same effect. When the compiler encounters a statement that requires the conversion of a class type to a basic type, it quietly calls the casting operator function to do the job.

The casting operator function should satisfy the following conditions:

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

Since it is a member function, the object invokes it and therefore, the values used for conversion inside the function belong to the object that invoked the function. This means that the function does not need an argument.

In the string example described in the previous section, we can do the conversion from string to char * as follows:

```
string :: operator char*( )
{
return(p);
}
```

One Class to Another Class Type :

We have just seen data conversion techniques from a basic to class type and a class to basic type.But there are situations where we would like to convert one class type data to another class type. Example:

```
objX = objY; // objects of different types
```

objX is an object of class X and objY is an object of class Y.The class Y type data is converted to the class X type data and the converted value is assigned to the objX. Since the conversion takes place from class Y to class X, Y is known as the source class and X is known as the destination class.

Either a constructor or a conversion function can carry out such conversions between objects of different classes. The compiler treats them the same way. Then, how do we decide which - form to use? It depends upon where we want the type-conversion function to be located, in the source class or in the destination class. We know that the casting operator function operator typename( )

converts the class object of which it is a member to typename. The typename may be a built-in type or a user-defined one (another class type). In the case of conversions between objects, typename refers to the destination class. Therefore, when a class needs to be converted, a casting operator function can be used (i.e. source class). The conversion takes place on the source class and the result is given to the destination class object.

Now consider a single-argument constructor function, which serves as an instruction for converting the argument's type to the class type of which it is a member. This implies that the argument belongs to the source class and is class for conversion .This makes it necessary that the conversion constructor be placed in the destination class.

Table provides a summary of all the three conversions. It shows that the conversion from a class to any other type (or any other class)should make use of a casting operator in the source class. On the other hand, to perform the conversion from any other type/class to a class type, a constructor should be used in the destination class.

Table: Type conversions

Conversion Required	**Conversion takes place in**	
	source class	Destination class
Basic -> class	Not applicable	Constructor
Class-> basic	Casting operator	Not applicable
Class ->class	Casting operator	Constructor

When a conversion using a constructor is performed in the destination class, we must be able to access the data members of the object sent (by the source class) as an argument. Since data members of the source class are private,we must use special access functions in the source class to facilitate its data flow to the destination class.

INHERITANCE

Inheritance is a process of creating new classes, called derived classes from the existing class or base class. The derived class inherits all the capabilities of the base class but can add members and refinements of its own. The base class is unchanged by this process. Inheritance has important advantages. Most importantly, it permits code reusability. Reusability is yet another important feature of OOP. C++ strongly supports the concept of reusability. The C++ classes can be reused in several ways. Once a class has been written and tested, it can be adapted by other programmers to suit their requirements. Creating new classes, reusing the properties of the existing ones, basically does this. The mechanism of deriving a new class from an old one is called inheritance (or derivation). The old class is referred to as the base class and the new one is called the derived class.

The derived class inherits some or all of the traits from the base class. A class can also inherit properties from more than one class or from more than one level. A derived class with only one base class is called single inheritance and one with several base classes is called multiple inheritance. On the other hand, more than one class may inherit the traits of one class. This process is known as hierarchical inheritance.The mechanism of deriving a class from another 'derived class' is known as multilevel inheritance. Figure shows various forms of inheritance that could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance.

Defining Derived Classes:

A derived class is defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:

```
class derived-class-name : visibility-mode base-class-name
{
---- //
---- // members of derived class
----//
};
```

The colon indicates that the derived-class-name is derived from the base-class-name. The visibility mode is optional and, if present, may be either private or public. The default visibility-mode is private. Visibility mode specifies whether the features of the base class are privately derived or publicly derived.

Examples:

```
class ABC: private XVZ //private derivation
{
    members of ABC
};
class ABC : public XVZ //public derivation 3
{
    members of ABC
};
class ABC : XVZ //private derivation by default
{
    members of ABC
};
```

When a derived class privately inherits a base class, 'public members' of the base class become 'private members' of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class.They are inaccessible to the objects of the derived class. Remember, a public member of a class can be accessed by its own objects using the dot operator. The result is that no member of the base class is accessible to the objects of the derived class.
On the other hand, when the base class is publicly inherited, 'public members' of the base class become 'public members' of the derived class and therefore they are accessible to the objects of the derived class. In

both the cases, the private members are not inherited and therefore, the private members of a base class will never become the members of its derived class.

In inheritance, some of the base class data elements and member functions are 'inherited'into the derived class. We can add our own data and member functions and thus extend the functionality of the base class. Inheritance, when used to modify and extend the capabilities of the existing classes, becomes a very powerful tool for incremental program development.

Single Inheritance:

A derived class with only one base class is called single inheritance

Accessing Base Class Members:

An important topic in inheritance is knowing when objects of the derived class can use a member function in the base class. This is called accessibility.

Substituting Base Class Constructors: In the main ( ) part of the above program:

```
CountDn c1;
```

This causes c1 to be created as an object of class countDn and initialized to 0. There is no constructor in the CountDn class specifier, if you don't specify a constructor,the derived class will use an appropriate constructor from the base class. This flexibility on the part of the compiler using one function because another isn't available is feature of inheritance. Generally the substitution is what you want, but sometimes it can be error also.

Substituting Base Class Member Functions: The object c1 of the countDn class also uses the operator++( ) and get-count ( ) functions from the counter class. The first statement is used to increment c1:

```
c1++;
```

and the second statement is used to display the count in c1:

```
cout << "\nt1=" << t1.get_count( );
```

Again the compiler, not finding these functions in the class of which c1 is a member, uses member functions from the base class. The ++ operator, the constructors, and the get_count( ) function in the counter class, and the - - operator in the countDn class, all work with objects of type countDn.

Making A Private Member Inheritable (protected):

C++ provides a third visibility modifier, protected,which serves a limited purpose in inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it.It cannot be accessed by the functions outside these two classes.

When a protected member is inherited in public mode, it becomes protected in the derived class too, and therefore is accessible by the member functions of the derived class. It is also ready for further inheritance. A protected member, inherited in the private mode derivation, becomes private in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance (since private members cannot be inherited). Table summarizes how the visibility of members undergo modifications when they are inherited.

Table: Visibility of inherited members

Base class visibility	Derived class visibility	
	Public derivation	Private derivation
Private	Not inherited	Not inherited
Protected	Protected	Private
Pubic	Pubic	Private

Derived Class Constructors:

The compiler will substitute a no-argument constructor from the base class, when there is no constructor for the derived class, but it draws the line at more complex constructors. To make such a definition work we must write a new set of constructors for the derived class

Overriding member functions:

You can use member functions in a derived class that have the same name as those in the base class. You might want to do this so that calls in your program work the same way for objects of both base and derived classes.

Multilevel Inheritance:

It is not uncommon that a class is derived from another derived class as shown.The class A serves as a base class for the derived class B that in turn serves as a base class for the derived class C. The class B is known as intermediate base class since it provides a link for the inheritance between A and C. The chain ABC is known as inheritance path.

```
class A{ }; //Base class
class B: public A { }; //B derived from A
class C: public B { }; //C derived from B
```

This process can be extended to any number of levels.

Multiple Inheritance:

A class can inherit the attributes of two or more classes. This is known as multiple inheritance. Multiple inheritance allows us to combine the features of several existing classes as a starting point for defining new classes. It is like a child inheriting the physical features of one parent (papa) and the intelligence of another (mama). The syntax of a derived class with multiple base classes is as follows:

```
Class D : visibility B-1, visibility B-2,
{
-----
----- (Body of D)
-----
};
```

where, visibility may be either public or private. The base classes are separated by commas.

Hybrid Inheritance:

There could be situations where we need to apply two or more types of inheritance to design a program.For instance, consider the case of processing the student results. Assume that we have to give weightage for sports before finalising the results. The weightage for sports is stored in a separate class called sports.

Containership: Classes Within Classes:

In inheritance, if a class B is derived from a class A, we can say "B is a kind of A". This is because B has all the characteristics of A, and in addition some of its own. For this reason inheritance is sometimes called a **"kind of"** relationship:

There's another kind of relationship, called a "has a" relationship, or containership. In object-oriented programming the "has a" relationship occurs when one object is contained in another. Here's a case where an object of class B is contained in a class A:

```
class A
{
    B b; // b is an object of class B
};
class B
{ };
```

In some situations inheritance and containership relationships can serve similar purposes.

Containership is clearly useful with classes that act like a data type. Then an object of that type can be used in a class in almost the same way a variable would be. In other situations you will need to examine the problem carefully and perhaps try different approaches to see what makes sense. Often the inheritance relationship is simpler to implement and offers a clearer conceptual framework.

POINTERS

Introduction:

Pointers are variables that hold address values. Pointers are defined using an asterisk (*) to mean pointer to. A data type is always included in pointer definitions, since the compiler must know what is being pointed to, so that it can perform arithmetic correctly on the pointer. We access the thing pointed to using the asterisk in a different way, as the indirection operator, meaning contents of the variable pointed to by.

Addresses and Pointers:

The ideas behind pointers are not complicated. Here's the first key concept: Every byte in the computer's memory has an address. Addresses are numbers,just as they are for houses on a street. The numbers start at 0 and go up from there-l, 2, 3, and so on. If you have 640K of memory, the highest address is 655,359; for 1 MB of memory, it is 1,048,575.

Your program, when it is loaded into memory, occupies a certain range of these addresses. That means that every variable and every function in your program starts at a particular address.

The Address Of Operator &:

You can find out the address occupied by a variable by using the address of operator &. Here's a short program that demonstrates this:

```
// address of variables
# include< iostream.h>
void main ( )
{
int var1 = 11; // define and initialize
int var2 = 22; // three variables
int var3 = 33;

cout << endl << &var1 // print out the addresses
        << endl << &var2 // of these variables
        << endl << &var3;
}
```

This simple program defines three integer, variables and initializes them to the values 11, 22, and 33. It then prints out the addresses of these variables. The actual addresses occupied by the variables in a program depend on many factors, such as the computer the program is running on, the size of the operating system, and whether any other programs are currently in memory. For these reasons you probably won't get the same addresses we did when you run this program.

Here's the output on our machine:

Oxt8f4ffff4	address of varl
Ox8f4ffff2	address of var2
Ox8f4ffffO	address of var3

The address of a variable is not at all the same as its contents. The contents of the three variables are 11,22, and 33. The << insertion operator interprets the addresses in hexadecimal arithmetic,as indicated by the prefix Ox before each number.This is the usual way to show memory addresses.

The addresses appear in descending order because automatic variables are stored on the stack, which grows downward in memory. If we had used external variables they would have ascending addresses, since external variables are stored on the heap, which grows upward.

Pointer Variable:
Address by themselves are rather limited.It's nice to know that we can find out where things are in memory,but printing out address values is not all that useful. The potential for increasing our programming power requires an additional idea: variables that hold address values. Addresses are stored similarly as variables of different data types.A variable that holds an address value is called a pointer variable or simply a pointer.

Declaring and Initializing Pointers:
Let's examine the process of declaring pointers. A computer needs to keep track of the type of value to which a pointer refers. For example,the address of a char looks the same as the address of a double, but char and double use different numbers of bytes and different internal formats for storing values. Therefore, a pointer declaration must specify what type of data it is to which the pointer points. For example, the statement.

int * p_updates;

This states that the combination *** p-updates** is type **int**. Because the ***** operator is used by applying it to a pointer, the **p_updates** variable itself must be a pointer. We say that p_updates points to type int. We also say that the type for p_updates is pointer-to-int or,more concisely, int *. To repeat **p_updates** is a pointer (an address}, and ***p-updates**is an int and not a pointer.

Incidentally, the use of spaces around the * operator are optional. For example:

int* ptr;

This emphasizes the idea that int* is a type, pointer-to-int. Where you put the spaces makes no difference to the compiler. Be aware, however, that the declaration

int* p1,p2;

Creates one pointer (p1) and one ordinary int variable (p2)

If you define more than one pointer of the same type on one line, you need only insert the type-pointed-to once, but you need to place an asterisk before each variable name:

char * ptr1, * ptr2, * ptr3; // three variables of type char*

Or you can use the asterisk-next-to-the-name approach

char *ptr1, *ptr2, *ptr3; // three variables of type char*

Pointers Must Have a Value: An address like OxHf4ffff4 can be thought of as a pointer constant. A pointer like ptr can be thought of as a pointer variable.Just as the integer variable var1 can be assigned the constant value 11, so can the pointer variable ptr be assigned the constant value Ox8f4ffff4.

When we first define a variable, it holds no value (unless we initialize it at the same time) it may hold a garbage value, but this has no meaning. In the case of pointers, a garbage value is the address of something in memory,but probably not of something that we want.So before a pointer is used,a specific address must be placed in it.

ptr = &var1; // put address of var l in ptr.

Following this the program prints out the value contained in ptr,which should be the same address printed for &var1.

To summarize:A pointer can hold the address of any variable of the correct type; However, it must be given some value, otherwise it will point to an address we don't want it to point to, such as into our program code or the operating system. Wrong pointer values can result in system crashes and are difficult to debug, since the compiler gives no warning.

Accessing the Variable Pointed to:

Suppose that we don't know the name of a variable but we do know its address. We can access the contents of the variable, by using a special syntax (indirection operator)to access the value of such variable.

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

void main( )
{
        int var1 =11; //two integer variables
        int var2 = 22;
        int* ptr; // pointer to integers
        ptr = &var1; // pointer points to var1
        cout << endl<< *ptr; // pointer contents of pointer (11)
        ptr = &var2; // pointer points to var2
        cout << endl << *ptr; // pointer contents of pointer (22)
        getch();
}
```

When an asterisk is used in front of a variable name, as it is in the * ptr expression; it is called the indirection operator. It means the value of variable pointed to by. Thus the expression *ptr represents the value of the variable pointed to by ptr. When ptr is set to the address of var1, the expression *ptr has the value 11; since var1 is 11. When ptr is changed to the address of var2, the expression *ptr acquires the value 22, since var1 is 22.You can use a pointer not only to display a variable's value but to perform any operation you would perform on the variable directly. The asterisk used as the indirection operator has a different meaning from the asterisk used to declare pointer variables.The indirection o perator precedes the variable and means value of the variable pointed to by the asterisk used in a declaration means pointer to.

```
int * ptr; // declaration:
*ptr =37; // indirection:
```

Using the indirection operator to access the value stored in an address is called indirect addressing, or sometimes dereferencing, the pointer:

```
int v; // defines variable v of type int
int* p; // defines p as a pointer to int
p = &v; // assigns address of variable v to pointer p
v = 3; // assigns 3 to v
*p = 3; // also assigns 3 to V
```

this Pointer:

The this pointer is predefined in member functions to point to the object of which the function is a member.The this pointer is useful in returning the object of which the function is a member. The member functions of every object have access to a sort of magic pointer named this, which points to the object itself. Thus any member function can find out the address of the object of which it is a member.

Virtual Functions:

Polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. An essential requirement of polymorphism is therefore the ability to refer to, objects without any regard to their classes. This necessitates the use of a single pointer variable to refer to the objects of different classes. Here, we use the pointer to base class to refer to all the derived objects. But, we just discovered that a base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class. The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer. How do we then achieve polymorphism? It is achieved using what is known as 'virtual' functions.

Virtual means existing in effect but not reality. A virtual function then, is one that does not really exist but nevertheless appears real to some parts of a program. Virtual functions provide a way for a program to decide: when it is running, what function to call. Ordinarily such decisions are made at compile time. Virtual functions take possible greater flexibility in performing the same kind of action on different kinds of objects. In particular, they allow the use of functions called from an array of type pointer-to-base that actually holds pointers to a variety of derived types. Typically a function is declared virtual in the base class, and functions with the same name are declared in derived classes.A pure virtual function has no body in the base class.

When we use the same function name in both the base and derived class the friction in base class is declared as virtual using the keyword virtual preceding its normal declaration.When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer rather than the type of the pointer, Thus by making the base pointer to point to different objects, we can execute different versions of the virtual function.

Rules for Virtual Functions:

When virtual functions are created for implementing late binding, we should observe some basic rules that satisfy the compiler requirements:

- The virtual functions must be members of some class.
- They cannot be static members.
- They are accessed by using object pointers.
- A virtual function can be a friend of another class.
- A virtual function in a base class must be defined, even though it may not be used.
- The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototype, C++ considers them as overloaded functions,and the virtual function mechanism is ignored.
- We cannot have virtual constructors/ but we can have virtual destructors.
- While a base pointer can point to any type of the derived object, the reverse is not true.That is,we cannot use a pointer to a derived class to access an object of the base type.
- When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class.It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
- If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Virtual Member Functions Accessed With Pointers:

We can make use of pointers to access the virtual member function.

We can make use of pointers to access the virtual member function. The program below illustrates this.

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

```
{
public:
    virtual void show( )
    {
    cout<<"\n base";
    }
};
class der1: public base //derived class 1
{
public:
    void show( )
    { cout<< "\n Derv1 "; }
};

class der2: public base //derived class 2
{
public:
    void show( )
    { cout<< "\n Derv2 "; }

};

void main( )
{
    der1 dv1; //object of derived class 1
    der2 dv2; //object of derived class 2
    der1 *ptr =&dv1; // put address of dv1 in pointer
    ptr->show( ); //execute show ( )
    der2 ptr =&dv2; // put address of dv2 in pointer
    der2 *ptr1->show( ); //execute show ( )
    getch();
}
```

The member functions of the derived classes are executed and not the base class. The function call

ptr -> show( );

executes different functions, depending on the contents of ptr. The rule is that the compiler selects the function based on the contents of the pointer ptr. The compiler has no problem with the expression

ptr -> show( )

It always compiles a call to the show( ) function in the base class.But in the program above the compiler doesn't know what class the contents of ptr may contain. It could be the address of an object of the Derv1 class or of the Derv2 class.Which version of draw( ) does the compiler call? In fact the compiler doesn't know what to do,so it arranges for the decision to be deferred until the program is running. At run time, when it is known what class is pointed to by ptr, the appropriate version of draw will be called. This is called late binding or dynamic binding. (Choosing functions in the normal way, during compilation, is called early, or static, binding.) Late binding requires some overhead but provides increased power and flexibility.

Pure Virtual Functions:

A pure virtual function is a virtual function with no body. It is a normal practice to declare a function virtual inside the base class and redefine it in the derived classes.The function inside the base class is seldom used

for performing any task,It only serves as a placeholder,For example, we have not defined any object of class media and therefore the function display( ) in the base class has been defined 'empty'. Such functions are called "do- nothing" functions, A 'do-nothing"function may be defined as follows:

```
virtual void display( ) = 0;
```

Such functions are called pure virtual functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function, Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. Such classes are called abstract base classes. The main objective of an abstract base class is to provide some traits, to the derived classes and to create a base pointer required for achieving runtime polymorphism

```
// Program to demonstrate pure virtual function
# include <iostream.h>
# include <conio.h>
class Base
{
public:
    virtual void show( ) = 0; // pure virtual function

} ;

class Derv1 : public Base // derived class 1
{
public:
    void show( )
    {
    cout << "\nDerv1"; }
};

class Derv2 : public Base //derived class 2
{ public :
    void show( )
    { cout << "\nDerv2"; }
};

void main( ) {
    Base* list[2]; // list of pointers to base class
    Derv1 dv1; // object of derived class 1
    Derv2 dv2; // object of derived class 2
    list[0] = &dv1; // put address of dv1 in list
    list[1] = &dv2; // put address of dv2 in list
    list[0]->show( ); // execute show( ) in both objects
    list[1]->show( );
    getch();
}
```

Now the virtual function is declared as

```
virtual void show( ) = 0; // pure virtual function
```

The equals sign here has nothing to do with assignment; the value 0 is not assigned to anything. The = 0 syntax is simply how we tell the compiler that a function will be pure that is, have no body. You might wonder, if we can remove the body of the virtual show ( ) function in the base class, why we can't remove the function altogether. That would be even cleaner, but it doesn't work. Without a function show( ) in the

base class statements like

list[0]->show();

would not be valid,because the pointers in the list[ ] array must point to members of class Base. The addresses of the member functions are stored-in an array of pointers, and accessed using array elements. This works in just the same way as using a single pointer.

Abstract classes:

An abstract class is one that is not used to create object. It is designed only to act as a base class (to be inherited by other classes). It is a design concept in program development and provides a base upon which other classes may be built. It is often defined as one that will not be used to create any objects, but exists only to act as a base class of other classes.

8.6 Virtual Base Classes:

Virtual base classes allow an object derived from multiple bases that themselves share a common base to inherit just one object of that shared base class.

Consider the situation with a base class, Parent; two derived classes Child1 and Child2; and a fourth class, Grandchild, derived from both Child1and Child2.In this arrangement a problem can arise if a member function in the Grandchild class wants to access data or functions in the Parent class.

```
//ambiguous reference to base class
class Parent
{
protected:
int basedata;
};

class Child:public parent
{ };

class Child2 : public Parent
{ };

class Grandchild: public Child, public Child2
{
public:
int getdata( )
{ return basedata; } // ERROR: ambiguous
};
```

A compiler error occurs when the getdata ( ) member function in Grandchild attempts to access base data in Parent. Because the Child1 and Child2 classes are derived from Parent,each inherits a copy of Parent; this copy is called a subobject. Each of the two subobjects contains its own copy of Parent's data, including basedata. Now, when Grandchild refers to basedata, which of the two copies will it access? The situation is ambiguous, and that's what the compiler reports. The duplication of inherited members due to these multiple paths can be avoided by making the common base class as virtual base class. To eliminate the ambiguity in the above program, we make Child1 and Child2 into virtual base classes.

```
class Parent
{
protected:
int basedata;
};

class Child1 : virtual public Parent // shares copy or Parent
{ };
```

class Child2 : virtual public Parent // shares copy of Parent
{ } ;

class Grandchild: public Child, public Child2
{
public:
int getdata( )
{ return basedata; } // OK: only one copy of Parent };

The use of the keyword virtual in these two classes causes them to share a single common subobject of their base class Parent. Now, since there is only one, copy of basedata, there is no ambiguity when it is referred to in Grandchild.

TEMPLATES

- Templates are used to define generic classes
- A template can be used to create a family of classes and functions
- The template definition for a class is substituted with the required data type when an object is defined. So templates are also called parameterized classes or functions
- General format of a class template
- **template <class T>**
  class classname
  { ….
  //class member specification with
  //anonymous type wherever appropriate
  };
- A class created from a class template is called template class
- The syntax of defining an object of a template class is:
  - classname <type> objectname(arglist);
- Instantiation - The process of creating a specific class from a class template. The compiler perform the error analysis at the time of instantiation

```
//Example of class Template
#include <iostream.h>
const size =3;
template <class T>
class vector
{
        T *v;
        public:
        vector()
        {
                v = new T[size];
                for(int i=0;i<size;i++)
                        v[i] = 0;
        }

        vector (T *a)
        {
                for(int i=0;i<size;i++)
```

```
                    v[i] = a[i];
        }
T operator *(vector &y)
        {
            T sum = 0;
            for(int i=0;i<size;i++)
                sum = sum + this->v[i] * y.v[i];
            return(sum);
        }
};

void main(void)
{
        int x[3] = {1,2,3};
        int y[3] = {4,5,6};
        vector <int> v1;
        vector <int> v2;
        v1 = x;
        v2 = y;
        int R = v1 * v2;
        cout<<"R   = "<<R<<"\n";
}
```

If you want to work with the float type, modify the main function as follows: (There is no need to modify the class definition)

```
void main(void)
{
        float x[3] = {1,2,3};
        float y[3] = {4,5,6};
        vector <float> v1;
        vector <float> v2;
        v1 = x;
        v2 = y;
        float R = v1 * v2;
        cout<<"R   = "<<R<<"\n";
}
```

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

- More than one generic data type can be used in a class template as shown below:

```
template <class T1, class T2, …)
{   ….
    …. (body of the class)
    ….
};
```

EXAMPLE

```
//class template with multiple parameters
#include <iostream.h>

template <class T1, class T2>
Class Test
{
        T1 a;
```

```
        T2 b;
    public:
        Test(T1 x, T2 y)
        {
            a = x;
            b = y;
        }
        void show(void)
        {
            cout<<"a = "<<a<<" b = "<<a<<"\n";
        }
};
void main()
{
        Test <float, int> test1(1.25,10);
        Test <int, char> test2(100,'A');
        test1.show();
        test2.show();
}
```

FUNCTION TEMPLATES

- Used to create a family of functions with different argument types

General format:

```
        template <class T>
        return type functionname(arguments of type T)
        {
            ….
            //body of function with type T
                //wherever appropriate
        }
//Example of function Template
#include <iostream.h>

template <class T>
void swap(T &x, T &y)
{
        T temp = x;
        x = y;
        y = temp;
}
void fun(int m, int n, float a, float b)
{
        cout<<"m and n before swap  : "<<m<<"  "<<n<<"\n";
        swap(m.n);
        cout<<"m and n after swap  : "<<m<<"  "<<n<<"\n";
        cout<<"a and b before swap  : "<<a<<"  "<<b<<"\n";
        swap(a,b);
        cout<<"a and b before swap  : "<<a<<"  "<<a<<"\n";
}
void main(void)
{
        fun(100,200,11.22, 33.44);
}
```

FUNCTION TEMPLATES WITH MULTIPLE PARAMETERS

- More than one more than one generic data type in the template statement as:

```
template <class T1, class T2,…>
return type functionname(arguments of type T1,T2,…)
{     ….
      …. (body of the function)
      ….
}
```

```
//function with two generic types
#include <iostream.h>
#include <string.h>
template <class T1, class T2>
void display(T1 x, T2 y)
{
        cout<<"x = "<<x"  y = "<<y;
}

void main()
{
        display(2010,"MPTC");
        display(12.34,100);
}
```

EXCEPTION HANDLING

- TWO TYPES OF COMMON ERRORS : SYNTAX and LOGIC
- EXCEPTIONS : RUN TIME ANOMALIES
- Eg: Division by zero, access to an array outside of its bounds
- Exceptions must be identified and dealt with effectively
- C++ provides built-in features to detect and handle exceptions

- TWO TYPES OF EXCEPTIONS: SYNCHRONOUS and ASYNCHROUNOUS

SYNCHRONOUS EXCEPTIONS

- Caused by program instructions like division by zero, array out of boundary etc
- C++ handles only synchronous exceptions

ASYNCHRONOUS EXCEPTIONS

- Caused by events beyond the control of the program. Eg: keyboard interrupts
- Exception handling mechanism provide a means to detect and report an "exceptional circumstance" so that appropriate action can be taken

The error handling code performs the following tasks:

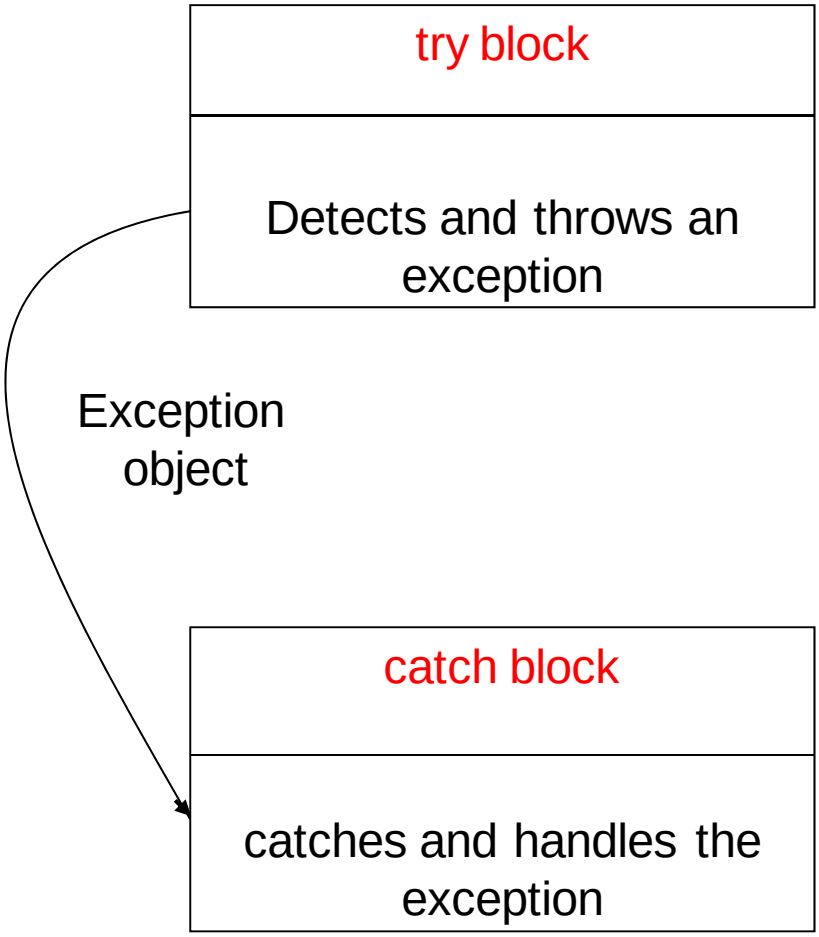
1. Find the problem (hit the exception)
2. Inform that an error has occurred (throw the exception)
3. Receive the error information (catch the exception)
4. Take corrective actions (handle the exception)

The error handling code consists of two segments:

1. To detect errors and to throw exceptions

2. To catch the exceptions and to take appropriate actions

EXCEPTION HANDLING MECHANISM IN C++



THE GENERAL FORM OF EXCEPTION HANDLING

```
try
{
        throw exception;     //block of statements which detects and
                             //throws an exception
}
catch(type arg)
{
        ……
        ……                   //block of statements that handles
        ……                   //the exception
        ……
}
```

try Block

- Perform a block of statements which may generate exceptions
- When an exception is detected, it is thrown using a throw statement in the try block

catch Block

- Catches the exception thrown by the throw statement in the try block and handles it

appropriately

- When the try block throws an exception, the program control leaves the try block and enters the catch statement.

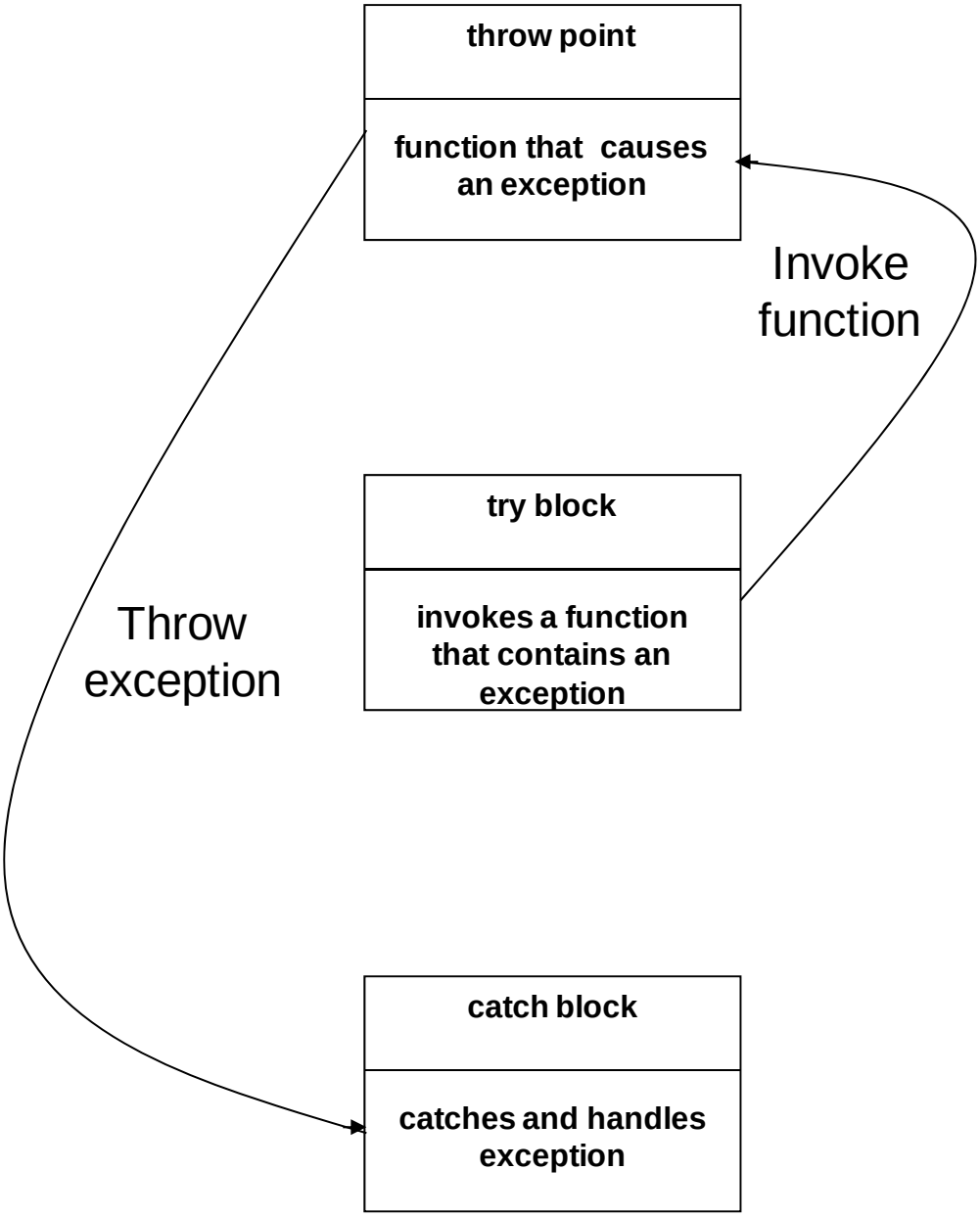
- If the type of the object matches the argtype in the catch statement, then the catch block is executed for handling the exception.

- If they do not match, the program is aborted.

- When no exception is detected and thrown, the control goes to the statement immediately after the catch block

- throw point: the point at which the throw is executed. Once an exception is thrown to the catch block, the control cannot return to the throw point

```
//try block throwing an exception
#include <iostream.h>
void main()
{       int a, b;
        cout<<"\nEnter the values of a and b : ";
        cin>>a>>b;
        int x  = a – b;
        try
        {
            if(x!=0)
            {
                cout<<"Result of a/x = "<<a/x;
            }
            else            //there is an exception
            {
                throw(x);       //throws int object
            }
        }
        catch(int i)
        {
            cout<<"Exception caught: x = "<<x;
        }
        cout<<"END";
}
```




```
type function(arg list)          //function with exception
{           ……
            throw(object);              //throws exception
            ……
}
……
……
try
{           ……..
            Invoke function here
            ……..
}
catch(type arg)
{
            Handles exception here
}
……

//throw point outside the try block
#include <iostream.h>
```

```
void divide(int x, int y, int z)
{
        if((x – y) != 0)
        {   int R = z/(x – y);
            cout<<“Result = “<<R;
        }
        else
            throw(x – y);   //throw point
}

void main()
{
        try
        {
            divide(10,20,30);
            divide(10,10,20);
        }
        catch(int i)
        {
            cout<<“Caught the exception”;
        }
}
```

MULTIPLE catch STATEMENTS

- It is possible that a program segment has more than one condition to throw an exception as shown below:

```
try
{
        //try block
}
catch(type1 arg)
{
        //catch block1
}
catch(type2 arg)
{
        //catch block2
}
……
catch(typeN arg)
{
        //catch blockN
}
```

When an exception is thrown, the exception handlers are searched in order for an appropriate match. When a match is found, the after executing the handler, the control goes to the first statement after the last catch block for that try. When no match is found, the program is terminated. If arguments of several catch statements match the type of an exception, the first handler that matches the exception type is executed

```
//Example using multiple catch statements
#include <iostream.h>
void test(int x)
{       try
        {   if(x==1)
                throw x;
            else
```

```
            if(x==0)
                  throw ‘x’;
            else
            if(x==-1)
                  throw 1.0;
            cout<<”End of the try block”;
      }
      catch(char c)          //catch 1
      {
            cout<<”Caught a character”;
      }
      catch(int m) //catch 2
      {
            cout<<”Caught an integer”;
      }
      catch(double e)       //catch 3
      {
            cout<<”Caught a double”;
      }
      cout<<”End of the try-catch system”;
}

void main()
{
      cout<<”Testing multiple catches”;
      cout<<”x  == 1 “;
      test(1);
      cout<<”x  == 0 “;
      test(0);
      cout<<”x  ==  -1 “;
      test(-1);
      cout<<”x  == 2 “;
      test(2);
}
```