Chapter 1

Overview of Object Oriented Programming (OOP)

Procedural Programming and Issues with Procedural Programming

Pascal, C, BASIC, FORTRAN, and similar languages are procedural languages. A program in a procedural language is a list of instructions and each instruction tells the computer to do something. To write a program, a programmer creates a list of instructions, and the computer carries them out. No other organizing principle or paradigm is needed for writing programs. Later, the concepts of functions and modules were introduced as a part of structured programming to make large programs more comprehensible. Some issues with procedural programming are:

- **Problems with Structured Programming:** As programs grow larger, even structured programming approach begins to show signs of strain. No matter how well the structured programming approach is implemented, the project becomes too complex, the schedule slips, more programmers are needed, and costs skyrocket.
- **Data Undervalued:** Data is given second-class status in the organization of procedural languages. A global data can be corrupted by functions. Since many functions access the same global data, the way the data is stored becomes critical.
- **Relationship to the Real World:** Procedural programs are often difficult to design because their chief components – functions and data structures – don't model the real world very well.
- **New Data Types:** It is difficult to create new data types with procedural languages. Furthermore, most Procedural languages are not usually extensible and hence procedural programs are more complex to write and maintain.

Structured Programming

Structured programming (sometimes known as modular programming) is a subset of procedural programming that enforces a top-down design model, in which developers map out the overall program structure into separate subsections to make programs more efficient and easier to understand and modify. A defined function or set of

similar functions is coded in a separate module or sub-module, which means that code can be loaded into memory more efficiently and that modules can be reused in other programs. In this technique, program flow follows a simple hierarchical model that employs three types of control flows: sequential, selection, and iteration.

Almost any language can use structured programming techniques to avoid common pitfalls of unstructured languages. Most modern procedural languages include features that encourage structured programming. Object-oriented programming (OOP) can be thought of as a type of structured programming, uses structured programming techniques for program flow, and adds more structure for data to the model. Some of the better known structured programming languages are Pascal, C, PL/I, and Ada.

Object-oriented Programming

The fundamental idea behind object-oriented programming is to combine or encapsulate both data (or instance variables) and functions (or methods) that operate on that data into a single unit. This unit is called an object. The data is hidden, so it is safe from accidental alteration. An object's functions typically provide the only way to access its data. In order to access the data in an object, we should know exactly what functions interact with it. No other functions can access the data. Hence OOP focuses on data portion rather than the process of solving the problem.

An object-oriented program typically consists of a number of objects, which communicate with each other by calling one another's functions. This is called sending a message to the object. This kind of relation is provided with the help of communication between two objects and this communication is done through information called message. In addition, object-oriented programming supports encapsulation, abstraction, inheritance, and polymorphism to write programs efficiently. Examples of object-oriented languages include Simula, Smalltalk, C++, Python, C#, Visual Basic .NET and Java etc.

Object-Oriented Concepts

The basic concepts underlying OOP are: Class, Object, abstraction, encapsulation, inheritance, and polymorphism.

- **Abstraction:** Abstraction is the essence of OOP. Abstraction means the representation of the essential features without providing the internal details and complexities. In OOP, abstraction is achieved by the help of class, where data and methods are combined to extract the essential features only. **Encapsulation:** Encapsulation is the process of combining the data (called fields or attributes) and functions (called methods or behaviors) into a single framework called class. Encapsulation helps preventing the modification of data from outside the class by properly assigning the access privilege to the data inside the class. So the term **data hiding** is possible due to the concept of encapsulation, since the data are hidden from the outside world.
- **Inheritance:** Inheritance is the process of acquiring certain attributes and behaviors from parents. For examples, cars, trucks, buses, and motorcycles inherit all characteristics of vehicles. Object-oriented programming allows classes to inherit commonly used data and functions from other classes. If we derive a class (called derived class) from another class (called base class), some of the data and functions can be inherited so that we can reuse the already written and tested code in our program, simplifying our program.
- **Polymorphism:** Polymorphism means the quality of having more than one form. The representation of different behaviors using the same name is called polymorphism. However the behavior depends upon the attribute the name holds at particular moment.

Advantages of Object-oriented Programming

Some of the advantages are:

- Elimination of redundant code due to inheritance, that is, we can use the same code in a base class by deriving a new class from it.
- Modularize the programs. Modular programs are easy to develop and can be distributed independently among different programmers.
- Data hiding is achieved with the help of encapsulation.
- Data centered approach rather than process centered approach.

- Program complexity is low due to distinction of individual objects and their related data and functions.
- Clear, more reliable, and more easily maintained programs can be created.

Object-based Programming

The fundamental idea behind object-based programming is the concept of objects (the idea of encapsulating data and operations) where one or more of the following restrictions apply:

1. There is no implicit inheritance
2. There is no polymorphism
3. Only a very reduced subset of the available values are objects (typically the GUI components)

Object-based languages need not support inheritance or subtyping, but those that do are also said to be object-oriented. An example of a language that is object-based but not object-oriented is Visual Basic (VB). VB supports both objects and classes, but not inheritance, so it does not qualify as object-oriented. Sometimes the term object-based is applied to prototype-based languages, true object-oriented languages that do not have classes, but in which objects instead inherit their code and data directly from other template objects. An example of a commonly used prototype-based language is JavaScript.

Output Using cout

The keyword *cout* (pronounced as 'C out') is a predefined stream object that represents the standard output stream (i.e monitor) in C++. A *stream* is an abstraction that refers to a flow of data.

The << operator is called *insertion* or *put to* operator and directs (inserts or sends) the contents of the variable on its right to the object on its left. For example, in the statement *cout<<"Enter first value:";* the << operator directs the string constant "Enter first value" to *cout*, which sends it for the display to the monitor.

Input with cin

The keyword *cin* (pronounced 'C in') is also a stream object, predefined in C++ to correspond to the standard input stream (i.e keyword). This stream represents data coming from the keyboard.

The operator >> is known as *extraction* or *get from* operator and extracts (takes) the value from the stream object *cin* its left and places it in the variable on its right. For example, in the statement *cin>>first;*, the >> operator extracts the value from *cin* object that is entered from the keyboard and assigns it to the variable *first*.

Cascading of I/O operators

We can use insertion operator (<<) in a *cout* statement repeatedly to direct a series of output streams to the *cout* object. The streams are directed from left to right. For example, in the statement cout<<"Sum="<<x + y;, the string "Sum=" will be directed first and then the value of x +y.

Similarly, we can use extraction operator (>>) in a *cin* statement repeatedly to extract a series of input streams from the keyboard and to assign these streams to the variables, allowing the user to enter a series of values. The values are assigned from left to right. For example, in the statement cin>>x>>y;, the first value entered will be assigned to the variable x and the second value entered will be assigned to variable y.

Manipulators

Manipulators are the operators used with the insertion operator (<<) to modify or manipulate the way data is displayed. The most commonly used manipulators are **endl**, **setw**, and **setprecision**.

The endl Manipulator

This manipulator causes a linefeed to be inserted into the output stream. It has the same effect as using the newline character '\n'. for example, the statement

cout<<"First value ="<<first<<endl<<"Second value ="<<second;

will cause two lines of output.

The setw Manipulator

This manipulator causes the output stream that follows it to be printed within a field of n characters wide, where n is the argument to setw(n). The output is right justified within the field. For example,

```
cout<<setw(11)<<"Kantipur"<<endl <<setw(11)<<"Engineering"<<endl
    <<setw(11)<<"College";
```

Output:

```
   Kantipur
Engineering
    College
```

The setprecision Manipulator

This manipulator sets the n digits of precision to the right of the decimal point to the floating point output, where n is the argument to setprecision(n). For example,

```
float a = 42.3658945, b = 35.24569, c = 58.3214789, d = 49.321489;
cout<<a<<endl <<setprecision(3)<<b<<endl<<c<<endl<<setprecision(2)<<d;
```

Output:

```
42.365894
35.246
58.321
49.32
```

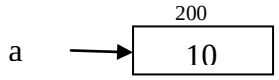
Note: The header file for setw and setprecision manipulators is **iomanip.h**.

Reference Variable

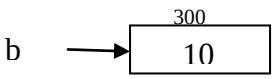
Reference variable is an alias (another name) given to the already existing variables of constants. When we declare a reference variable memory is not located for it rather it points to the memory of another variable.

Consider the case of normal variables

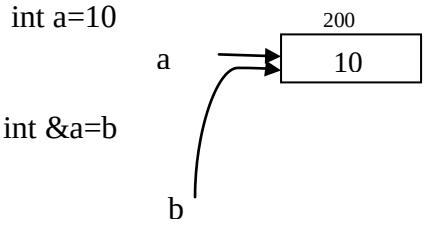
int a=10



int a=b



Consider the case of reference variables



Type Conversion

There are two types of type conversion: automatic conversion and type casting.

Automatic Conversion (Implicit Type Conversion)

When two operands of different types are encountered in the same expression, the lower type variable is converted to the type of the higher type variable by the compiler automatically. This is also called type promotion. The order of types is given below:

Data Type	Order
long double	(highest)
double	
float	
long	
int	
char	(lowest)

Type Casting

Sometimes, a programmer needs to convert a value from one type to another in a situation where the compiler will not do it automatically. For this C++ permits explicit type conversion of variables or expressions as follows:

(type-name) expression //C notation

type-name (expression) //C++ notation

For example,

```
int a = 10000;
int b = long(a) * 5 / 2; //correct
int b = a * 5/2; //incorrect (can you think how?)
```

Default Arguments

When declaring a function we can specify a default value for each of the last parameters which are called default arguments. This value will be used if the corresponding argument is left blank when calling to the function. To do that, we simply have to use the assignment operator and a value for the arguments in the function declaration. If a value for that parameter is not passed when the function is called, the default value is used, but if a value is specified this default value is ignored and the passed value is used instead. For example:

```
// default values in functions
#include <iostream>
using namespace std;
int divide (int a, int b=2)
{
        int r;
        r=a/b;
        return (r);
}
int main ()
{
        cout << divide (12);
        cout << endl;
        cout << divide (20,4);
        return 0;
}
```

As we can see in the body of the program there are two calls to function divide. In the first one:

divide (12)

We have only specified one argument, but the function divide allows up to two. So the function divide has assumed that the second parameter is 2 since that is what we

have specified to happen if this parameter was not passed (notice the function declaration, which finishes with int b=2, not just int b). Therefore the result of this function call is 6 (12/2).

In the second call:

divide (20,4)

There are two parameters, so the default value for b (int b=2) is ignored and b takes the value passed as argument, that is 4, making the result returned equal to 5 (20/4).

Inline Functions

Function call is a costly operation. During the function call it's execution our system takes overheads like: Saving the values of registers, Saving the return address, Pushing arguments in the stack, Jumping to the called function, Loading registers with new values, Returning to the calling function, and reloading the registers with previously stored values. For large functions this overhead is negligible but for small function taking such large overhead is not justifiable. To solve this problem concept of inline function is introduced in c++.

The functions which are expanded inline by the compiler each time it's call is appeared instead of jumping to the called function as usual is called inline function. This does not change the behavior of a function itself, but is used to suggest to the compiler that the code generated by the function body is inserted at each point the function is called, instead of being inserted only once and perform a regular call to it, which generally involves some additional overhead in running time.

Example:

```
#include <iostream.h>
#include<conio.h>
inline void sum(int a int b)
{
        int s;
        s= a+b;
        cout<<"Sum="<<s<<endl;
}
```

```
Void main()
{
        clrscr();
        int x, y;
        cout<<"Enter two numbers"<<endl;
        cin>>x>>y
        sum(x,y);
        getch();
}
```

Here at the time of function call instead of jumping to the called function, function call statement is replaced by the body of the function. So there is no function call overhead.

Overloaded functions

In C++ two different functions can have the same name if their parameter types or number are different. That means that you can give the same name to more than one function if they have either a different number of parameters or different types in their parameters. This is called function overloading. For example:

```
// overloaded function
#include <iostream>
int mul (int a, int b)
{
    return (a*b);
}
float mul (float a, float b)
{
    return (a*b);
}
int main ()
{
    int x=5,y=2;
```

```
    float n=5.0,m=2.0;
    cout << mul (x,y);
    cout << "\n";
    cout << mul(n,m);
    cout << "\n";
    return 0;
}
```

In the first call to “mul” the two arguments passed are of type int, therefore, the function with the first prototype is called; This function returns the result of multiplying both parameters. While the second call passes two arguments of type float, so the function with the second prototype is called. Thus behavior of a call to mul depends on the type of the arguments passed because the function has been overloaded.

Notice that a function cannot be overloaded only by its return type. At least one of its parameters must have a different type.

Arguments passed by value and by reference

In case of pass by value, Copies of the arguments are passed to the function not the variables themselves. For example, suppose that we called our function addition using the following code:

```
// function example
#include <iostream>
int addition (int a, int b)
{
        int r;
        r=a+b;
        return (r);
}
int main ()
{
        int z;
        int x=5, y=3;
```

```
        z = addition (x,y);
        cout << "The result is " << z;
        return 0;
}
```

What we did in this case was to call to function addition passing the values of x and y, i.e. 5 and 3 respectively, but not the variables x and y themselves.

When the function addition is called, the value of its local variables a and b become 5 and 3 respectively, but any modification to either a or b within the function addition will not have any effect in the values of x and y outside it, because variables x and y were not themselves passed to the function, but only copies of their values at the moment the function was called.

In case of pass by reference, Address of the variables (variable itself) not copies of the arguments are passed to the function. For example, suppose that we called our function addition using the following code:

```
// function example
#include <iostream>
int addition (int &a, int &b)
{
        int r;
        r=a+b;
        return (r);
}
int main ()
{
        int z;
        int x=5, y=3;
        z = addition (x,y);
        cout << "The result is " << z;
        return 0;
}
```

What we did in this case was to call to function addition passing the variables x and y themselves (not values 5 and 3) respectively.

When the function addition is called, the value of its local variables a and b points to the same memory location respectively, therefore any modification to either a or b within the function addition will also have effect in the values of x and y outside it.

Return by Reference

If we return address (refrence) rather than value from a function then it is called return by reference.

```
#include<iostream.h>
#include<conio.h>
int& min(int &x, int &y)
{
        if(x<y)
        return x
        else
        return y;
}
void main()
{
        clrscr();
        int a,b,r;
        cout<<"Enter two numbers"<<endl;
        cin>>a>>b;
        min(a,b)=0;
        cout<<"a="<<a<<endl<<"b="<<b;
        getch();
}
```

Here min function return the reference of the variable which is smaller between a and b and the statement "min(a.b)=0" makes the value of the variable zero which is smaller.

Chapter 3

Classes and Objects

Introduction

Object-oriented programming (OOP) encapsulates data (attributes) and functions (behavior) into a single unit called classes. The data components of the class are called data members and the function components are called member functions. The data and functions of a class are intimately tied together. Class is a blueprint of real world objects. A programmer can create any number of objects of the same class. Classes have the property of information hiding. It allows data and functions to be hidden, if necessary, from external use. Classes are also referred to as programmer-defined data types.

Extensions to C Structure made by C ++

C++ has made following three extensions to c structure, which makes the c++ structure more powerful than C structure.

- Allows adding functions as a member of structure

```
class Employee
{

        ……..
        ……..
        void getdata( )
        {

        }
        ……..
        ……..
}
```

- Allows us to use the access specifiers private and public to use inside the class

```
class Employee
{
        public:
        int eid,sal;
        private:
        void getdata( )
        {
                //function body
        }
        ……..
```

```
        ……..
    }
```

- Allows us to use structures similar to that of primitive data types while defin ing variables.

```
        struct Employee e;//C style
        Employee e;    //C++ style
```

Complete Example:

```
class Employee
{
        public:
                int eid,sal;
        private:
                void getdata( )
                {
                        cout<<"Enter ID and Salary of an employee"<<endl;
                        cin>>eid>>sal;
                }
                Void display( )
                {
                      cout<<"Emp ID:"<<eid<<endl<<"Salary:"<<sal<<endl;
                }
};

void main( )
{
        clrscr( );
        Employee e;
        e.getdata( );
        cout<<"Employee Details::::"<<endl;
        e.display( );
        getch(  );
}
```

Think!!!!

What modification in above program is needed if we need to read and display records of 10 employees?

Specifying a Class

The specification starts with the keyword **class** followed by the class name. Like structure, its body is delimited by braces terminated by a semicolon. The body of the

class contains the keywords **private**, **public**, and **protected** (discussed later in inheritance). Private data and functions can only be accessed from within the member functions of that class. Public data or functions, on the other hand are accessible from outside the class. Usually the data within a class is private and functions are public. The data is hidden so it will be safe from accidental manipulation, while the functions that operated on the data are public so they can be accessed from outside the class.

The class also contains any number of data items and member functions. The general form of class declaration is:

```
class class_name
{
        private:
        data-type variable1;
        data-type variable2;
        data-type function1(argument declaration)
        {
                Function body
        }

        data-type function2(argument declaration)
        {
                Function body
        }
        ……
        ……
        public:
        data-type variable3;
        data-type variable4;
        data-type function3(argument declaration)
        {
                Function body
        }
```

```
                    data-type function4(argument declaration)
                    {
                        Function body
                    }
                    ……
                    ……
        };
```

For example, we can declare a class for rectangles as follows:

```
class rectangle
{
    private:
    int length;
    int breadth;
    public:


    void setdata(int l, int b)
    {
        length = l;
        breadth = b;
    }

    void showdata()
    {
        cout<<"Length = "<<length<<endl<<"Breadth = "<<breadth<<endl;
    }

    int findarea()
    {
        return length * breadth;
    }
```

```
        int findperemeter()
        {
                return 2 * length * breadth;
        }
};
```

Creating Objects

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

```
rectangle r;
```

creates a variable (object) r of type rectangle. We can create any number of objects from the same class. Fro example,

```
rectangle r1, r2, r3;
```

Objects can also be created when a class is defined by placing their names immediately after the closing brace. For example,

```
class rectangle {
                ……
                ……
                ……
        }r1, r2, r3;
```

Accessing Class Members

When an object of the class is created then the members are accessed using the '.' dot operator. For example,

```
r.setdata(4, 2);
r.showdata();
cout<<"Area = "<<r.findarea()<<endl;
cout<<"Peremeter = "<<r.findperemeter();
```

Private class members cannot be accessed in this way from outside of the class. For example, if the following statements are written inside the main function, the program generates compiler error.

```
r.length = length;
r1.breadth = breadth
```

A Complete Program

```
#include<iostream.h>
#include<conio.h>
class rectangle
{
        private:
        int length;
        int breadth;
        public:


        void setdata(int l, int b)
        {
                length = l;
                breadth = b;
        }
        void showdata()
        {
                cout<<"Length = "<<length<<endl
                        <<"Breadth = "<<breadth<<endl;
        }
        int findarea()
        {
                return length * breadth;
        }
        int findperemeter()
        {
                return 2 * length * breadth;
```

```
        }
};

void main()
{
        clrscr();
        rectangle r;
        r.setdata(4, 2);
        r.showdata();
        cout<<"Area= "<<r.findarea()<<endl;
        cout<<"Peremeter= "<<r.findperemeter();
        getche();
}
```

Defining Member Functions Outside Of the Class

The member functions that are declared inside a class have to be defined separately outside the class. The general form of this definition is:

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

The symbol :: is called the binary scope resolution operator. For example, we can rewrite the same rectangle class as follows:

```
class rectangle
 {
        private:
        int length;
        int breadth;
        public:
        void setdata(int l, int b);
        void showdata();
```

```
        int findarea();
        int findperemeter();
};
void rectangle :: setdata(int l, int b)
{
        length = l;
        breadth = b;
}
void rectangle :: showdata()
{
        cout<<"Length = "<<length<<endl
           <<"Breadth = "<<breadth<<endl;
}

int rectangle :: findarea()
{
        return length * breadth;
}

int rectangle :: findperemeter()
{
        return 2 * length * breadth;
}
```

Functions defined outside the class are not normally inline. But a function defined inside a class is treated as an inline function. We can define a member function outside the class definition and still make it inline by just using the qualifier **inline** as follows:

```
inline void rectangle :: setdata(int l, int b)
{
        length = l;
        breadth = b;
}
```

Note: The scope resolution operator can also be used to uncover hidden global variable. In this case it is called unary scope resolution operator. The general form is:

:: variable-name

Example:

```
#include<iostream.h>
#include<conio.h>
int m = 4;
void main()
{
        clrscr();
        int m = 2;
        cout<<m<<endl;
        cout<<::m;
        getch();
}
```

Output:

2

4

Memory Allocation for Objects

For each object, the memory space for data members is allocated separately because the data members will hold different data values for different objects. However, all the objects in a given class use the same member functions. Hence, the member functions are created and placed in memory only once when they are defined as a part of a class specification and no separate space is allocated for member functions when objects are created.

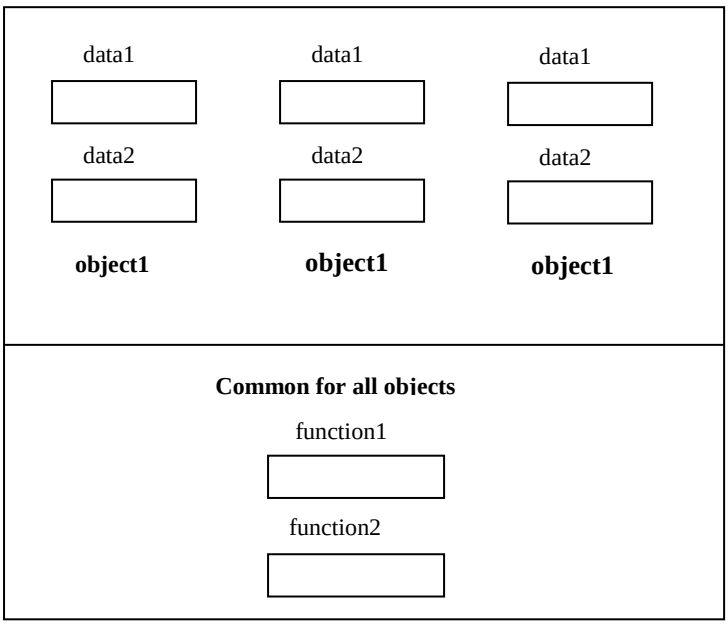


Fig: Objects in Memory

Nesting of Member Functions

A member function can be called by using its name inside another member function of the same class. This is known as nesting of member functions. For example, the class below shows the nesting of member function findinterest inside the member function findtotal.

```
class total
{
        private:
        float principle, time, rate;
        float findinterest()
        {
                return principle * time * rate / 100;
        }
        public:
        void setdata(float p, float t, float r)
        {
                principle = p;
                time = t;
                rate = r;
        }
```

```
        float findtotal()
        {
            return principle + findinterest();
        }
};
```

Remember: Like private data member, some situations may require certain member functions to be hidden from the outside call. In the above example, the member function findinterest is in private block and is hidden from the outside call. A private member function can only be called by another member function that is a member function of its class.

Program::

WAP to add two objects of Complex class

```
class Complex
{
    private:
        int real,img;
    public:
        void getData( )
        {
            cout<<"Enter values od real and imaginary"<<endl;
        }
        void display()
        {
            Cout<<"("<<real<<"+i"<<img<<")"<<endl;
        }
        void addComplex(Complex c1, Complex c2)
        {
```

```
                real=c1.real+c2.real;
                img=c1.img+c2.img;
        }
};
void main( )
{
        clrscr( );
        Complex c1,c2,c3;
        c1.getdata();
        c2.getdata();
        c3.addComplex(c1,c2);
        cout<"C3=";
        c3.display();
        getch();
}
```

Think!!!!!

What changes need to be done above program if we change the function call statement as "c3=c1.addComplex(c2)".

Static Data Members

If a data member in a class is defined as **static**, then only one copy of that member is created for the entire class and is shared by all the objects of that class, no matter how many objects are created. Hence, these data members are normally used to maintain values common to the entire class and are also called class variables.

Example:

```
class rectangle
{
        private:
```

```
        int length;
        int breadth;
        static int count; //static data member

        public:
        void setdata(int l, int b)
        {
                length = l;
                breadth = b;
                count++;
        }

        void displaycount()
        {
                cout<<count<<endl;
        }
};

int rectangle :: count;

        void main()
        {
                clrscr();
                rectangle r1, r2, r3;
                r1.displaycount();
                r2.displaycount();
                r3.displaycount();
                r1.setdata(15, 6);
                r2.setdata(9, 8);
                r3.setdata(12, 9);
                r1.displaycount();
                r2.displaycount();
                r3.displaycount();
                getche();
```

```
}
```

Output:

```
0
0
0
3
3
3
```

In this program, the static variable count is initialized to zero when the objects are created. The count is incremented whenever data is supplied to an object. Since the data is supplied three times, the variable count is incremented three times. Because there is only one copy of the count shared by all the three objects, all the three calls to the displaycount member function cause the value 3 to be displayed.

The type and scope of each static data member must be defined outside the class definition because these data members are stored separately rather than as a part of an object. For example, the following statement in the above program is defined outside the class definition.

```
int rectangle :: count;
```

While defining a static variable, some initial value can also be assigned to the variable. For example, the following definition gives count the initial value 5.

```
int rectangle :: count = 5;
```

Static Member Functions

Like static member variables, we can also have static member functions. A static member function can have access to only other static members (functions or variables) declared in the same class and can be called using the class name (instead of objects) as follows:

```
class-name :: function-name;
```

Example:

```
class rectangle
```

```
{
        private:
        int length;
        int breadth;
        static int count;
        public:
        void setdata(int l, int b)
        {
                length = l;
                breadth = b;
                count++;
        }
        static void displaycount()
        {
                cout<<count<<endl;
        }
};

int rectangle :: count;

void main()
{
        clrscr();
        rectangle r1, r2, r3;
        rectangle :: displaycount();
        r1.setdata(6, 5);
        r2.setdata(8, 4);
        r3.setdata(15, 7);
        rectangle :: displaycount();
        getche();
}
```

Output:

0

3

Another Complete Program

This program adds two distance objects each having two private data members feet (type int) and inches (type float).

```
#include<iostream.h>
#include<conio.h>
        class distance
        {
                private:
                int feet;
                float inches;
                public:
                void setdata(int f,float i)
                {
                        feet=f;
                        inches=i;
                }

                distance adddistance(distance d2)
                 {
                        distance d3;
                        d3.feet = feet + d2.feet;
                        d3.inches = inches + d2.inches;
                        d3.feet=d3.feet + d3.inches/12;
                        d3,inches=d3.inches%12;
                        return d3;
                }

                void display()
                {
                        cout<<"("<<feet<<", "<<inches<<")"<<endl;
                }
        };
```

```
void main()
{
        clrscr();
        distance d1, d2, d3;
        d1.setdata(5, 6.5);
        d2.setdata(7, 8.7);
        d3 = d1.adddistance(d2);
        cout<<"d1 = "; d1.display();
        cout<<"d2 = "; d2.display();
        cout<<"d3 = "; d3.display();
        getche();
}
```

Output:
d1 = (5, 6.5)
d2 = (7, 8.7)
d3 = (13, 3.2)

Objects as Function Arguments

Like any other data type, an object may be used as a function argument in three ways: pass-by-value, pass-by-reference, and pass-by-pointer.

1. Pass-by-value: In this method, a copy of the object is passed to the function. Any changes made to the object inside the function do not affect the object used in the function call. For example,

```
distance adddistance(distance d)
{
        distance dd;
        dd.feet = feet + d.feet;
        dd.inches = inches + d.inches;
        dd.feet=dd.feet +dd.inches/12
        dd.inches=dd.inches%12;
        return dd;
}
```

2. Pass-by-reference: In this method, an address of the object is passed to the function. The function works directly on the actual object used in the function call.

This means that any changes made to the object inside the function will reflect in the actual object. For example,

```
distance adddistance(distance& d2)
{
        distance d3;
        d3.feet = feet + d2.feet;
        d3.inches = inches + d2.inches;
        d3.feet=dd.feet +dd.inches/12
        d3.inches=dd.inches%12;
        return d3;
}
```

3. Pass-by-pointer: Like pass-by-reference method, pass-by-pointer method can also be used to work directly on the actual object used in the function call. For example,

```
distance adddistance(distance* d2)
{
        distance d3;
        d3.feet = feet + d2->feet;
        d3.inches = inches + d2->inches;
        d3.feet=dd.feet +dd.inches/12
        d3.inches=dd.inches%12;
        return d3;
}
```

This function must be called as follows:

```
d3 = d1.adddistance(&d2);
```

Returning Objects

A function cannot only receive objects as arguments but also can return them. A function can return objects by value, by reference, and by pointer.

1. Return-by-value: In this method, a copy of the object is returned to the function call. For example, in the following function d3 is returned by value.

```
        distance adddistance(distance d2)
```

```
{
        distance d3;
        d3.feet = feet + d2.feet;
        d3.inches = inches + d2.inches;
        d3.feet=dd.feet +dd.inches/12
        d3.inches=dd.inches%12;
        return d3;
}
```

2. Return-by-reference: In this method, an address of the object is returned to the function call. We cannot return automatic variables by reference. For example, in the following function d3 is not an automatic variable and is returned by reference.

```
distance& adddistance(distance d2, distance& d3)
 {
        d3.feet = feet + d2.feet;
        d3.inches = inches + d2.inches;
        d3.feet=d3.feet +d3.inches/12
        d3.inches=d3.inches%12;
        return d3;
}
```

This function must be called as follows:

```
d3 = d1.adddistance(d2, d3);
```

3. Return-by-pointer: Like return-by-reference method, return-by-pointer returns address of the object to the function call. For example, in the following function d3 is returned by pointer.

```
distance* adddistance(distance d2)
{
        distance* d3;
        d3->feet = feet + d2.feet;
        d3->inches = inches + d2.inches;
        d3.feet=dd.feet +dd.inches/12
```

```
        d3.inches=dd.inches%12;
        return d3;
    }
```

In the main function, d3 must be declared as a pointer variable and the members of d3 must be accessed as follows:

```
distance d1, d2, *d3;
cout<<"d3 = "; d3->display();
```

Note: If we declare an object as pointer variable, we must access the members of that object by using member selection via pointer (–>) operator.

const (Constant) Object and const Member Functions

Some objects need to be modified and some do not. We can use the keyword **const** to specify that an object is not modifiable and that any attempt to modify the object should result compiler error. The statement

```
const distance d1(5, 6.7);
```

declares a **const** object **d1** of class **distance** and initializes it to 5 feet and 6.7 inches. These objects must be initialized. A **const** object can only invoke a **const** member function. If a member function does not alter any data in the class, then we may declare and define it as a **const** member function as follows:

```
    void display()const
    {
        cout<<"("<<feet<<", "<<inches<<")"<<endl;
    }
```

The qualifier **const** is inserted after the function's parameter list in both declarations and definitions. The compiler will generate an error message if such functions try to alter the data values. A **const** member function cannot call a non-**const** member function on the same class.

Access Modifiers

Access modifiers are constructs that defines the scope and visibility of members of a class. There are three access modifiers:

- Private

  Private members are accessible only inside the class and not from any other location outside of the class.
- Protected

  Protected members are accessible from the class as well as from the child class but not from any other location outside of the class.
- Public

  Public members are accessible from any location of the program.

```
class Test
{
        private:
                int x;
        public:
                int y;
        void getdata()
        {
                cout<<"Enter x and y"<<endl;
                cin>>x>>y;
        }
        void display()
        {
                cout<"x="<<x<<"y="<<y<<endl;
        }
}



void main()
{
        clrscr();
        Test p;
        p.getdata();
        cout<<"Enter  value of x"<<endl;
        cin>>p.x;
```

```
        cout<<"Enter value of y"<<endl;
        cin>>p.y;
        getch();
    }
```

Here the statement "cin>>p.x;" generates error because x is private data member because, x is private data member and is not accessible from outside of the class. But the statement "cin>>p.y;" does not generates error because y is public data member and is accessible from everywhere. Thus we can say that use of private access specifier is used to achieve data hiding in class.

Friend Functions

The concepts of data hiding and encapsulation dictate that private members of a class cannot be accessed from outside the class, that is, non-member functions of a class cannot access the non-public members (data members and member functions) of a class. However, we can achieve this by using friend functions. To make an outside function friendly to a class, we simply declare the function as a **friend** of the class as shown below:

```
class sample
{
        int a;
        int b;
    public:
        void setvalue()
        {
            a = 25;
            b = 40;
        }
        friend float mean(sample s);
};
```

```
float mean(sample s)
{
        return float(s.a + s.b)/2;
}

void main()
{
        clrscr();
        sample x;
        x.setvalue();
        cout<<"Mean value = "<<mean(x);
        getch();
}
```

Output:

Mean value = 32.5

Some special characteristics of friend functions are:

1. Since, it is not in the scope of the class to which it has been declared as a friend, it cannot be called using the object of the class.
2. It can be invoked like a normal function without the help of any object.
3. Unlike member functions, it cannot access the member names directly and has to use an object name and dot membership operator with each member name.
4. It can be declared either in the public or the private part of a class without affecting its meaning.
5. Usually, it has the objects as arguments.

Furthermore, a fried function also acts as a bridging between two classes. For example, if we want a function to take objects of two classes as arguments and operate on their private members, we can inherit the two classes from the same base class and put the function in the base class. But if the classes are unrelated, there is nothing like a friend function. For example,

```
class beta;
class alpha
{
        private:
                int data;
        public:
                void setdata(int d)
                {
                        data = d;
                }
                friend int frifunc(alpha, beta);
};

class beta
{
        private:
                int data;
        public:
                void setdata(int d)
                {
                        data = d;
                }
                friend int frifunc(alpha, beta);
};

int frifunc(alpha a, beta b)
{
        return a.data + b.data;
}

void main()
{
        clrscr();
        alpha aa;
```

```
    aa.setdata(7);
    beta bb;
    bb.setdata(3);
    cout<<frifunc(aa, bb);
    getch();
}
```

Output:

10

Friend Classes

The member functions of a class can all be made friends of another class when we make the former entire class a **friend** of later. For example,

```
class alpha
{
    private:
        int x;
    public:
        void setdata(int d)
        {
            x = d;
        }
        friend class beta;
};

class beta
{
    public:
        void func(alpha a)
        {
            cout<<a.x<<endl;
        }
};
```

```
void main()
{
        clrscr();
        alpha a;
        a.setdata(99);
        beta b;
        b.func(a);
        getch();
}
```

Output:

99

In the above program, in class **alpha** the entire class **beta** is declared as **friend**. Hence, all the member functions of **beta** can access the private data of **alpha**. Alternatively, we can declare **beta** to be a class before the **alpha** class specifier as follows and then, within alpha, we can referred to beta without the class keyword as follows:

```
class beta;
class alpha
{
        private:
                int x;
        public:
                void setdata(int d)
                {
                        x = d;
                }
                friend beta;
};
```

Chapter 4
Constructors and Destructors

Constructors

A constructor is a special member function that is executed automatically whenever an object is created. It is used for automatic initialization. Automatic initialization is the process of initializing object's data members when it is first created, without making a separate call to a member function. The name of the constructor is same as the class name. For example,

```
class rectangle
{
        private:
                int length;
                int breadth;
        public:
                rectangle()
                {          //constructor
                        length = 0;
                        breadth = 0;
                }
                ……
                ……
};
```

Some special characteristics:

Constructors have some special characteristics. These are:

1. Constructors should be defined or declared in the public section.
2. They do not have return types.
3. They cannot be inherited but a derived class can call the base class constructor.
4. Like functions, they can have default arguments.
5. Constructors cannot be **virtual**.
6. We cannot refer to their addresses.

7. An object with a constructor (or destructor) cannot be used as a member of a union.
8. They make 'implicit calls' to the new and delete operators when a memory allocation is required.

Types Of constructor

Default Constructor

The constructor used in this example is **default constructor**. A constructor that accepts no parameters is called default constructor. If a class does not include any constructor, the compiler supplies a default constructor. If we create an object by using the declaration

rectangle r1;

Default constructor is invoked

Parameterized Constructors

Unlike a default constructor, a constructor may have arguments. The constructors that take arguments are called parameterized constructors. For example, the constructor used in the following example is the parameterized constructor and takes two arguments both of type int.

```
class rectangle
{
        private:
                int length;
                int breadth;
        public:
        rectangle(int l, int b)
        {       //parameterized constructor
                length = l;
                breadth = b;
        }
                ……
                ……
```

};

We can use parameterized constructors in two ways: by calling the constructor explicitly and by calling the constructor implicitly (sometimes called shorthand method). The declaration

rectangle r1 = rectangle(5, 6.7);

illustrates the first method of calling and the declaration

rectangle r1(5, 6.7);

illustrates the second method of calling.

Remember: If a class contains parameterized constructor(s), we must supply default constructor explicitly to use it.

Copy Constructor

A copy constructor is used to declare and initialize an object with another object of the same type. For example, the statement

rectangle r2(r1);

Creates new object r2 and performs member-by-member copy of r1 into r2. Another form of this statement is

rectangle r2 = r1;

The process of initializing through assignment operator is known as copy initialization. A copy constructor takes reference to an object of the same class as its argument. For example,

```
rectangle(rectangle& r)
{
        lentht = r.length;
        breadth = r.breadth;
}
```

Remember: We cannot pass the argument by value to a copy constructor.

Constructor Overloading

We can define more than one constructor in a class either with different number of arguments or with different type of argument which is called constructor overloading.

Example:

```
class Item
{
        int code, price;
    public:

        Item()
        {//Default Constructor
            code= price =0;
        }
        Item(int c,int p)
        {//Parameterized Constructor
            code=c;
            price=p;
        }
        Item(Item &x)
        {//Copy Constructor
            code=x.code;
            price= x.price;
        }
        void display()
        {
            Cout<<"Code::"<<code<<endl<<"Price::"<<price<<endl;
        }
};

void main()
{
    clrscr();
    Item I1;
    Item I2(102,300);
    Item I3(I2);
    I1.display();
    I2.display();
    I3.display();
    getch();
}
```

Thiink!!!
What will be the output of above program

Destructors

A destructor is a special member function that is executed automatically just before lifetime of an object is finished. A destructor has the same name as the constructor (which is the same as the class name) but is preceded by a tilde (~). Like constructors, destructors do not have a return value. They also take no arguments. Hence, we can use only one destructor in a class.

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

```
Class Test
{
        private:
                int x,y;
        public:
                Test()
                {
                        cout<<"Memory is allocated"<<endl;
                }
                ~Test()
                {
                        cout<<"Memory is deallocated"<<endl;
                }
}
void main()
{
        clrscr();

        {
                Test p;
        }//life time of p finishes here, and destructor is called
        getch();
}
```

Output:

Memory is allocated

Memory is de-allocated

Whenever **new** operator is used to dynamically allocate memory in the constructors, we should use **delete** operator to free that memory. For example,

```
class string
{
        private:
                char* name;
        public:
                string(char* n)
                {
                        int length = strlen(n);
                        name = new char[length + 1];
                        strcpy(name, n);
                }
                ~string()
                {    //destructor
                        delete[] name;
                }
                ……
                ……
};
```

Note: The objects are destroyed in the reverse order of creation.

Chapter 5
Operator Overloading

Introduction

Operator overloading is one of the most exciting features of C++. The term operator overloading refers to giving the normal C++ operators additional meaning so that we can use them with user-defined data types. For example, C++ allows us to add two variables of user-defined types with the same syntax that is applied to the basic type. We can overload all the C++ operators except the following:

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

Although the semantics of an operator can be extended, we cannot change its syntax and semantics that govern its use such as the number of operands, precedence, and associativity.

Operator overloading is done with the help of a special function, called operator function. The general form of an operator function is:

```
return-type operator op(arguments)
{
        Function body
}
```

Where return-type is the type returned by the operation, operator is the keyword, and op is the operator symbol. For example, to add two objects of type distance each having data members feet of type int and inches of type float, we can overload + operator as follows:

```
distance operator +(distance d2)
{
        //function body
}
```

And we can call this operator function with the same syntax that is applied to its basic types as follows:

```
d3 = d1 + d2;
```

Note: Operator functions must be either member functions or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no argument 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.

Remember: We cannot use friend functions to overload certain operators. These operators are:

=	assignment operator	( )	function call operator
[]	subscripting operator	–>	class member access operator

Overloading Unary Operators

Let us consider the increment (++) operator. This operator increases the value of an operand by 1 when applied to a basic data item. This operator should increase the value of each data member when applied to an object. For example,

```
class rectangle
{
    private:
        int length;
        int breadth;
    public:
        rectangle(int l, int b)
        {
            length = l;
            breadth = b;
        }
        void operator ++()
        {
            ++length;
```

```
                ++breadth;
        }
        void display()
         {
                cout<<"Length = "<<length<<endl
                   <<"Breadth = "<<breadth;
        }
};

void main()
{
        clrscr();
        rectangle r1(5, 6);
        ++r1;     //r1.operator ++();
        r1.display();
        getch();
}
```

Output:

Length = 6

Breadth = 7

In this example, we used prefix notation. We can also use postfix notation as follows:

```
void operator ++(int)
{
        length++;
        breadth++;
}
```

This int isn't really an argument, and it doesn't mean integer. It is simply a signal to the compiler to create the postfix version of the operator. We can call this operator function as follows:

```
r1++;
```

It is also possible to overload unary operators using a friend function as follows:

```
class rectangle
{
        private:
                int length;
                int breadth;
        public:
                rectangle(int l, int b)
                {
                        length = l;
                        breadth = b;
                }
                void operator ++(rectangle&);

                void display()
                 {
                        cout<<"Length = "<<length<<endl
                           <<"Breadth = "<<breadth;
                }
};
void operator ++(rectangle& r)
                {
                        ++r.length;
                        ++r.breadth;
                }

void main()
{
        clrscr();
        rectangle r1(5, 6);
        ++r1;     //r1.operator ++();
        r1.display();
        getch();
```

```
}
```

Output:

Length = 6

Breadth = 7

Overloading Binary Operators

Let us consider the addition (+) operator. This operator adds the values of two operands when applied to a basic data item. This operator should add the values of corresponding data members when applied to two objects. For example,

```
class distance
{
        private:
                int feet;
                float inches;
        public:
                void getdata( )
                {
                        cout<<"Enter feet and inch"<<endl;
                        cin>>feet>>inches;
                }
                distance operator +( distance d2)
                {
                        distance d3;
                        d3.feet = feet + d2.feet;
                        d3.inches = inches + d2.inches;
                        d3.feet=d3.feet +d3.inches/12
                        d3.inches=d3.inches%12;
                        return d3;
                }

                void display()
                {
```

```
                cout<<"("<<feet<<", "<<inches<<")"<<endl;
        }

};

void main()
{
    clrscr();
    distance d1(5, 6.5), d2(7, 8.7), d3;
    d3 = d1 + d2;     //d1.operator +(d2);
    cout<<"d1 = "; d1.display();
    cout<<"d2 = "; d2.display();
    cout<<"d3 = "; d3.display();
    getche();
}
```

Output:

d1 = (5, 6.5)

d2 = (7, 8.7)

d3 = (13, 3.2)

It is also possible to overload binary operators using a friend function as follows:

```
class distance
{
    private:
        int feet;
        float inches;
    public:
        void getdata( )
        {
            cout<<”Enter feet and inch”<<endl;
            cin>>feet>>inches;
        }
```

```
                friend distance operator +(distance, distance);
                void display()
                {
                        cout<<"("<<feet<<", "<<inches<<")"<<endl;
                }

};
friend distance operator +( distance d2)
{
                distance d3;
                d3.feet = feet + d2.feet;
                d3.inches = inches + d2.inches;
                d3.feet=d3.feet +d3.inches/12
                d3.inches=d3.inches%12;
                return d3;
}

void main()
{
        clrscr();
        distance d1(5, 6.5), d2(7, 8.7), d3;
        d3 = d1 + d2;     //d1.operator +(d2);
        cout<<"d1 = "; d1.display();
        cout<<"d2 = "; d2.display();
        cout<<"d3 = "; d3.display();
        getche();
}
```

Output:

d1 = (5, 6.5)

d2 = (7, 8.7)

d3 = (13, 3.2)

Nameless Temporary Objects: We can also use nameless temporary objects to add two distance objects whose purpose is to provide a return value for the function. For example,

```
distance operator + ( distance d2)
{
        int ft = feet + d2.feet;
        int  in = inches + d2.inches;
        ft=ft+in/12;
        in=int%12;
        return distance(ft, in);     //an unnamed temporary object
}

//Overload + operator to concatenate two strings
        #include<iostream.h>
        #include<string.h>
        #include<conio.h>
        class String
        {
                char s[20];
                public:
                void getdata()
                {
                        cout<<"Enter a string"<<endl;
                        cin>>s;
                }
                void display()
                {
                        cout<"s="<<s<<endl;
                }
                String operator(String t)
                {
                        String temp;
                        strcpy( temp.s,s);
```

```
                strcat(temp.s, t.s);
                return temp;
        }
};



void main()
{
        clrscr();
        String s1,s2,s3;
        s1.getdata();
        s2.getdata();
        s3=s1+s2;
        s3.display();
        getch();
}

//Overload the < operator to compare two Objects of Time Class
        class Time
        {
                int hr,min;
                public:
                void getdata()
                {
                        cout<<"Enter hour and minute"<<endl;
                        cin>>hr>>min;
                }
                void display()
                {
                        cout<<hr<<" hr "<<min<<" min "<<endl;
                }
                int operator(Time t)
                {
```

```
                Time temp;
                if(hr<t.hr)
                return 1;
                else if(hr == t.hr && min< t.min)
                return 1;
                else
                return 0;
        }
};

void main()
{
        clrscr();
        Time t1,t2,t3;
        t1.getdata();
        t2.getdata();
        if(t1<t2)
        {
                cout<<"t1 is smaller"<<endl;
        }
        else
        {
                cout<<"t2 is smaller"<<endl;
        }
        getch();
}
```

Exercises

1. Write a program to overload unary minus (-) operator to invert sign of data members of a distance object.
2. Write a program to compare two distance objects using < and > operators.
3. Write a program to add two complex numbers using + operator.
4. Write a program to concatenate two strings using + operator.
5. Write a program to compare two strings using = = operator.

Type Conversion

The type conversions are automatic as long as the data types involved are built-in types. If the data types are user defined, the compiler does not support automatic type conversion and therefore, we must design the conversion routines by ourselves. Three types of situations might arise in the data conversion in this case.

1. Conversion from basic type to class type
2. Conversion from class type to basic type
3. Conversion from one class type to another class type

Conversion from basic type to class type

In this case, it is necessary to use the constructor. The constructor in this case takes single argument whose type is to be converted. For example,

```
class distance
{
    private:
        int feet;
        int inch;
    public:
        distance(int f,int i)
        {
            feet=f;
            inch=i;
        }
        distance(float m)
        {
            feet = int(m);
            inch = 12 * (m - feet);
        }
        void display()
        {
            cout<<"Feet = "<<feet<<endl <<"Inch = "<<inch;
        }
```

```
};

void main()
{
    clrscr();
    float f = 2.6;
    distance d = f;
    d.display();
    getch();
}
```

Output:

Feet = 2

Inches = 7

Conversion from class type to basic type

In this case, it is necessary to overload casting operator. To do this, we use conversion function.

Example:

```
class distance
{
    private:
            int feet;
            int inch;
    public:
            distance(int f,int i)
            {
                    feet=f;
                    inch=i;
            }

            operator float()
            {
                    float a= feet + inches/12.0;
```

```
                return a;
        }
};

void main()
{
    clrscr();
    distance d(8, 6);
    float x = (float)d;
    cout<<"x = "<<x;
    getch();
}
```

Output:

x = 8.5

Conversion from one class type to another class type

This type of conversion can be carried out either by a constructor or an operator function. It depends upon where we want the routine to be located – in the source class or in the destination class.

a. **Function in the source class:** In this case, it is necessary that the operator function be placed in the source class. For example,

```
class distance
 {
        int feet;
        int  inch;
        public:
                distance(int f, int i)
                {
                        feet = f;
                        inch = i;
                }

                void display()
```

```
        {
            cout<<"Feet = "<<feet<<endl<<"Inches = "<<inches;
        }
};

class dist {
    int meter;
    int centimeter;
    public:
        dist(int m, int c)
        {
            meter = m;
            centimeter = c;
        }

        operator distance( )
         {
            distance d;
            int f,i;
            f= meter*3.3.;
            i=centimeter*0.4;
            f=f+i/12;
            i=i%12;
            return distance(f,i);
        }
};

void main()
{
    clrscr();
    distance d1;
    dist d2(4,50);
    d1=d2;
    d1.display();
```

```
        getche();
}
```

b. Function in the destination class: In this case, it is necessary that the constructor be placed in the destination class. This constructor is a single argument constructor and serves as an instruction for converting the argument's type to the class type of which it is a member. For example,

```
class distance
{
        int meter;
        float centimeter;
        public:
                distance(int m, int c)
                {
                        meter = m;
                        centimeter = c;
                }

                int getmeter()
                {
                        return meter;
                }

                float getcentimeter()
                {
                        return centimeters;
                }
};

class dist
{
        int feet;
        int inch;
```

```
        public:
                dist(int f, int i)
                {
                        feet = f;
                        inch = i;
                }

                dist(distance d)
                {
                        int m,c;
                        m=d.getmeter();
                        c=d.getcentimeter();
                        feet= m*3.3;
                        inch= c*0.4;
                        feet=feet+inch/12;
                        inch= inch%12;

                }
                void display()
                {
                        cout<<"Feet = "<<feet<<endl<<"Inches = "<<inches;
                }
};

void main( )
{
        clrscr();
        distance d1(6,40);
        dist d2;
        d2=d1;
        d2.display();
        getche();
}
```

Chapter 6

Inheritance

Introduction

Inheritance (or derivation) is the process of creating new classes, called **derived classes**, from existing classes, called **base classes**. The derived class inherits all the properties from the base class and can add its own properties as well. The inherited properties may be hidden (if private in the base class) or visible (if public or protected in the base class) in the derived class.

Inheritance uses the concept of code **reusability**. Once a base class is written and debugged, we can reuse the properties of the base class in other classes by using the concept of inheritance. Reusing existing code saves time and money and increases program's reliability. An important result of reusability is the ease of distributing classes. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular situations.

Types of Inheritance

A class can inherit properties from one or more classes and from one or more levels. On the basis of this concept, there are five types of inheritance.

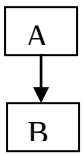
1. Single inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance

Single Inheritance

In single inheritance, a class is derived from only one base class. The example and figure below show this inheritance.

Example

```
class A
{
        members of A
};
class B  :  public A
{
        members of B
};
```

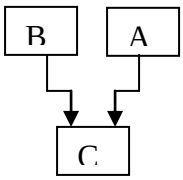


Multiple Inheritance

In this inheritance, a class is derived from more than one base class. The example and figure below show this inheritance.

Implementation Skeleton:

```
class A
{
        members of A
};
class B
{
        members of B
};
class C  :  public A,  public B
{
        members of C
};
```



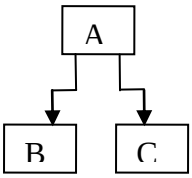
Hierarchical Inheritance

In this type, two or more classes inherit the properties of one base class. The example and figure below show this inheritance.

Implementation Skeleton:

```
class A
```

```
{
        members of A
};
class B
{
        members of B
};
class C  :  public A,  public B
{
        members of C
};
```

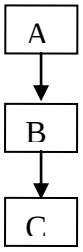


Multilevel Inheritance

The mechanism of deriving a class from another derived class is known as multilevel inheritance. The process can be extended to an arbitrary number of levels. The example and figure below show this inheritance.

Implementation Skeleton:

```
class A
{
        members of A
};
class B : public A
{
        members of B
};
class C  :  public B
{
        members of C
};
```

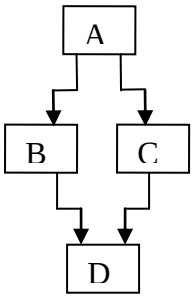


Hybrid Inheritance

This type of inheritance includes more than one type of inheritance mentioned previously. The example and figure below show this inheritance.

Example

```
class A
{
        members of A
};
class B : public A
{
        members of B
};
class C  :  public A
{
        members of C
};
class D : public B, public C
{
        members of D
};
```



Protected Access Specifier

If a class member is **private**, it can be accessed only from within the class where it lies. However, if a class member is **public**, it can be accessed from within the class and from outside the class.

A **protected** member, on the other hand, can be accessed from within the class where it lies and from any class derived from this class. It can't be accessed from outside these classes. The table below summarizes this concept

.

Access Specifier	Accessible from Own Class	Accessible from Derived Class	Accessible from Objects Outside the Class
Public	Yes	yes	yes
protected	Yes	Yes	no
Private	Yes	No	no

Note: Only public and protected members are visible in the derived class.

Public, protected and private derivation

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

class derived-class-name : visibility-mode base-class-name

{

    Members of derived classes

};

The visibility mode is optional. If present, may be private, protected, or public. The default visibility mode is private. For example,

class ABC : [private] [protected] [public] XYZ

{

    members of ABC

};

The visibility mode specifies the visibility of inherited members. If the visibility mode is private, public and protected members of the base class become private members in the derived class and therefore these members can only be accessed in the derived class. They are inaccessible from outside this derived class.

If the visibility mode is protected, both public and protected members of the base class become protected members in the derived class and therefore these members can

only be accessed in the derived class and from any class derived from this class. They are inaccessible from outside these classes.

If the visibility mode is public, public and protected members of the base class do not change. The table below summarizes these concepts:

Base Class Visibility	**Derived Class Visibility**		
	Pubic Derivation	**Private Derivation**	**Protected Derivation**
Private	not visible	not visible	not visible
protected	protected	Private	protected
Public	Public	Private	protected

Note: If we want to disallow the further inheritance of the members of base class from derived class, then private derivation is used.

Derived Class Constructors

When applying inheritance we usually create objects using the derived class. If the base class contains a constructor it can be called from the initializer list in the derived class constructor as follows:

```
class A
{
        protected:
                int adata;
        public:
                A(int a)
                {
                        adata = a;
                }
};
class B :  public A
{
        int bdata;
```

```
        public:
                B(int a, int b) : A(a)
                {
                        bdata = b;
                }
                void showdata()
                {
                        cout<<"adata = "<<adata<<endl <<"bdata = "<<bdata;
                }
};
void main()
{
        clrscr();
        B b(5, 6);
        b.showdata();
        getch();
}
```

Output:

adata = 5

bdata = 6

If the base class contains no constructor, we can write the derived class constructor as follows:

```
B(int a, int b)
{
        adata = a;
        bdata = b;
}
```

In case of multiple inheritance, constructors in the base classes are placed in the initializer list in the derived class constructor separated by commas. For example,

```
class A
```

```
{
        protected:
                int adata;
        public:
                A(int a)
                {
                        adata = a;
                }
};
class B
{
        protected:
                int bdata;
        public:
                B(int b)
                {
                        bdata = b;
                }
};
class C: public A,  public B
{
        int cdata;
        public:
                C(int a, int b, int c) : A(a), B(b)
                {
                        cdata = c;
                }
};
```

Order of execution of constructors

The base class constructor is executed first and then the constructor in the derived class is executed. In case of multiple inheritance, the base class constructors are

executed in the order in which they appear in the definition of the derived class. Similarly, in a multilevel inheritance, the constructors will be executed in the order of inheritance. Furthermore, the constructors for virtual base classes are invoked before any non-virtual base classes. If there are multiple virtual base classes, they are invoked in the order in which they are declared in the derived class.

Exanple:

```
class A
{
        public:
        A( )
        {
                cout<<"Class A Constructor"<<endl;
        }
};
class B:public A
{
        public:
        B( )
        {
                cout<<"Class B Constructor"<<endl;
        }
};
class C: public B
{
        public:
        C( )
        {
```

```
        cout<<"Class C Constructor"<<endl;
    }
};
```

Output:

Class A constructor

Class B constructor

Class C constructor

Order of execution of constructors

The derived class destructor is executed first and then the destructor in the base class is executed. In case of multiple inheritances, the derived class destructors are executed in the order in which they appear in the definition of the derived class. Similarly, in a multilevel inheritance, the destructors will be executed in the order of inheritance. Example:

```
class A
{
    public:
    ~A( )
    {
        cout<<"Class A Destructor"<<endl;
    }
};
class B:public A
{
    public:
    ~B( )
    {
```

```
        cout<<"Class B Destructor"<<endl;
    }
};
class C: public B
{
    public:
    ~C( )
    {
        cout<<"Class C Destructor"<<endl;
    }
};
```

Output:

Class C Destructor

Class B Destructor

Class A Destructor

Overriding Member Functions

We can use member functions in a derived class that override those in the base class. In this case, both base and derived class functions have same name, same number of parameters, and similar type of parameters. For example,

```
class A
{
    public:
        void show()
        {
            cout<<"This is class A";
        }
};
class B :  public A
{
    public:
        void show()
        {
            cout<<"This is class B"<<endl;
```

```
        }
};

void main()
{
    clrscr();
    B b;
    b.show();     //invokes the member function from class B
    b.A :: show();     //invokes the member function from class A
    getche();
}
```

Output:

This is class B

This is class A

Ambiguity in Multiple Inheritance

Suppose two base classes have an exactly similar member. Also, suppose a class derived from both of these base classes has not this member. Then, if we try to access this member from the objects of the derived class, it will be ambiguous. For example,

```
class A
{
    public:
        void show()
        {
            cout<<"This is class A";
        }
};
class B
{
    public:
        void show()
        {
```

```
                cout<<"This is class B"<<endl;
        }
};
class C : public A, public B
{

};
void main() {
    clrscr();
    C c;
    c.show();  //ambiguous – will not compile
    c.A :: show();  //OK
    c.B :: show();  //OK
    getche();
}
```

We can also remove this ambiguity by adding a function in class *C* as follows:

```
class C : public A, public B
{
    public:
        void show()
        {
            A :: show();
        }
};
```

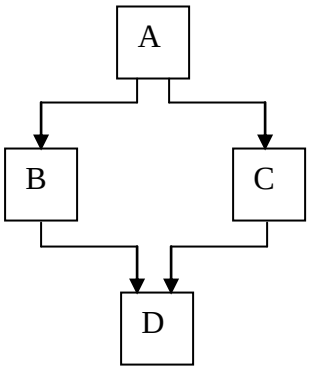
Another kind of ambiguity arises if you derive a class from two classes that are each derived from the same class. We can remove this kind of ambiguity by using the concept of virtual base classes.

Virtual Base Classes

Consider a situation where we derive a class **D** from two classes **B** and **C** that are each derived from the same class **A**. This creates a diamond-shaped inheritance tree(called hybrid multipath inheritance) and all the public and protected members from class **A** inherited into class **D** twice once through the path A→B→D and gain through the path A→B→C . This causes ambiguity and should be avoided.

We can remove this kind of ambiguity by using the concept of virtual base class. For this we make direct base classes (B and C) virtual base classes as follows:

```
class A {
        ……
};
class B : virtual public A {
        ……
};
class C : public virtual A {
        ……
};
class D : public B, public C {
        ……
};
```



In this case, class D inherits only one copy from the classes C and D.

Note: The keywords **virtual** and **public** may be used in either order.

Example:

```
class A
{
        protected:
                int adata;
};
class B : virtual public A
```

```
{


};
class C : public virtual A
{
};
class D : public B, public C
{
        public:
                D(int a) {
                        adata = a;
                }
                int getdata() {
                        return adata;
                }
};
void main() {
        clrscr();
        D d(5);
        cout<<d.getdata();
        getch();
}
```

Output:

5

Containership: (Aggregation)

Inheritance is often called a "kind of" relationship. 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 example, we can say that

bulldog is a kind of dog: A bulldog has the characteristics shared by all dogs but has some distinctive characteristics of its own.

There is another type of relationship, called a “has a” relationship, or containership. We say that a bulldog has a large head, meaning that each bulldog includes an instance of a large head. In object oriented programming, has a relationship occurs when one object is contained in another. For example,

```
class Employee
{
        int eid, sal;
        public:
        void getdata()
        {
                cou<<”Enter id and salary of employee”<<endl;
                cin>>eid>>sal;
        }
        void display()
        {
                cout<<”Emp ID:”<<eid<<endl<<”Salary:”<<sal;
        }
};
class Company
{
        int cid, cname;
        Employee e;
        public:
        void getdata()
        {
```

```
            cou<<"Enter id and name of the company:"<<endl;
            cin>>cid>>cname;
            e.getdata();
        }
        void display()
        {
            cout<<"Comp ID:"<<cid<<endl<<"Comp Name:"<<cname;
            e.display();
        }
};

void main()
{
        clrscr();
        Company c;
        c.getdata();
        c.display();
        getch();
}
```

Think: If Company conatins 10 employees what modification is needed in above program?

Local Classes

Classes can also be defined and used inside a function or a block. Such classes are called local classes. For example,

```
void test (int a)
{
        ……
```

```
        ……
        class A
        {
                ……
                ……
        };
        ……
        ……
        A a;  //create object of type A
        ……
}
```

Local classes can use global variables and static variables declared inside the function but cannot use automatic local variables. The global variables should be used with the scope resolution operator.

Chapter 7

Polymorphism

Introduction

Polymorphism means state of having many forms. We have already seen that polymorphism is implemented using the concept of overloaded functions and operators. In this case, polymorphism is called **early binding** or **static binding** or **static linking**. This is also called **compile time polymorphism** because the compiler knows the information needed to call the function at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself.

There is also another kind of polymorphism called **run time polymorphism**. In this type, the selection of appropriate function is done dynamically at run time. So, this is also called **late binding** or **dynamic binding**. C++ supports a mechanism known as virtual functions to achieve run time polymorphism. Run time polymorphism also requires the use of pointers to objects.

Virtual Functions

Virtual means existing in appearance but not in reality. When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. Furthermore, when we use virtual functions, different functions can be executed by the same function call. The information regarding which function to invoke is determined at run time.

We should use virtual functions and pointers to objects to achieve run time polymorphism. For this, we use functions having same name, same number of parameters, and similar type of parameters in both base and derived classes. The function in the base class is declared as virtual using the keyword virtual. When a function in the base class is made virtual, C++ determines which function to use at run time based on the type of object pointed by the base class pointer, rather than the type of the pointer. For example,

```
class A
```

```
{
        public:
                virtual void show()
                {
                        cout<<"This is class A\n";
                }
};
class B : public A
{
        public:
                void show()
                {
                        cout<<"This is class B\n";
                }
};
class C : public A
{
        public:
                void show()
                {
                        cout<<"This is class C\n";
                }
};
void main() {
        clrscr();
        A *p, a;
        B b;
        C c;
        p = &b;
        a->show();
        p = &c;
        a->show();
        p = &a;
        a->show();
        getch();
```

```
}
```

Output:

This is class B

This is class C

This is class A

If we remove the keyword virtual from the class A, All the time show() function of class is called because at this time function call is made on the basis of type of pointer.

Note: A base class pointer object can point to any type of derived objects but the reverse is not true.

Abstract Classes and Pure Virtual Functions

An abstract class is one that is not used to create objects. It is used only to act as a base class to be inherited by other classes. We can make abstract classes by placing at least one pure virtual function in the base class. A pure virtual function is one with the expression = 0 added to the declaration – that 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. For example,

```
class A
{
        protected:
                int data;
        public:
                A(int d)
                {
                        data = d;
                }
                virtual void show() = 0;
};
class B : public A
{
        public:
```

```
        B(int d) : A(d)
        {
        }
        void show()
        {
                cout<<data<<endl;
        }
};
class C : public A {
    public:
        C(int d) : A(d)
        {
        }
        void show()
        {
                cout<<data;
        }
};
void main()
{
    clrscr();
    A *a;   //The declaration A a; will cause error (Why?)
    B b(5);
    C c(6);
    a = &b;         a->show();
    a = &c;         a->show();
    getch();
}
```

Output:

5

6

Exceptions

Introduction

Exceptions are runtime anomalies or unusual conditions that a program may encounter while executing. Exceptions might include conditions such as division by zero, access to an array outside of its bounds, running out of memory or disk space, not being able to open a file, trying to initialize an object to an impossible value etc.

When a program encounters an exceptional condition, it is important that it is identified and dealt with effectively. C++ provides built-in language features to detect and handle exceptions, which are basically runtime errors.

The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstance" so that appropriate action can be taken. The mechanism suggests a separate error handling code that performs the following tasks:

- Find the problem (**Hit the exception**).
- Inform that an error has occurred (**Throw the exception**).
- Receive the error information (**Catch the exception**).
- Take corrective action (**Handle the exception**).

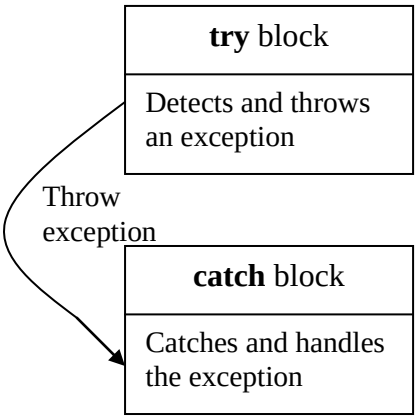
The error handling code basically consists to two segments, one to detect errors and to throw exceptions, and the other to catch the exceptions and to take appropriate actions.

Exception Handling Mechanism

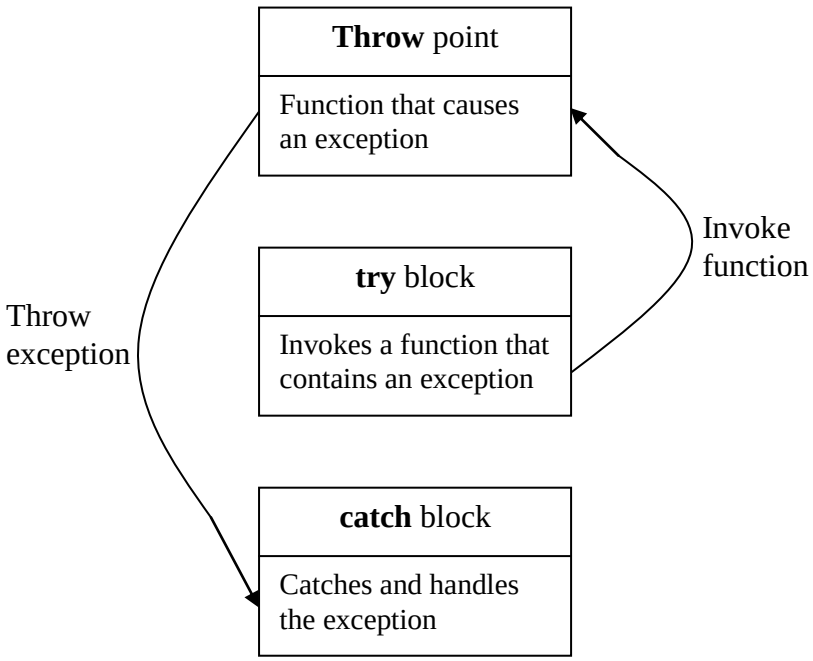
Exception handling mechanism in C++ is basically built upon three keywords: **try**, **throw**, and **catch**. The keyword **try** is used to surround a block of statements, which may generate exceptions. This block of statements is known as try block.

When an exception is detected, it is thrown using the **throw** statement situated either in the try block or in functions that are invoked from within the try block. This is called throwing an exception and the point at which the **throw** is executed is called the throw point.

A catch block defined by the keyword **catch** catches the exception thrown by the **throw** statement and handles it appropriately. This block is also called exception handler. The catch block that catches an exception must immediately follow the try block that throws an exception. The figure below shows the exception handling mechanism if try block throws an exception. The general form in this case is:



The figure below shows the exception handling mechanism if function invoked by try block throws an exception.



Examples

Example1: Try block throwing exception

```
#include<iostream.h>
#include<conio.h>
void main()
{
    clrscr();
    int a, b;
    cout<<"Enter values of a & b:\n";
    cin>>a>>b;
    try
    {
        if(b == 0)
            throw b;
        else
            cout<<"Result = "<<(float)a/b;
    }
     catch(int i)
    {
        cout<<"Divide by zero exception: b = "<<i;
    }
    cout<<"\nEND";
    getch();
}
```

Example2: Function invoked by try block throwing exception

```
#include<iostream.h>
#include<conio.h>
void divide(int a, int b)
{
    if(b == 0)
        throw b;
  else
```

```
        cout<<"Result = "<<(float)a/b;
}
void main()
{
    clrscr();
    int a, b;
    cout<<"Enter values of a & b:\n";
    cin>>a>>b;
     try
    {
        divide(a, b);
    }
    catch(int i)
    {
        cout<<"Divide by zero exception: b = "<<i;
    }
    cout<<"\nEND";
    getche();
}
```

Output:(In both examples)

First Run

Enter the values of a & b:

5

2

Result = 2.5

END

Second Run

Enter the values of a & b:

9

0

Divide by zero exception: b = 0

END

In the first example, if exception occurs in the try block, it is thrown and the program control leaves in the try block and enters the catch block. In the second example, try block invokes the function divide. If exception occurs in this function, it is thrown and control leaves this function and enters the catch block.

Note that, exceptions are used to transmit information about the problem. If the type of exception thrown matches the argument type in the **catch** statement, then only catch block is executed for handling the exception. After that, control goes to the statement immediately after the catch block. If they do not match, the program is aborted with the help of **abort()** function, which is invoked by default. In this case, statements following the catch block are not executed. When no exception is detected and thrown, the control goes to the statement immediately after the catch block skipping the catch block.

Throwing Mechanism

When an exception is detected, it is thrown using the **throw** statement in one of the following forms:

```
throw(exception);
throw;  //used for rethrowing an exception (discussed later)
```

The operand exception may be of any type (built-in and user-defined), including constants. When an exception is thrown, the catch statement associated with the try block will catch it. That is, the control exits the current try block, and is transferred to the catch block after the try block. Throw point can be in a deeply nested scope within a try block or in a deeply nested function call. In any case, control is transferred to the catch statement.

Catching Mechanism

Code for handling exceptions is included in catch blocks. The general form of catch block is:

```
catch(type arg)
{
    Body of catch block
```

}

The type indicates the type of exception that catch block handles. The parameter arg is optional. If it is named, it can be used in the exception handling code. The catch statement catches an exception whose type matches with the type of catch argument. When it is caught, the code in the catch block is executed. After its execution, the control goes to the statement immediately following the catch block. If an exception is not caught, abnormal program termination will occur. The catch block is simply skipped if the catch statement does not catch an exception.

⇒ **Multiple Catch Statements:** It is possible that a program segment has more than one condition to throw an exception. In such cases, we can associate more than one **catch** statement with a **try** as shown below:

```
try
{
    Try block
}

catch(type1 arg)
{
    Catch block1
}
catch(type2 arg)
{
    Catch block 2
}
……
……
}
catch(typeN arg)
{
    Catch block N
}
```

When an exception is thrown, the exception handlers are searched in order for an appropriate match. The first handler that yields a match is executed. After that, the control goes to the first statement after the last **catch** block for that **try** skipping other exception handlers. When no match is found, the program is terminated.

Note: It is possible that arguments of several catch statements match the type of an exception. In such cases, the first handler that matches the exception type is executed.

Example:

```
#include<iostream.h>
#include<conio.h>
void test(int x)
{
    try
    {
            if(x == 0) throw x;
            if(x == 1) throw 1.0;
     }

catch(int m)
{
            cout<<"Caught an integer\n";
  }
catch(double d)
{
            cout<<"Caught a double";
 }
}

void main()

{
  clrscr();
  test(0);
  test(1);
```

```
    test(2);
    getche();
}
```

Output:

Caught an integer

Caught a double

⇒ **Catch All Exceptions:** If we want to catch all possible types of exceptions in a single **catch** block, we use **catch** in the following way:

```
catch(…)
{
    Statements for processing all exceptions
}
```

Example:

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



void test(int x)
{
    try
    {
            if(x == 0) throw x;
            if(x == 1) throw 1.0;
     }
    catch(…)
    {
            cout<<"Caught an exception\n";
     }
}
void main()
```

```
{
  clrscr();
  test(0);
  test(1);
  test(2);
  getche();
}
```

Output:

Caught an exception

Caught an exception

Rethrowing an Exception

A handler may decide to rethrow the exception caught without processing it. In such situations, we may simply invoke throw without any arguments as shown below:

```
throw;
```

This causes the current exception to be thrown to the next enclosing **try/catch** sequence and is caught by a **catch** statement listed after that enclosing **try** block. For example,

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

void divide(int a, int b)

{
    try
    {
        if(b == 0)
                throw b;
        else
                cout<<"Result = "<<(float)a/b;
  }
  catch(int)
  {
        throw;
```

```
  }
}

void main()

{
        clrscr();
        int a, b;
        cout<<"Enter values of a & b:\n";
        cin>>a>>b;
        try
        {
                divide(a, b);
        }
        catch(int i)
        {
                cout<<"Divide by zero exception: b = "<<i;
        }
        cout<<"\nEND";
        getch();
}
```

Specifying Exceptions

It is also possible to restrict a function to throw only certain specified exceptions. This is achieved by adding a throw list clause to the function definition. The general form is as follows:

```
type function(arg-list) throw (type-list)

{
        Function body
}
```

The type-list specifies the type of exceptions that may be thrown. Throwing any other type of exception will cause abnormal program termination. If we wish to prevent a function from throwing any exception, we may do so by making the type-list empty. Hence the specification in this case will be

```
type function(arg-list) throw ()
{
        Function body
}
```

Note: A function can only be restricted in what types of exception it throws back to the try block that called it. The restriction applies only when throwing an exception out of the function (and not within the function).

Example:

```
#include<iostream.h>
#include<conio.h>
void test(int x) throw (int, double)
{
        if(x == 0) throw x;
         if(x == 1) throw 1.0;
}

void main()
{
  clrscr();
  try
{
         test(1);
}
catch(int m)
{
        cout<<"Caught an integer\n";
 }
catch (double d)
{
        cout<<"Caught a double";
  }
```

```
  getch();
}
```

Output:

Caught a double

Chapter 8

Templates

Function templates

Function templates are special functions that can operate with generic types. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using template parameters. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type. The format for declaring function templates with type parameters is:

```
template <class identifier> function_declaration;
template <typename identifier> function_declaration;
```

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way. For example, to create a template function that returns the greater one of two objects we could use:

```
template <class Type>
myType GetMax (Type a, Type b)
{
    return (a>b?a:b);
}
```

To use this function template we use the following format for the function call:

```
function_name <type> (parameters);
```

For example, to call GetMax to compare two integer values of type int we can write:

```
int x,y;
GetMax <int> (x,y);
```

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Complete Example:

```
// function template
#include <iostream.h>
template <class T>
T GetMax (T a, T b)
{
    T result;
    result = (a>b)? a : b;
    return (result);
}
int main ()
{
    int i=5, j=6, k;
    long l=10, m=5, n;
    k=GetMax<int>(i,j);
    n=GetMax<long>(l,m);
    cout << k << endl;
    cout << n << endl;
    return 0;
}
```

In the example above we used the function template GetMax() twice. The first time with arguments of type int and the second one with arguments of type long. The

compiler has instantiated and then called each time the appropriate version of the function.

As you can see, the type T is used within the GetMax() template function even to declare new objects of that type:

```
T result;
```

Therefore, result will be an object of the same type as the parameters a and b when the function template is instantiated with a specific type.

We can also define function templates that accept more than one type parameter, simply by specifying more template parameters between the angle brackets. For example:

```
template <class T, class U>
T GetMin (T a, U b)
{
return (a<b?a:b);
}
```

In this case, our function template GetMin() accepts two parameters of different types and returns an object of the same type as the first parameter (T) that is passed. For example, after that declaration we could call GetMin() with:

```
int i,j;
long l;
i = GetMin<int,long> (j,l);
```

Class templates

We also have the possibility to write class templates, so that a class can have members that use template parameters as types. For example:

```
template <class T>
class mypair
{
    T values [2];
    public:
```

```
        mypair (T first, T second)
        {
                values[0]=first; values[1]=second;
        }
};
```

The class that we have just defined serves to store two elements of any valid type. For example, if we wanted to declare an object of this class to store two integer values of type int with the values 115 and 36 we would write:

```
mypair<int> myobject (115, 36);
```

This same class would also be used to create an object to store any other type:

```
mypair<double> myfloats (3.0, 2.18);
```

```
// class templates
#include <iostream>
template <class T>
class mypair
{
        T a, b;
        public:

        mypair (T first, T second)
        {
                a=first; b=second;
        }
        T getmax ()
        {
                T retval;
                retval = a>b? a : b;
                return retval;
        }
};

int main ( )
```

```
{
        mypair <int> myobject (100, 75);
        cout << myobject.getmax();
        return 0;
}
```