SUNBEAM

SUNBEAM

Institute of Information Technology

LEARNING INITIATIVE

PreCAT
OOPs using C++

Index

Preface

This book is simply beginner's guide to C++ .

This book is doing syntactical study of C++ and avoids talking about detailed internals of C++ compilers. Learning internals of C++ like virtual tables, virtual base pointers, internal reference implementation, RTTI internals, etc is out of scope of this book, even though I have given hints regarding that at some places.

I cannot promise that this is a complete book on C++ , as the topics like STL is skipped fully, while certain topics like file handling, console IO are discussed only at introductory level to meet design goals. Future editions of this book may expect more focus on these topics.

The book is designed for C–DAC entrance preparation and it covers the topics of Object Oriented Programming concepts as well as C++ language features. Even though design goal for this book is limited, book contents are helpful for all C++ beginners.

Thing that should be made clear is that this book is specific to C++ and assumes that reader is comfortable with fundamentals of C language. To get sufficient knowledge of C, one may prefer to read the most simplified book "Let us C" by "Yashwant Kanetkar" or a standard book "ANSI C Programming" by "Kernighan & Ritchie".

Section I : **Object Oriented Programming concepts**

Chapter 1. Introduction to Object Oriented Programming

Object–oriented programming can trace its roots to the 1960s. As hardware and software became increasingly complex, quality was often compromised. Researchers studied ways to maintain software quality and developed object–oriented programming in part to address common problems by strongly emphasizing discrete, reusable units of programming logic. The methodology focuses on data rather than processes, with programs composed of self–sufficient modules (objects) each containing all the information needed to manipulate its own data structure. This is in contrast to the existing modular programming which had been dominant for many years that focused on the *function* of a module, rather than specifically the data, but equally provided for code reuse, and self–sufficient reusable units of programming logic, enabling collaboration through the use of linked modules (/subroutines). This more conventional approach, which still persists, tends to consider data and behavior separately.

An object–oriented program may thus be viewed as a collection of cooperating *objects*, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects and can be viewed as an independent 'machine' with a distinct role or responsibility. The actions (or "operators") on these objects are closely associated with the object. For example, the data structures tend to carry their own operators around with them (or at least "inherit" them from a similar object or class).

In some ways, the history of programming language theory predates even the development of programming languages themselves. The lambda calculus, developed by Alonzo Church and Stephen Cole Kleene in the 1930s, is considered by some to be the world's first programming language.

In the 1960s, the Simula language was developed by Ole–Johan Dahl and Kristen Nygaard; it is widely considered to be the first example of an object–oriented programming language.

Object–oriented analysis, design and programming

Object–oriented analysis and design (OOAD) is a software engineering approach that models a system as a group of interacting objects. Object–oriented analysis (OOA) applies object–modeling techniques to analyze the functional requirements for a system. Object–oriented design (OOD) elaborates the analysis models to produce implementation specifications. OOA focuses on *what* the system does, OOD on *how* the system does it.

Object–oriented analysis

Object–oriented analysis (OOA) looks at the problem domain, with the aim of producing a conceptual model of the information that exists in the area being analyzed. Analysis models do not consider any implementation constraints that might exist, such as concurrency, distribution, persistence, or how the system is to be built. Implementation constraints are dealt with during object–oriented design (OOD). Analysis is done before the design.

The sources for the analysis can be a written requirements statement, a formal vision document, and interviews with stakeholders or other interested parties. A system may be divided into multiple domains, representing different business, technological, or other areas of interest, each of which are analyzed separately.

The result of object–oriented analysis is a description of *what* the system is functionally required to do, in the form of a conceptual model. That will typically be presented as a set of use cases, one or more UML class diagrams, and a number of interaction diagrams. It may also include some kind of user interface mock–up.

The UML : Drawing Diagrams to describe your applications needs something more than simple application

UML (Unified Modeling Language) is the definitive notation for describing object oriented applications. The UML is just a notation – a series of symbols and diagrams. The set of symbols that comprises the UML is designed to optimize the information conveyed by any diagram that you should draw. The UML defines nine types of diagrams :

UML defines nine types of diagrams : class (package), object, use case, sequence, collaboration, state chart, activity, component and deployment.

Class Diagrams :

The purpose of a class diagram is to depict the classes within a model. In an object oriented application, classes have attributes (member variables), operations (member functions) and relationships with other classes. The UML class diagram can depict all these things quite easily. The fundamental element of the class diagram is an icon that represents a class. This icon is shown in the following Figure.

Class
Attribute (data member)
Operation (member function)

Fig. class diagram– for class

Object–oriented design

The input for object–oriented design is provided by the output of analysis. Object–oriented design (OOD) transforms the conceptual model produced in object–oriented analysis to take account of the constraints imposed by the chosen architecture and any non–functional – technological or environmental – constraints, such as transaction throughput, response time, run–time platform, development environment, or programming language.

The concepts in the analysis model are mapped onto implementation classes and interfaces. The result is a model of the solution domain, a detailed description of *how* the system is to be built.

Object–oriented programming :

Object–oriented programming is concerned with realizing a software design using an Object–oriented programming language. An object–oriented programming language, such as C++ , supports the direct implementation of objects and provides facilities to define object classes.

Object oriented programming needs object oriented analysis and object oriented design.

Object oriented programming structure (OOPS) :

It is a programming methodology to organize complex program into simple programs by using concepts such as abstraction, encapsulation, polymorphism and inheritance.

The languages that support abstraction, encapsulation, polymorphism and inheritance are called object oriented programming languages. Example C++ , Java, C#

Advantages of OOP

- Exploits the expressive power of all object oriented programming languages
- Encourages the reuse of software components
- Leads to the systems that are more resilient to change
- Reduces the development risk

Chapter 2. Pillars of Object Oriented Programming

There are few fundamental features any language must have in order to call the language as Object Oriented Programming language. These features can be called as pillars of Object Oriented Programming. There are four major pillars and three minor pillars of object oriented programming structure.

Major pillars :

- Abstraction
- Encapsulation
- Modularity
- Hierarchy

By major, we mean that a model without any one of these elements is not object oriented.

Minor pillars :

- Typing
- Concurrency
- Persistence

By minor, we mean that each of these elements is a useful, but not essential.

Abstraction :

Getting only essential things and hiding unnecessary details is called abstraction.

- An abstraction denotes essential characteristics of an object relative to perspective of user.
- Abstraction is the selective examination of certain aspects of a problem. The goal of abstraction is to isolate aspects that are important (interface) for some purpose from unimportant aspects (implementation).
- Deciding right set of abstractions for a given domain is central problem in OOD.
- Behavior of an object implies the services or operations provided by it to its clients.
- Abstraction focuses on the outer view of the object which is simply interface or contract seen by the clients. This contract encompasses responsibility of the object.

Abstraction in C++

- It is the public access region of the class.
- Generally, it declares the operations that can be invoked by the clients on object of class.
- Avoid using data members in public region of class, as it may be directly changed by the client possibly in undesired way. It may spoil the state of the object.
- Abstraction is also achieved through Composition. For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to interface with them i.e., send messages to them, receive messages from them.

Kinds of abstractions

- Entity abstraction : An object that represents a useful model of a problem –domain or solution domain.
- Action abstraction : An object that provides a generalized set of operations, all of which performs same kind of function.
- Virtual machine abstraction : An object that groups together operations that are all used by some superior level of control, or operations that all use some junior level set of operations.
- Coincidental abstractions : An object that packages a set of operations that have no relation to each other.

Message Passing

- The accessible operations or methods or functions are also known as messages.
- Invoking a member function on object is also termed as sending message to object.
- The object responds to this message depending on the state of the object by executing corresponding method of object.
- The feature is known as message passing.

Encapsulation

Binding of data and code together is called encapsulation.

Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and encapsulation.

Abstraction and encapsulation are complementary concepts : abstraction focuses upon the observable behavior of an object, where as encapsulation focuses upon the implementation this gives rise to this behavior.

- Implementation comprises the representation of abstraction as well as the mechanisms that achieve desired behavior.
- Encapsulation maintains integrity of object.
- Encapsulation conceals the functional details of a class from objects that send messages to it.
- For example, the Dog class has a bark () method. The code for the bark() method defines exactly how a bark happens.

Information Hiding

Information hiding is the process of hiding all the secrets of an object that do not contribute to its essential characteristics; typically, structure of an object is hidden, as well is implementation of its methods.

- The encapsulation is most often achieved through information hiding, which is a process of hiding all secrets of an object. Typically, structure of the object and implementation of its methods is hidden.

- Information hiding is most of the times confused with encapsulation. But there is a blurred line of difference.
- Encapsulation packages the things but don't talk about the nature of package, whether it is transparent or opaque.
- But information hiding implies the opaque package. It makes the things inaccessible to the client.
- These features cannot be differentiated in every language, e.g. Smalltalk (in which functions are always accessible and data is always hidden)

Modularity :

The act of portioning a program into individual components can reduce its complexity to some degree.

Modularity can be a physical modularity (e.g. divide program into .h, .cpp, .rc files) or logical modularity (e.g. namespace)

- Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.
- Even though class is basic unit of decomposition, complex system may have hundreds of classes. These classes can be packaged into different modules for the ease of developer to keep track of system.
- Modularization consists of dividing program into modules that can be compiled separately, but which are interconnected.
- Developer may choose to package classes in such a way that may be reused conveniently.
- Even many compilers generate object code in segments, one for each module.
- Modularity helps in easier maintenance of a complex software.
- Most of the languages support concepts of module interface and its implementation.
- The C/C++ family of languages keep module interface in header (.h) file while implementation into .cpp file. Thus encapsulation & modularity are parallel.

Modularity in other languages –

- Object Pascal language asserts dependencies among units (modules) in the interface of the module.
- Ada introduces concept of package for modularity. It has specification & body.
- C++ also introduce similar concept of namespace which is logical collection of different types.

Hierarchy :

Hierarchy is ranking or ordering of abstractions.

Main purpose of hierarchy is to achieve reusability.

- The two most important hierarchies in complex systems are its class structure (the "is–a" hierarchy) and its object structure (the "has–a" hierarchy)

- "Is–a" relationship is also called "kind–of" relationship or "generalization–specialization" hierarchy and "has–a" relationship is also called "part–of" relationship
- "Is–a" relationship is known as Inheritance and "has–a" relationship is known as composition.

Composition (has–a) :

When object is made from other small objects it is called Composition.

e.g. Room has wall, System unit has motherboard

- The concept is also there into non–OO languages that has record like structure.
- Composition allows physical grouping of logically related structures while inheritance allows common groups to be easily reused among different abstractions.
- Composition talks about ownership or collaboration of the objects.
- Composition can be done either by value or by reference just like C programming.

Each instance of type Circle seems to contain an instance of type Point. This is a relationship known as *composition*. It can be depicted in UML using a class relationship.

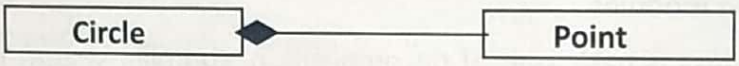


Fig. class diagram–composition

The black diamond represents composition. It is placed on the Circle class because it is the Circle that is composed of a Point. The arrowhead on the other end of the relationship denotes that the relationship is navigable in only one direction. That is, Point does not know about Circle.

- There are mainly two types of composition :
- Tightly coupled (Aggregation) : Lifetime of contained depends on lifetime of container.

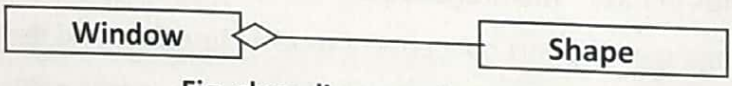


Fig. class diagram–Aggregation

Figure shows an aggregation relationship. In this case, the Window class contains many Shape instances. In UML the ends of a relationship are referred to as its "roles". This indicates that the Window contains many Shape instances. Notice also that the role has been named. This is the name that Window knows its Shape instances by. i.e. it is the name of the instance variable within Window that holds all the Shapes

- Loosely coupled (Association) : No such constraint. Contained is not dependent on container.

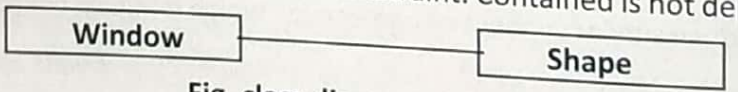


Fig. class diagram–association

An association is nothing but a line drawn between the participating classes.

Containment

Composition that is used to store several instances of the composited data type is referred

to as containment. Examples of such containers are arrays, linked lists, binary trees and associative arrays.

Inheritance (is–a)

Acquiring all the properties (data members) and behaviors of one class by another class is called inheritance.

- Inheritance is relation in which, one class shares behavior of one or more classes.
- In inheritance, subclass represents general behavior of its super class. Thus subclass (derived) is specialized version of super (base) class. Subclass is a super class.
- Subclass represents specialization in which fields and methods from super class are added, modified or even hidden.
- There are two types of clients for a class :

  One who is invoking functions on the objects and

  Subclasses that are inherited from the class

The inheritance relationship in UML is depicted by a peculiar triangular arrowhead.

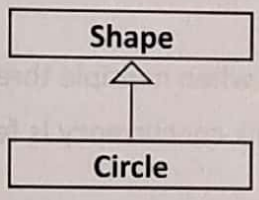
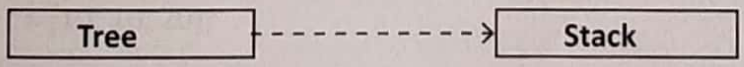


Fig. class diagram–Inheritance

Figure shows the form of the inheritance relationship. In this diagram we see that Circle derives from Shape. Note also that the operations.

Dependency (use–a)

Sometimes the relationship between two classes is very weak. They are not implemented with member variables at all. Rather they might be implemented as member function arguments.



In UML dependency is represented by using dotted line.

Polymorphism :

It is Greek word which is a combination of poly (many) + morphism (forms/behavior) One interface having many behaviors such phenomenon is called polymorphism. Consider role of person, role of person in his family is family member, role of person in his friend circle is as a friend, and role of person in his company is employee, so one person having many behaviors it is nothing but polymorphism.

Types of polymorphism

There are two types of polymorphism

1. Compile time polymorphism
2. Run time polymorphism

Compile time polymorphism : when call to the function is resolved at compile time it is called compile time polymorphism. Compile time polymorphism is also called static polymorphism, false polymorphism or early binding.

In C++, compile time polymorphism is achieved by using function overloading and operator overloading.

Run time polymorphism : when call to the function resolved at run time it is called run time polymorphism. Run time polymorphism is also called dynamic polymorphism, true polymorphism or late binding. In C++, run time polymorphism is achieved by using function overriding.

Concurrency

- Some complex systems demands simultaneous execution of more than one implementations.
- The things are achieved either by splitting the application on two different systems or by creating illusion of sharing the same processor for this purpose.
- The second approach talks about creating two different path of execution known as threads & it works only if operating system supports the concept of multithreading.
- But implementing it is difficult task due to problems associated with threads like deadlocks, starvation, mutual exclusion, etc.
- The concurrency problem arises when multiple threads simultaneously access same object.
- For some languages like Smalltalk concurrency is feature of language supported in form of class "process".
- We may use a class library for providing threading support. E.g. C++ provides thread support on Windows using MFC.
- We may use interrupts to give illusion of concurrency which need hardware details.
- You need to take care of object synchronization when concurrency is introduced in the system.

Persistence

- The property of an object through which its existence goes beyond time (after its creator ends) and/or space (moving object from address space in which it was created).
- It is property by which object maintains its state across time and space.
- The concept gives rise to the concept of Object oriented databases.
- It talks about concept of serialization & also about transferring object across network.
- Very few languages like Smalltalk, C++ gives in–built support for serialization up to certain limit.
- Java supports both persistence with time as well as space with standard classes.

Chapter 3. Moving towards Object Oriented Programming

Process oriented programming :

In process oriented programming, program is divided into small code snippets i.e. functions. Program design starts from deciding steps or algorithm and then to divide it into logically related tasks. All these functions can share the global data and can modify them. In short, data can be easily modified into functions and when length of program increases, it really becomes difficult to keep the track of changes into such data. Also when a team works on a same project, it is almost impossible to control the way data can be modified.

Object oriented programming :

In object oriented programming, program is divided into small data units i.e. classes. Program design starts from deciding related data and club them together into the same class along with functions that can manipulate that data. Thing important is that the data is now not accessible outside the class and hence there is no chance of modifying that data into any function in the program.

Moving from Procedural to Object Oriented

In C language, we write structures to combine logically related data. We also write functions that can manipulate the data. e.g.

```
/*struct example in C*/
struct time {
        int hr, min, sec;
};
void display( struct time *p) {
        printf("%d :%d :%d", p–>hr, p–>min, p–>sec);
}
int main() {
        struct time t = {10, 10, 20};
        display (&t);
        return 0;
}
```

Thing to note is that, data (in function) and operation (in function) are isolated from each other. At the same time any other function (like main ()) can easily modify the data of the function.

The first step taken in C++ towards the solution is to combine data and functions together in the structure.

```
/*struct example in C++ */
struct time {
        int hr, min, sec;
```

```
        void display (){
            printf ("%d:%d:%d", this->hr, this->min, this->sec); }
        };
        int main() {
            time t;
            t.display ();
            return 0;
    }
```

Here we can see that, not only data but functions also are members of structures and are accessed in main () using '.' operator.

Here you may get the first Question, what is "this"? To understand it let's understand how C++ compiler works on this code. The member function of structure is simply converted as follows :

```
void time display(time * const this){
    printf ("%d:%d:%d", this->hr, this->min, this->sec);
}
```

Also call to function is converted from : t.display (); to time display (&t);

I think now it is too clear that, "this" is a keyword which is local to the function and holds the address of a structure variable on which that function is called.

But still data in structure is accessible anywhere. To solve this problem, a second step taken by C++ is to introduce access specifiers. Access specifiers can be applied to data members and member functions, which decide where that member can be accessible in the code.

By making the members "private", we can restrict the access of those members in that structure only. Such members cannot be accessed outside that structure. By default all members in the structure are "public", it means that they can be accessed in member functions of the structure as well as outside the structure.

Class and Object :

Thus unlike C, structures in C++ is combination of data members as well as member functions and also can restrict the access of the members using access specifiers. C++ introduced a concept of "class" with little modification of the "structure". By default every member in class is private. Class is also a user defined data type combining data members and member functions.

We can create the variable of the class, typically called as "Object". The object gets physical memory and contains data members which are declared into the class. We may say consider "class" as a blueprint of the "object" while "object" as an instance of the "class".

```
class Complex {          // class declaration
private:
        int real, imag; // data members
public:
        //member functions
};
int main() {
        Complex c1 ; // c1       is an object of class Complex
        // other code
        return 0;
}
```

Every object has certain characteristics as follows :

1. State : state of the object is determined by the values of data members of that object. If values of data members are same for two objects, we can say that object is having same state.

2. Behavior : A member function in the class decides behavior of the object. Obviously calling the same member functions on two objects having different state will show different behavior.

3. Identity : Every object has a characteristic that makes the object unique. In C++, we may notice that every object has unique address.

Data members :

Generally data members are made as private to ensure data security. We can also make them public, but in that case it will accessible outside the class.

Member functions :

Member functions of the class can be classified as follows :

1. Constructor : constructor is a member function of the class having same name as that of class. It doesn't have any return type. This function is automatically called, when object is created. Job of constructor is to initialize data members of the class. Note that constructor initialize object; do not create the object.

Constructor may take any number of arguments. We may write any number of constructors provided argument types are different for each constructor. Moreover, if we don't write any constructor, compiler provides default constructor for the class.

2. Destructor : destructor is the member function of the class having the same name of that of class preceded by"~". Like constructor, destructor also doesn't have return type. This function is automatically called, before object is destroyed i.e. when object goes out of scope. Job of destructor is to de–initialize the data members. It is mainly used to release any resource that is allocated by the object.

Destructor does not take any number of arguments. If we don't write any destructor, compiler provides default destructor for the class.

3. Mutators : mutators are the member functions of the class, which modify state of the object. These functions modify values of the data members. These are typically setter functions in the class.

4. Inspectors : inspectors are the member functions of the class, which read the state of the object. These are typically getter functions in the class.

5. Facilitators : facilitators are the member functions of the class, which do not directly contribute to the operation of that class. This function provides additional functionality like scanning or printing object data.

Section II : **Procedure Oriented Features of C++**

Chapter 4. Namespaces

Namespaces are used to localize the identifiers to avoid name collisions. Name collision may occur for global variables, global functions, classes, etc. Namespaces are necessary when two or more third–party libraries are used by the same program, because a name defined by one library may conflict with the name defined by the other library.

Default namespace is global namespace and can access global data and functions by preceding : : operator. For example : – : : a gives access to global data variable 'a'.

We can create our own namespace. Anything declared within namespace has scope limited to namespace and (: :) scope resolution is used to resolve ambiguity, 'using' statement can be used to use namespace or any member within it.

```
namespace N1 {
        int num;
}
namespace N2 {
        int num;
}
int main() {
        N1::num=10;
        N2::num=20;

        using N1::num;
        cout << num;
        using N2::num;
        cout << num;
        return 0;
}
```

As per ANSI/ISO standard of C++ , use of namespaces is compulsory. All predefined functions or classes are declared within the standard namespace called as "std". A simple C++ application can be written as :

```
#include <iostream>
int main() {
    std::cout << "hello word" << std::endl;
    return 0;
}
```

Or this can be written avoiding repeated use of "std" namespace as follows :

```
#include <iostream>
using namespace std;
int main() {
    cout << "hello word" << endl;
return 0;          }
```

Chapter 5. C++ Data Types

Like C, data types in C++ are classified as built–in types and user defined types. Moreover, all the types of C language are supported in C++ . Those are int, float, char, double, long int, short int, unsigned int, stuct, union, enum, etc. Thing to note is that the "struct" data type is modified in C++ as discussed in chapter 3.

At the same time, some additional data types are available in C++ . C++ adds two built–in types into the language.

1. **bool :** "bool" variable can store "true" or "false" values. It occupies one byte in memory.
2. **wchar_t :** "wchar_t" variable stores character values as per UNICODE standard. It occupies 2 bytes in memory and is equivalent to "unsigned short".

Also C++ adds a derived type, parallel to pointer called as "reference". Reference is simply alias to some variable. The detailed discussion on this type is available in next chapter.

And as discussed already in chapter 3, C++ adds a user–defined type called "class", which is modified version of "struct" keyword.

Chapter 6. References

Reference is sometimes referred as "alias" for the variable. It can be used as another name for the same variable. References can be used to access variable similar to pointers but without using pointer syntax.

```
int main() {
    double d1=1.1, d2=2.2;          //declaring double variables
    double &r1=d1, &r2=d2;          //declaring reference, must be initialize at declaration
    cout << d1 << sizeof(d1) << r1 << sizeof(r1) << endl; // prints 1.1 8 1.1 8
    r2 = 4.4;              // modifies value of d2 variable
    cout << d2 << sizeof(d2) << r2 << sizeof(r2) << endl; //prints 4.4 8 4.4 8
    r1 = r2;                    // assign value of r2 (d2) to the r1(d1)
    cout << d1 <<  r1 << d2 << r2 << endl;                  // prints 4.4 4.4 4.4 4.4
    return 0;
}
```

Internally references maintain the address of the variable for which it is an alias.

In C, we can pass the arguments to the functions by value or by address. C++ provides both of these features. Additionally it allows passing the arguments to the functions by references. Since references are aliases for actual arguments value of variables can be modified into the functions.

```
void sum(int a, int b, int &r) {
        r = a + b;
}
int main() {
        int z, x=15, y=28;
        sum(x, y, z);
        cout << "result = " << z << endl;        // prints  : result = 43
        return 0;
}
```

Thus in many different ways we can use references instead of pointers. Difference between pointer and reference :

No.	Pointers	References
1	pointers may not be initialized at the point of declaration.	reference must be initilized at the point of declaration.
2	pointers must be dereferenced explicitly using value at (*) operator.	References are automatically dereferenced.
3	address stored in a pointer can be modified later.	reference keeps referring to the same variable till it goes out of scope.
4	we can have pointer arithmetic, null pointers, dangling pointers.	such concepts do not exist for references.

Chapter 7. Dynamic Memory Allocation

In C, we use library functions malloc() and calloc() for allocating memory at runtime. In many situations we may not know the size required for an array or size needed for certain data structures like linked list and trees at compile time. Using this feature we can allocate the memory at runtime as per the need.

C++ has "new" operator for allocating memory, while "delete" operator for releasing that memory.

```
int *ptr = new int[5];
```

This allocates the array of 5 integer elements at runtime and returns its starting address. This address is collected in "ptr". Now this memory can be accessed using pointer notation or array notation as shown in example.

The memory must be explicitly released by the programmer using "delete" operator when it is no more required. By some mistake if we lose the address of that memory before releasing it or we forget to release that memory that memory is considered as "leaked". This memory then cannot be used by any running program or operating system, till current program terminates.

```
int main() {
        int *ptr, cnt, i, sum=0;
        cout << "enter number of elements :";
        cin >> cnt;
        ptr = new int[cnt];
        cout << "enter the elements :";
        for(i=0; i<cnt; i++) {
                cin >> ptr[i];              // can use : cin >> *(ptr+i);
                sum = sum + ptr [i];
        }
        for(i=0; i<cnt; i++)
                cout <<ptr [i] << "+";   // can use : cout << *(ptr+i) << "+";
        cout << "\b" << "=" << sum << endl;
        delete[] ptr;
        return 0;
}
```

"new" operator can be used for allocating memory of built–in types as well as user–defined types.

e.g.

```
time *t = new time;
//...
delete t;
time *arr = new time[3];
//...
delete[] arr;              // note that array is deleted using [] symbol
```

Chapter 8. Exception Handling

Exceptions are certain situations which may occur in program causing failure of the program. Typical examples are dividing by zero, array index out–of–bounds etc. C++ provides a powerful method of handling such exception using try, catch and throw keywords.

Exception handling enables us to separate error handling logic from the actual code. This feature really makes application maintainable.

"throw" keyword is used to throw the exception of any type. We can throw a variable / value of any type which indicates certain code or information representing the exception. e.g. throw i; or throw 2.3;

"try" block contains the code that may throw exception. This code may directly contain throw keyword or more often calls for few functions which may throw exception.

"catch" block contains exception handling logic depending on exception code or information thrown using throw statement, "try" block may have one or more catch blocks.

```
int main() {
        int n1, n2;
        cin >> n1 >> n2;
        try {
            if(n2==0)                   // if denominator is zero
                throw 0;        // throw  exception, control will jump to corresponding  catch
            int res = n1 / n2;  // if no exception(demoninator non–zero), calculate the result
            cout << "result :" << res << endl;          // print result
        }
        catch(int n) {              // catch the exception if thrown
            cout << "exception code" << n << endl;          // prints : exception code 0
        }
        catch (...) {         // generic catch block
            //this will not be called in this case
            //as exception is already caught above
        }
        return 0;
}
```

We may write a generic catch block, which may catch the exception of any type. But this method does not provide a way to get the information about the exception.

If function is throwing exception, its prototype must append the throw keyword indicating types of exceptions can be thrown by the functions. It allows caller of the function to write appropriate catch blocks. This feature is called as "exception specification list".

e.g. `void fun() throw(int, float, double);`

`// function can throw int, float or double`

```
void fun();
// function do not throw any exception
void fun() throw();
//function can throw exception of any type i.e. generic catch block
```

If function throws any exception that is not mentioned into exception specification list, or if no matching catch block is found, program terminates abnormally.

Chapter 9. C++ Functions

Inline functions :

In C, we can use macros as well as functions. The major difference between macros and functions is that, macros are replaced before compilation by preprocessor, while functions are called at runtime. So macros are much faster than functions. Also function arguments have types, while macro arguments don't have types. In other words, functions are type safe, while macros are not.

"Inline" functions try to achieve advantages of both macros and functions. Functions can be made inline using "inline" keyword before function declaration. Inline functions are replaced at their calls by the compiler. It ensures faster execution of functions like macros. Every inline function may not be replaced by the compiler; rather compiler avoids replacement in certain cases to optimize the code. Typically function containing switch, loop or recursion may not be replaced.

Default arguments :

In C++, functions may have arguments with the default values. Passing these arguments while calling that function is optional. If such argument is not passed, then its default value is considered. But if argument is passed, its default value is ignored and passed value is considered. Default arguments should be given in right to left order. In other words, all default arguments must be at the right side as follows :

```
int sum (int a, int b, int c=0, int d=0) {
        return a + b + c + d;
}
```

The above function may be called as

```
res=sum(10,20);        // 3rd argument is 0 and 4th argument is 0
res=sum(10,20,40);     // 3rd argument is 40 and 4th argument is 0
res=sum(10,30,40,50);  // 3rd argument is 40 and 4th argument is 50
```

Function overloading :

Functions with same name and different signature are called as overloaded functions. Functions should differ in types of arguments, count of argument or order of arguments. Return type is not considered for function overloading.

Here are few examples of overloaded functions.

```
int sum(int, int);
int sum(int, int, int);
int sum(int, float);
int sum(float, int);
```

Function call is resolved according to types of arguments passed while calling the functions. Compiler internally changes the names of the functions according to their arguments. This is called as "name mangling". To avoid this name mangling, we can use linkage directive – extern "C". This feature may be required to call C++ function from a C function or vice–versa.

const functions :

A member function of class can be made as constant function by appending "const" keyword in function prototype. State of object invoking const function cannot be modified within const function. In other words, we cannot modify any data member of the class on this pointer. So inspector functions and all functions that do not modify state of object should be made const.

Constant functions cannot modify data members of the class. But there is an exception to this rule that, data members declared with "mutable" keyword can be modified in function.

If any object is declared as "const object", one cannot modify the state of the object. In other words, const object can invoke only const functions of the class. Following example explains const object and const function :

```
class A {
        int a;
        int b;
public :
        // other code
        void set(int x) {
                a = x;
        }
        void disp() const {
                cout << a << b << endl;
        }
};
int main() {
        A obj1;
        const A obj2;       // cannot modify
        obj1.set (1);
        obj1.disp();        // can't modify obj1 in disp()
//      obj2.set(2);        // not allowed
        obj2.disp();        // allowed
        return 0;
}
```

Section III : Object Oriented Features of C++

Chapter 10. Friend Keyword

Friend functions :

Friend functions are non–member functions of the class, which can access private members of the class. The function can be made as friend by using "friend" keyword in its declaration. Note that friend function can be global function or member of another class.

```
class abc {
private:
        int a, b;
public:
        // other code
        void display() {
                cout << a << b;
        }
        friend void myfun(abc *p);
};
void myfun(abc *p) {
                p–>a = 1;           // can access private number
                p–>b = 2;           // through pointer 'p'
}
int main() {
                abc obj;
                myfun(&obj);
                obj.display();
                return 0;
}
```

Friend class :

All member functions of friend class, can access private members of the class in which it is made friend. For example, if class A is friend of class B, all member functions of class A, can access all members of friend B. Note that class B cannot access members of class A.

```
class A;//forward declaration
class B {
private :
        int b;
public:
        // other code
};
class A {
public:
        // other code
        void myfun() {
                B obj;
                obj.b = 11;          // can access private member
                //...          }     };
```

Chapter 11. Operator Overloading

In C, we have seen that operators are used with built–in types only and we cannot use them with user defined types. Operator overloading means extending meaning of the operators so that they can be used with user defined types i.e. classes.

Operators are overloaded in form of functions using "operator" keyword. Operators can be overloaded as member functions or friend functions. When operator is overloaded as member function, first argument is passed implicitly as "this" pointer and others should be passed explicitly (if required), while in case of friend operator all arguments should be passed explicitly.

Binary operators :

To overload any binary operator, we need to pass two arguments to the function and single argument is passed for member function. The following example will make the point further clear. It overloads operator '+' as member and operator '–' as friend.

```
class Complex {
        int real, imag;
public:
        // other code
        Complex operator+(Complex c) {
            Complex temp;
            temp.real = this–>real + c.real;
            temp.imag = this–>imag + c.imag;
            return temp;
        }
        friend Complex operator–(Complex cl, Complex c2);
};
friend Complex operator–(Complex c1, Complex c2) {
        Complex temp;
        temp.real = c1.real – c2.real;
        temp.imag = c1.imag – c2.imag;
        return temp;
}
int main() {
        Complex c1, c2, c3;
        // initialize c1 and c2
        c3 = c1 + c2;       // will resolve  : c3 = c1.operator+(c2);
        c3 = c1 – c2;       // will resolve  : c3 = operator–(c1, c2);
        // other code
        return 0;
}
```

In this way we can overload binary operators. All relational operator returns bool type and generally overloaded in pairs e.g. < >, <= >=, ==, !! =.

Unary operators :

To overload any unary operator as friend we should pass one argument, while for member function no argument needs to be passed explicitly.

operator extraction << and insertion >> :

If we want to overload << and >> operators for output and input purposes respectively, then operators must be overloaded as friend functions.

```
class Complex {
        int real, imag;
public:
        // other code
        ostream& operator<<(ostream& out, complex& c) {
                out << c.real << " " << c.imag;
                return out;
        }
        istream& operator>>(istream& in, complex& c) {
                in >> c.real >> c.imag;
                return in;
        }
};
int main() {
        Complex c1, c2, c3;
        cin >> c1;                          // resolve as  : operator>>(cin, c1);
        cout << c1;                                 // resolve as  : operator<<(cout, c1);
        cin >> c2 >> c3;                    // operator>>( operator>>(cin, c2), c3);
        cout << c2 << c3;           // operator<<( operator<<(cout, c2), c3);
}
```

Index operator :

Overloading index operator allows using object of the class as an array. The example is overloading index operator for checking array bounds for arrays. Returning by reference allows us to use operator on left side as well as on right side of assignment operator.

```
class String {
        char *buff;
        int size;
public : //...
char& operator[] (int i){
        if(i>=0 && i<size)
                return buff[i];
        else
        throw i;
        }
};
int main() {
        String obj("Sun");
        ch = obj[2];        // ch = obj.operator[](2);
```

```
        obj[2] = ch;        // obj.operator[](2) = ch;
        //...
        return 0;
}
```

Operator overloading makes code more readable. There are certain limitations on operator overloading as follows :

1. Few operators cannot be overloaded. They are sizeof, conditional operator, dot operator, scope resolution operator, pointer to member operator, typeid operator.
2. Certain operators must be overloaded as members and cannot be overloaded as friend functions. They are : assignment =, index [], arrow –>, function call ().
3. We cannot change number of operands for that operator.
4. We cannot change associativity, or precedence of any operator.

copy constructor :

Copy constructor is invoked when we try to create a new object as a copy of existing object. This case mainly arrives when :

1. When we create new object by passing other object to constructor.
   ```
   Complex c1 (2, 4);
   Complex c2 (c1);
   ```
2. When we pass object to function by value.
3. When we return object from function by value.

If not written, one copy constructor is by default provided by compiler. This default copy constructor creates shallow copy i.e. all members of the object are copied as they are. Such kind of copy constructor is useful for class having simple members but in typical case where class contain pointer member that owns dynamically allocated memory, shallow copy can create problems like dangling pointer. To avoid that, we expect that, entire memory owned by the object should be duplicated. To do this we use the deep copy as shown in following example :

```
class Array {
        int len, *arr;
public:
        Array(int size) {
                len = size;
                arr = new int[len];
        }
        ~Array() {
                delete[] arr;
        }
        Array(const Array& a) {
                len = a.len;
                arr = new int[len];
                for(int i=0; i<len; i++)
                        arr[i] = a.arr[i];
        }
```

Thing to note is that, we are passing object to copy constructor, because pass by value needs copy constructor itself. Also const argument indicates that object is not going to be modified within constructor function.

Assignment operator :

Assignment operator is used to assign one object to another object. Compiler provided default assignment operator do shallow copy. We generally overload assignment operator to do deep copy as follows :

```
class Array {
    int len, *arr;
public:
    Array& operator=(const Array& a) {
        delete [] arr;
        len = a.len;
        arr = new int[len];
        for(int i=0; i<len; i++)
            arr[i] = a.arr[i];
        return *this;
    }
    //...
};
int main() {
    Array a1(4), a2(2);
    Array a3(a2);           // copy constructor called
    a2 = a1;                    // a2.operator=(a1);
    a1 = a2 = a3;           // a1.operator=( a2.operator(a3) );
    //...
    return 0;
}
```

Making assignment operator and copy constructor private, we can prevent the copy of the object.

Chapter 12 Composition

Composition represents "has a" relation. A simple example is a computer has a CPU, computer has a keyboard and computer has a mouse. In other words, we can say that a computer is composed of CPU, keyboard, mouse, etc.

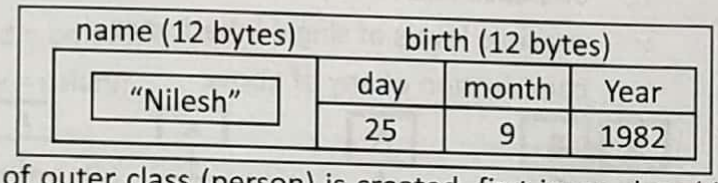
The following example explains composition :

```
class date {
        int day, month, year;
public:
        date(int day, int month, int year) {
            this->day = day;
            this->month = month;
            this->year = year;
        }
        date() {
            day = 0;
            month = 0;
            year = 0;
        }
        void display() {
            cout << day << '/' << month << '/' << year;
        }
        // other member functions
};
class person {
        char name[12];
        date birth;                 // date composed in person
public:
        person() {          // calls parameter-less constructor of birth
            strcpy(this->name, name);
        }
        person(char name, int day, int month, int year)
            :date(day, month, year) {          // init member object
            strcpy(this->name, name);
```

```
    }
    void display() {
        cout << name << endl;
        birth.display();    // display birth date
    }
    // other member functions
};
```

When object of person is created, its structure will look as follows.

person obj; // total size is 24 bytes



Whenever object of outer class (person) is created, first inner class (date) object is called and then that of outer class (person) is called. Obviously destructor calling sequence is reverse.

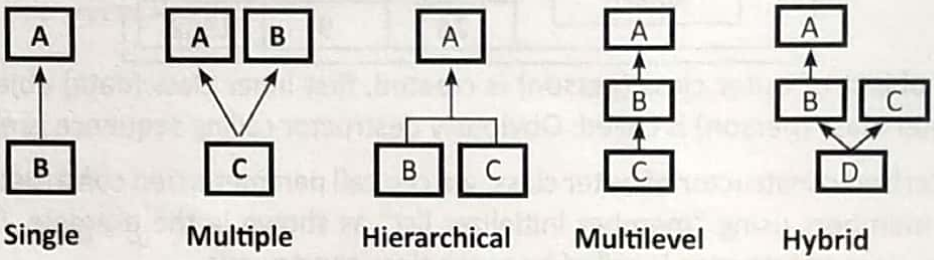
In parameterized constructor of outer class, we can call parameterized constructor of inner class to initialize its members using "member initializer list" as shown in the example. By default inner class parameter– less constructor is called by outer class constructor.

Chapter 13 Inheritance

Inheritance is an object oriented feature that allows child class to inherit all properties of parent class. In other words, if class B is inherited from class A, all data members as well as member functions of class A will be available in class B. Parent class is also called super class or base class, while child class is also called as subclass or derived class. We can see that derived class is a specialized version of the base class i.e. derived class has all members of base and at the same time it has some extra members of its own.

Types of inheritance :

1. Single : one base and one derived
2. Multiple : multiple bases and one derived
3. Hierarchical : one base and multiple derived
4. Multilevel : multiple levels of single inheritance
5. Hybrid : combination of any of above



Inheritance represents "is a" or "kind of" relationship. class Employee can be inherited from class Person, representing that Employee is a Person or Employee is kind of Person.

```
class Person {
        char name[32];
        int age;
Public:
        Person() {
                age = 0;
                name[0] = "\0";
        }
        Person(char name[], int age) {
                strcpy(this–>name, name);
                this–>age = age;
        }
        void display_person() {
                cout << name <<"<< age;
        }
        //...
};
```

```
class Employee  : public Person {
        int empid;
        double salary;
public:
        Employee(){  // by default calls parameter less constructor of base class
                empid = 0;
                salary = 0.0;
        }
        Employee(char name[], int age, int empid, double salary)
                :Person(name, age)              // calling base class parameterized constructor
        {
                this->empid = empid;
                this->salary = salary;
        }
        void display_employee() {
                Person::display_person();       //calls function from base class
                cout << empid << '' << salary;
        }
        //...
};
int main() {
        Person p("sandeep", 25);
        Employee e("nilesh", 26, 4, 20000);
        cout << sizeof(p) << "<< sizeof(e) << endl;  // prints  : 36 48
        e.display_employee();                   // prints  : nilesh 26 4 20000
        e.display_person();             // prints  : nilesh 26
        return 0;
}
```

Thing to note in above code is that, whenever object of derived class is created, first base class constructor is executed and then that of derived class is executed. In fact, derived class constructor first gives call to the base class constructor. By default it calls parameter–less constructor of the base class. To call parameterized constructor we can use member initializer list as shown in example.

Access specifiers :

In C++ , we can use three access specifiers : private, protected, public

access	in methods of class	in methods of derived class	outside the class
public	YES	YES	YES
protected	YES	YES	NO
private	YES	NO	NO

Modes of Inheritance :

Also keyword "public" in class declaration represents, mode of inheritance. There can be three modes of inheritance : private, protected, public. Mode of inheritance allows restricting the access of base class members into derived class.

A. In public mode of inheritance,

1. public member of base becomes public members of derived.
2. protected members of base become protected members of derived.
3. private members of base become private members of derived [not accessible in derived].

B. In protected mode of inheritance,

1. public member of base becomes protected members of derived.
2. protected members of base become protected members of derived.
3. private members of base become private members of derived [not accessible in derived].

C. In private mode of inheritance,

1. public member of base becomes private members of derived.
2. protected members of base become private members of derived.
3. private members of base become private members of derived [not accessible in derived].

virtual base class :

As a special case of hybrid inheritance, see the following example. In this example, class A is base class for class B and class C, while class B and C are base classes for class D. This typical inheritance is sometimes also referred as diamond inheritance.

Thing can be noted that, members of class A are inherited to class D via class B as well as class C. So calling any data member or member function of class A on class D object causes ambiguity error. This is called as "diamond problem".

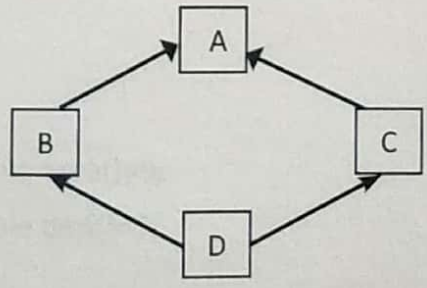
To avoid this problem, we should have a single copy of the base class A into class D. This can be achieved by making class A as a virtual base class. When class A is virtually inherited to class B and class C, single direct copy of class A members comes to class D, and remaining members of class B and C are also inherited to the class D.

To keep the track of virtual base class members within derived class object, internally virtual base pointer is created within the object.

In this program, when we create an object of class D, constructor calling sequence will be : A, B, C and then D. Obviously destructor calling sequence will be reverse of that.

```
class A {
public:
        int a;
        A() {
            a=1;
        }
        void disp() {
            cout << "a:" << a << endl;
        }
};
class B  : virtual public A {
```

```
public:
        int b;
        B() {
                b=2;
        }
        void dispB() {
                A::disp();
                cout << "b:" << b << endl;
        }
};
class C : virtual public A {
public:
        int c;
        C() {
                c=3;
        }
        void dispC() {
                A::disp();
                cout << "c:" << c << endl;
        }
};
class D  : public B, public C {
public:
        int d;
        D() {
                d=4;
        }
        void dispD() {
                B::dispB();
                C::dispC();
                cout << "d:" << d << endl;
        }
};
```



Chapter 14 - Virtual functions

Member functions are called depending on the object (or pointer or reference) on which it is called. But virtual member function is called on type of object rather than type of pointer or reference on which function is called.

To make a function virtual, "virtual" keyword is preceded in its declaration. If function is declared virtual, it remains virtual in all its derived classes. The function can be redefined in derived class, with same signature. This is called as function overriding.

	Function overloading	Function overriding
1	Function with same name and different signature.	Redefining Functions in derived class with same name and same signature.
2	Functions are in same scope.	Functions in base class are overridden in derived class.
3	Compile time polymorphism	Run time polymorphism
4	Function call is resolved based on arguments passed.	Function call is resolved based on object on which it is called.
5	In C++ , don't need a keyword.	In C++ , "virtual" keyword required.
6	Function call is resolved by the compiler. For that compiler internally modifies names of the functions, which is called as "name mangling".	Function call is resolved at runtime. Internally it use the v–table and v–pointer mechanism.
7	Since compiler modifies name of function, it is also called as "false polymorphism".	It is also called as "true" polymorphism.

Note : v–table of a class keeps address of virtual functions of that class, while v–pointer of an object keeps address of v–table. Detailed discussion of internal mechanisms is out of scope of this book.

The following program shows simple example of virtual functions.

```
class person {
private:
        char name[20];
        int age;
public:
        // other code
        virtual void display() {
            cout << name << age;
        }
};
class emp  : public person {
private:
    int empid;
```

```
public:
    // other code
    void display() {
        cout << empid;
        person::display();
        }
};
int main() {
        emp e;
        person *p = &e;
        p->display();     //will call emp::display()
        return 0;
}
```

Abstract class :

If virtual function is equated to zero, it is called as pure virtual function. Generally pure virtual functions do not have body. Class containing at least one pure virtual function is called as "abstract class". Object of abstract class cannot be created.

Pure virtual functions must be overridden in derived classes. If the function is not overridden, even object of derived class cannot be created. Abstract classes can have data members as well as other member functions. Derived classes use this implementation of abstract base classes. An abstract base class contains the code that can be reused by all its derived classes.

A special abstract class, that contains only pure virtual functions and no data members or other functions called as "interface". Obviously object of interface cannot be created. But overriding each method of interface is mandatory for the derived classes. Thus interfaces form a kind of contract with the derived classes.

Even though object of abstract class or interface is not possible, pointers of references can be created and will be used to call virtual functions of derived classes. The main use of abstract class or interface is to create a model for derived class by using certain pure virtual functions into it. Use of interface ensures that a derived class contains all functions overridden from the interface, as they are pure virtual functions.

Following program shows the example of interface class :

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

```
class circle  : public shape
{
        double radius;
public:
    // other code
    virtual double area() {
        return 3.14 * radius*radius;
    }
    virtual double peri() {
        return 2 * 3.14 * radius; } };
```

Chapter 15 Generic Programming

Many times we need to write the code that is common for many data types. The simplest example is Swap() function. We can observe that logic of swapping remain same irrespective of the data type used.

We can write templates of function, instead writing separate function for each data type. The syntax is as follows :

```
template<class T>
void Swap(T& a, T& b) {
    T c = a;
    a = b;
    b = c;
}
```

Now the same function can be used to swap, two integer variables of float variables or even any user defined type's variables.

If function is called as :

```
Swap(a, b);     // where a and b are int variable
```

Compiler converts above template function, to the function for swapping integer variables. To do this, compiler simply converts above function to a new function by replacing "T" with "int". Thus compiler creates the functions corresponding to every data type for which function has been called.

The way we can use template functions to represent generic algorithms, we can write template class to represent generic classes. The best examples are data structure classes like stack, queue, linked list, etc. Template classes are also called as "meta classes", because compiler generates real classes from template classes and then create the objects.

The syntax of template class is as follows :

```
template<class T>
class stack {
    T arr[5];
    int top;
Public:
    stack() {
        top = –1;
    }
    void push(T val) {
        top++;
```

```
            arr[top] = val;
        }
        T pop() {
            T val = arr[top];
            top - -;
            return val;
        }
        // other functions
};
```

While creating the object, one should specify the data type for which stack is to be created.

```
int main() {
    stack<int> s1;                          // create stack of int
    s1.push(11);
    stack<double> s2;                       // create stack of double
    s2.push(2.2);
    stack<emp> s3;                          // create stack of "emp" objects
    emp e;
    s3.push(e);
    return 0;
}
```

Section IV : **Miscellaneous Topics**

Chapter 16 File and Console I/O basics

Chapter 17 RTTI & casting operators

Standard classes for file handling are given into following table.

	Class name	Meaning
1	ifstream	For reading from file
2	ofstream	For writing to file
3	fstream	For reading and writing

File can be opened by using open() member function and closed by using close() member function. The syntax can be given as follows :

```
ifstream file;
file.open("file.txt", ios::in);
//...
file.close();
```

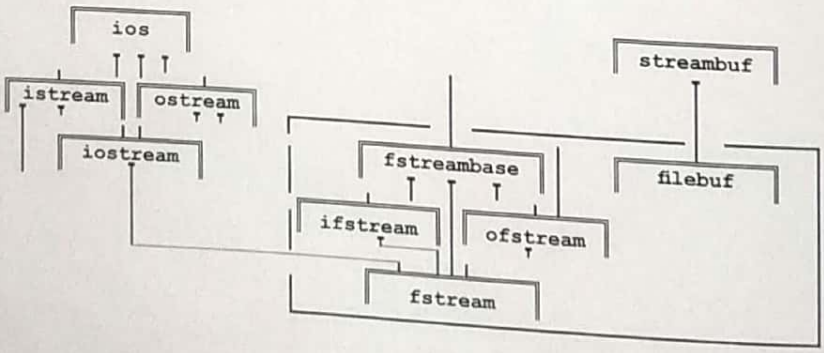
open() member function takes path of file as its first argument, while mode as second argument. The number of possible modes along with their meanings are listed in following table :

	Mode	Meaning
1	ios::in	read mode
2	ios::out	write mode
3	ios::app	append mode
4	ios::binary	binary mode
5	ios::trunc	truncate existing contents
6	ios::nocreate	do not create new file
7	ios::noreplace	do not replace existing file

These are few important modes of opening file. Modes can be combined using bitwise OR operator e.g. ios::in | ios::binary

If function fails, stream evaluates to false.

The following diagram show all classes related to File handling and console input/output. It also shows the relationships between the classes.



Chapter 17 RTTI & casting operators

C++ provides feature of Run Time Type Information, which enables to find type of the object at runtime. For this feature, language contains "typeid" operator that returns reference to constant object of "type_info" class. This class is declared in header <typeinfo> and contains information about the data type i.e. name of type.

For example, assuming that class circle and rectangle are inherited from shape interface, following code explains use of RTTI :

```
int main() {
    shape *p;=NULL;
    int choice;
    cin >> choice;
    if(choice==1)
        p = new circle;
    else
        p = new rectangle;
    const type_info& info = typeid(*p);
    cout << info.name() << endl;
    delete p;
    return 0;
}
```

If user choice is 1, then p will point to circle object, else p will point to rectangle object. Thus using RTTI one can determine the type of object and get its information into type_info object.

Important member functions of type_info are listed below :

1	const char* name() const;	prints name of data type
2	bool operator==(const type_info& rhs) const;	check equality of two type_info objects
3	bool operator!=(const type_info& rhs) const;	check non–equality of two type_info objects

Note that RTTI internally use v–table, so for "run–time" type information, class must have minimum one virtual function.

Casting operators :

C++ supports C style casting and also provides other four casting operators. These are mainly used to cast pointers or references of one type to another. Syntax for casting operator is as follows :

```
dest = cast_operator<dest_type>(source);
```

1. **static_cast :**

This casting operator can be used for any standard cast. If casting operator is used for any casting

pointer of one class to pointer of another class, it verifies inheritance relation between these two types at compile time. If there is no such relation, cast fails and gives compile time error.

```
int main() {
        int a=23, b=5, c;
        c = static_cast<double>(a) / c;
        // other code
        return 0;
}
```

2. dynamic_cast :

This casting operator verifies validity of cast at run–time. If it is valid them cast succeeds else cast fails. If cast fails in case of pointers, then it returns NULL pointer and if it fails in case of references, then it throws "bad_cast" exception.

dynamic_cast internally uses RTTI & hence class must have minimum one virtual function in the class. In other words, class must be polymorphic class. Otherwise code causes compile time error.

Assuming that class D is inherited from class B, following code explains dynamic_cast operator :

```
int main() {
    D d, *pd;=NULL;
    B b, *pb; =NULL;
    // init pb to d or b
    pb = &d or pb = &b;
    pd = dynamic_cast<D*>(pb);
    if(pd!=NULL)
        cout << "success" << endl;
    else
        cout << "failed" << endl;
    return 0;
    }
```

In this case, if "pb" is initialized to D class object (base pointer to derived object), dynamic_cast will try to convert D object address to "pd" (derived pointer to derived object). This is valid cast and hence cast will succeed, and any function of D class can be called properly on "pd" pointer.

If "pb" is initialized to B class object (base pointer to base object), dynamic_cast will try to convert B object address to "pd" (derived pointer to base object). This is invalid cast and hence cast will fail i.e. return value is NULL.

3. const_cast :

This casting operator is used to remove const–ness of any pointer temporarily.

Following code will explain it :

```
void fun(const int *p) {
    int *q = const_cast<int*>(p);          // convert "const int*" to "int*"
    *q = 20;                               // can modify the value
    }
int main() {
    int a=10;
    cout << "a  : "  << a << endl;         // print 10
    fun(&a);
    cout << "a  : " << a << endl;          //print 20
    return 0;
}
```

4. reinterpret_cast :

This casting operator can convert from one type to any other type. It does not check for any relationship, just casts the value from one type to another. Using such casting, results may be surprising. The example shows a typical case of reinterpret_cast :

```
class A {
        int a;
public:
        A() {
            a = 10;
        }
        void display() {
            cout << "a : " << a << endl;
        }
};
int main() {
        A obj;
        obj.display();     // prints  : 10
        int *p = reinterpret_cast<int*>(&obj);
        *P = 20;
        obj.display();     // prints  : 20
        return 0;
}
```