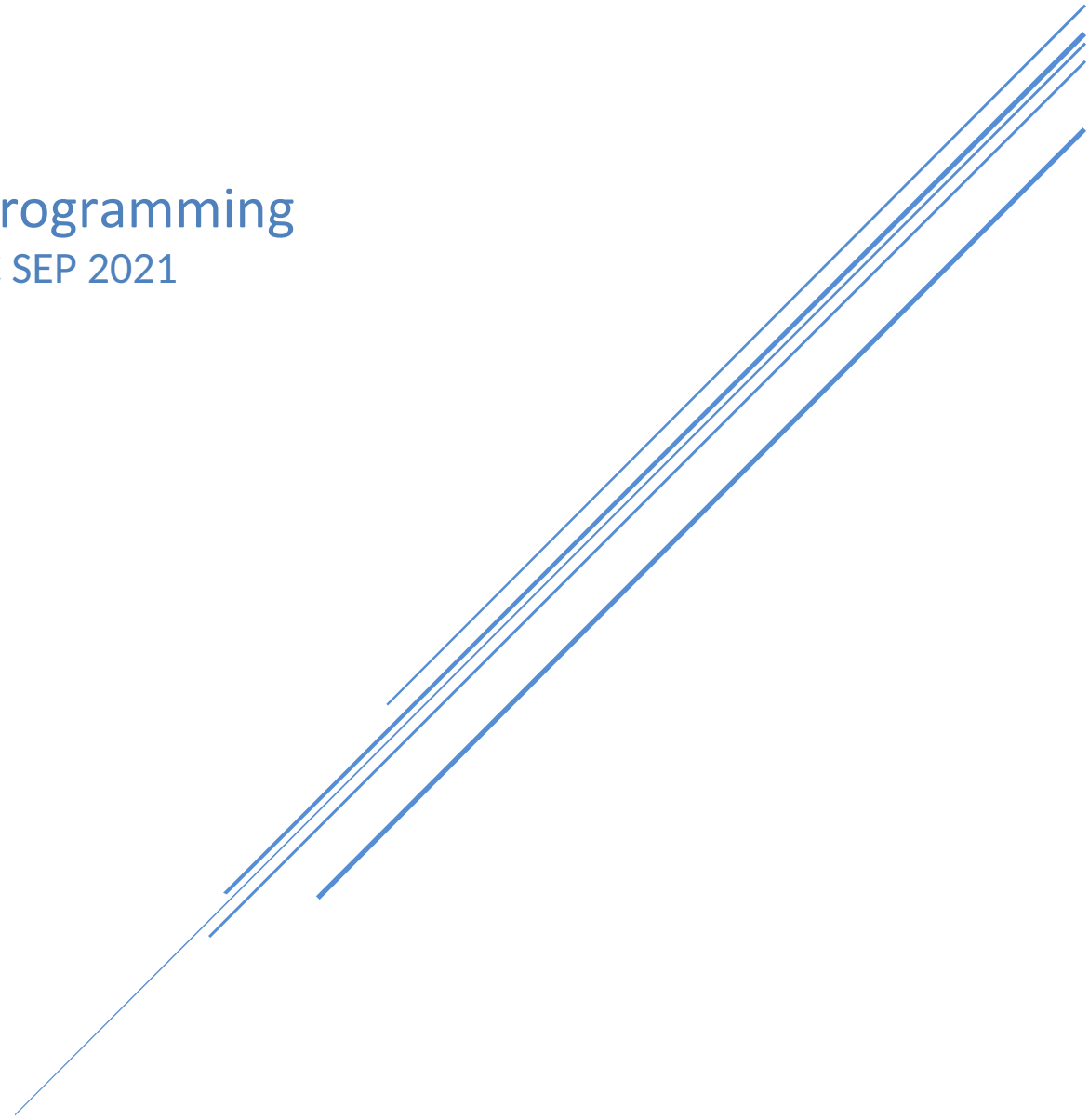
INSTITUTE FOR ADVANCED COMPUTING AND SOFTWARE DEVELOPMENT (IACSD), AKURDI

CPP Programming

PG-DAC SEP 2021



Contents

Chapter 1

Object Oriented Programming

Limitations of Procedure Oriented Programming Model

Well, although procedural-oriented programs are extremely powerful, they do have some limitations as follows.

Concentration is given on functions(procedures) but not data. Global data are vulnerable. All data are placed globally which can be accessed directly from all the functions. Each function have may have its own local data with same name as that of global data. These global data may get changed by chance from any of the functions which cannot be easily traceable in real time.

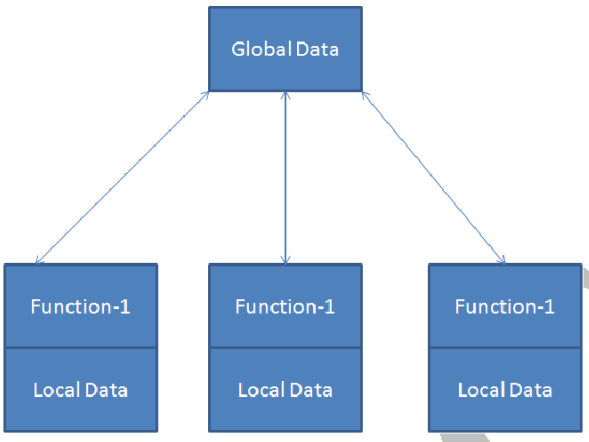


Fig. Procedure Oriented Approach

Procedural languages are difficult to relate with the real world objects.
Procedural codes are very **difficult to maintain**, if the code grows larger.
No extensibility. i.e. to add new feature to existing software the entire SDLC has to be repeated. This leads to slow development speed.

Ideal Programming Model Demands…

Reduce the communication gap between user and developer.
Natural way of binding the data and subprograms.
Data should be the priority i.e. the mechanisms for data hiding.
Enhanced reusability
Extensibility i.e. ability to add new features in software without disturbing existing procedures.
Exception handling mechanism.
Automatic Memory Management.
Robust paradigms.

What is an object?

Object is a real world entity having its own characteristics and behavior.
It possesses following characteristics:

1. State
2. Behavior
3. Identity
4. Responsibility

1. State is the current values of all attributes of an object.

Emp Id
Name
Gender
Blood group
Static attributes

Qualification
Phone Number
Email
Salary
Designation
Department ID
Dynamic attributes

Employee object

2. Behavior is all the operations/actions performed by an object
   E.g. Employee prints the details, Fills the timesheet, Swipe the Card for login and logout,
   Salary is computed.

3. Identity is an attribute that distinguishes an object from all other similar objects. It could be a single attribute or group of attributes.
   E.g.Emp Id is an identity of all employees.

4. Responsibility of an object is defined as the role of an object in the system
   E.g. Employee's role is come on time, do all assigned tasks, get salary.

Object Oriented Paradigms

There are four major pillars of OOP.

1. Abstraction
2. Encapsulation
3. Inheritance
4. Polymorphism

1. Abstraction:

It is mapping the real world entity into an object in software domain at design level. It's a process of showing the key features of an object for a particular context/ domain. All the implementation details are ignored (hidden from outside world). **We show only what an object can do, and hide the implementation details i.e. how the operations are done.**

The benefit of this approach is the capability of improving the implementation over time. The changes made in the implementation does not have any impact on client side code.

Data abstraction: It's the abstraction of attributes of an object.

Method Abstraction/Control Abstraction: it's the abstraction of action performed by an object.

Ex. For a person as an employee the data required is empid, salary,designation,etc.and functions would be computesalary,computetax,printdetails etc.

The same person object as patient the data required will be patientid,blood group, weight, previous record, etc. and the functions would be getPulse, getBP,paybill ,etc.

2. Encapsulation:

It is binding the data members with the member functions for the purpose of data protection/security at implementation level. The data/state of an object cannot be directly accessed by external functions or objects. Member functions are the public interface that accesses the data. Any change to implementation is not going to affect the external objects performing operations. By achieving encapsulation, we implement abstraction at implementation level.

Eg For employee all the functions are bound to its data only.

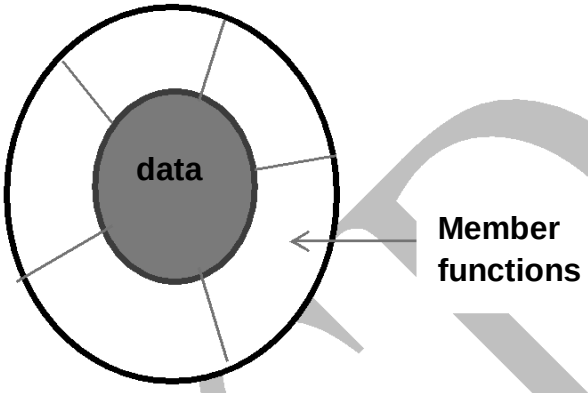


Fig. encapsulation

3. Containment:

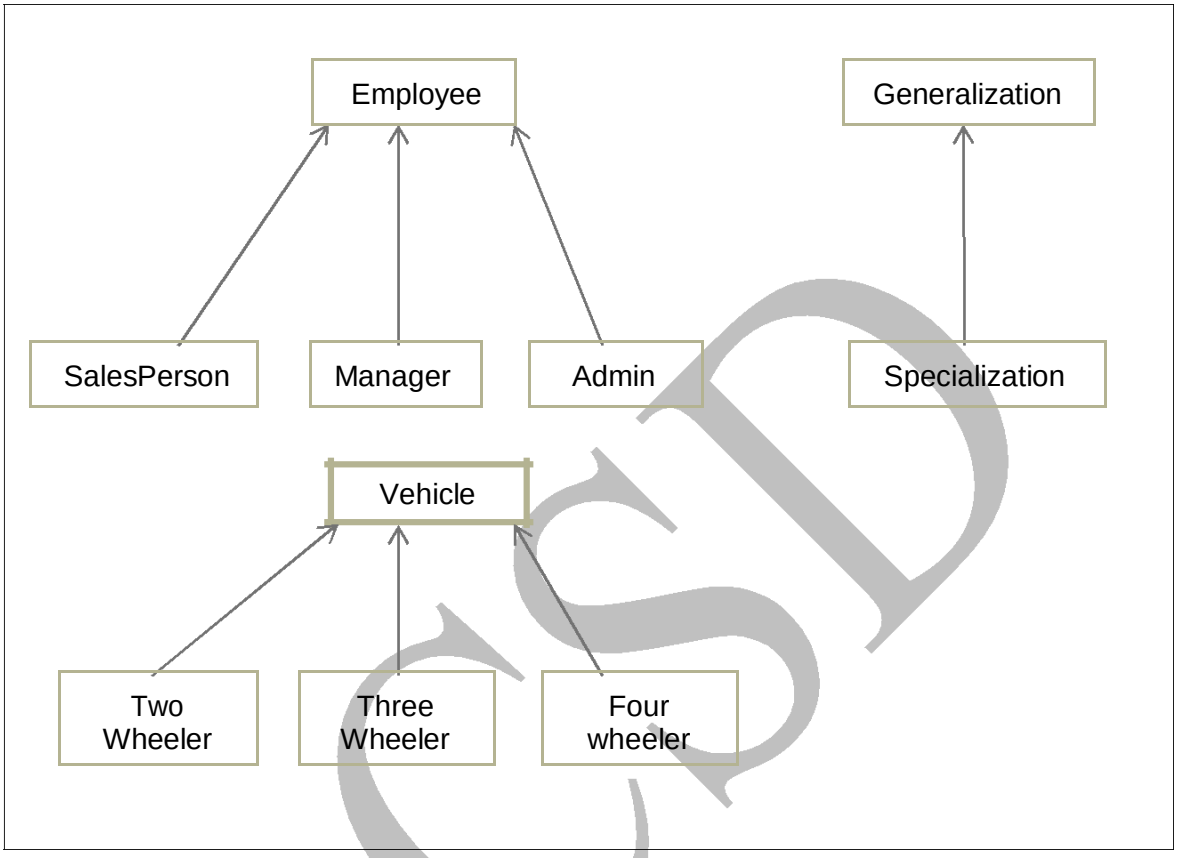
It is a feature where one object is a part of another object as its attribute. The container object contains the contained object as its attribute. There is has-a relationship. The advantage of containment is reusability. E.g. Computer has harddisk,ram,mouse, etc.

Car has an engine, wheels,etc.

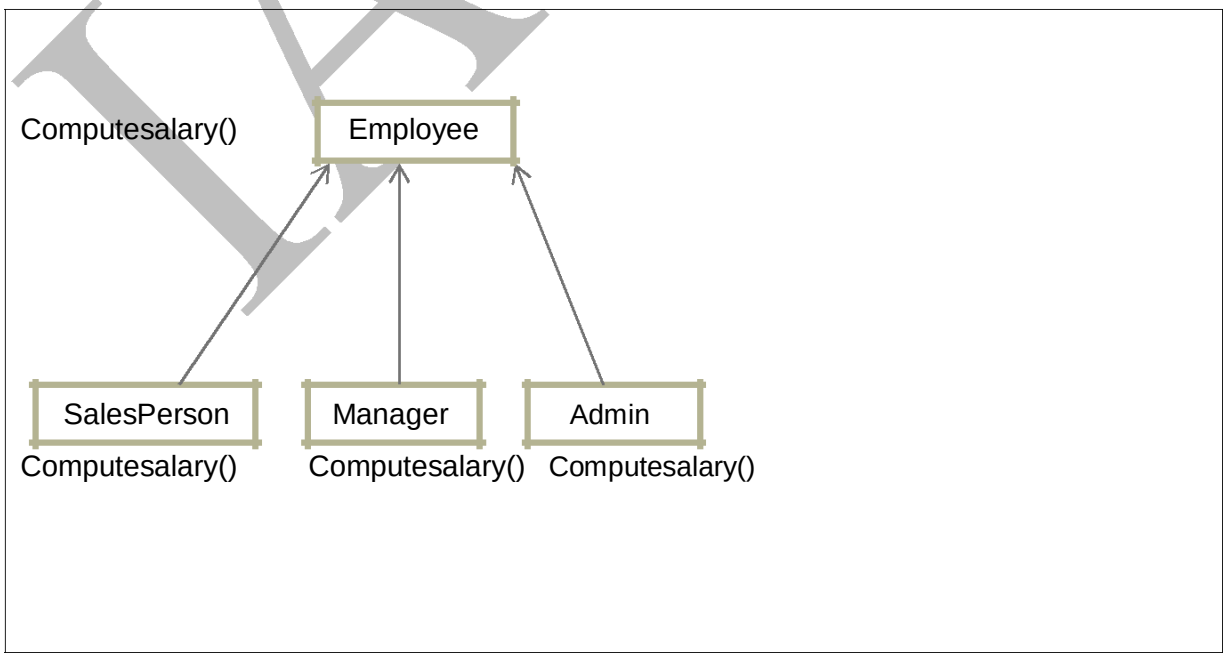
4. Inheritance:

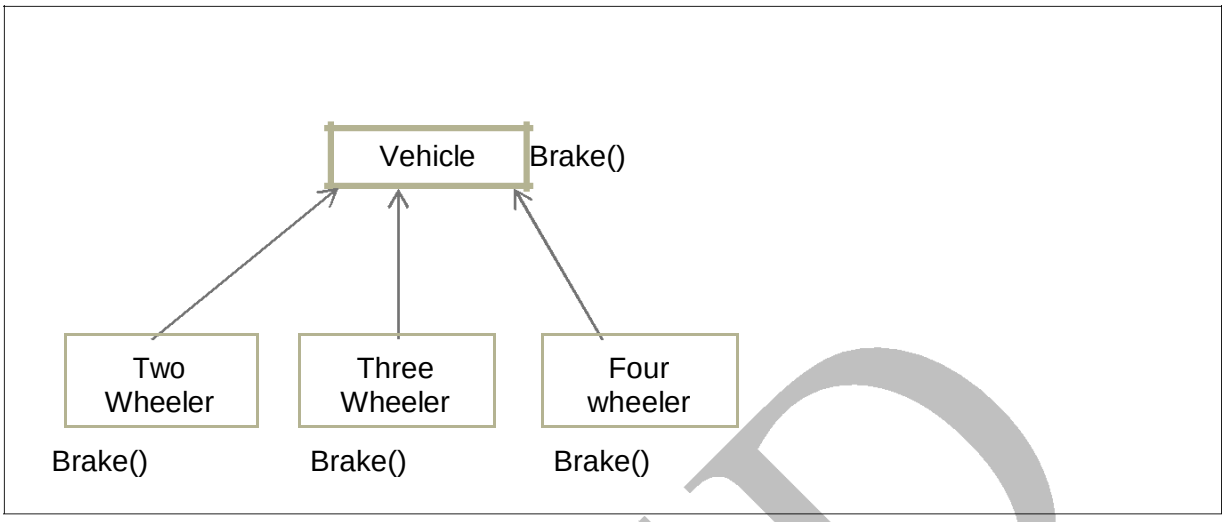
It is a feature that gives classification of all objects. There exists 'is-a' relationship.
One object takes the form of another objects i.e. one object acquires all properties and functionality of another object. A hierarchy is formed where a generalized broad category is created. And all its derived specialized subcategories are created. It implements two concepts:

a. Generalisation:
   It is factoring out all the common elements from all subcategories and assigning it as properties to the most super class/ base class. Advantage is reusability.
   Eg All employees has empid,salary,designation,departmentid in common.
b. Specialization:
   It is process of identifying the special attributes of derived classes. Child class possesses the general as well as special attributes. Advantage is extensibilty.
   Eg. Manager has special attribute as incentives. SalesPerson has number_of_sales, commission_pct.



5. Polymorphism:
In Polymorphism different related objects respond to generalized/same message in different manner/ways. Its advantage is to design extensible software.





As shown in above example Vehicle is the generalized/ broad category which is a parent of special/ derived subcategories TwoWheeler, ThreeWheeler, and so on.There is a common/ generalized message for all subcategories i.e Brake() of any Vehicle. But the implementation/ response is going to vary according to special categories. So providing a way to implement such a behavior where response for a generalized function varies according to special categories object, is called polymorphism.

Chapter 2

CPP Programming fundamental

Input/Output objects

COUT
It is a predefined object of ostream class. It is used in conjunction with insertion operator(<<), for printing the text on the console. It is typesafe i.e. no need of format specifiers to give information of type. All types are inferred at compile time.

CIN
It is a predefined object of istream class. It is used in conjunction with extraction operator(>>), for input from console and assigning it to variable. It is typesafe i.e. no need of format specifiers to give information of type. All types are inferred at compile time.

Reference

It is an alternative name given to a variable. No separate memory is allocated to a reference.
It is used like a variable. It has to be initialized. Once initialized it cannot be assigned to another variable, hence it is a rigid connection.

Syntax:

Data_type&reference_name= variable name;

```
intnum=10;
int&refnum=num;
cout<<num<<endl;
cout<<refnum<<endl;
refnum++;
cout<<num<<endl;
cout<<refnum<<endl;
```

1000 num/refnum

output is
10
10
11
11

```
int num1=10,num2=20;
int&refnum=num1;
----
----
refnum=num2;
---
---
```

Difference in Pointers and Reference

Pointers	Reference
It's a separate variable that stores an address of another variable	It's an alternative name given to the variable
It has its own separate block of memory	It doesn't have a separate block of memory
It's a flexible connection i.e. a pointer declared can point to any variable, provided it's a non-const	It's a rigid connection. i.e. A reference associated with a variable while initialization can't be assigned to another variable.
It needs an indirection operatorfordereferencing	It doesn't need any operator for dereferencing

Inline functions

Functions and Macro-functions are used for avoiding repetition of a instructions in a program. Both has got its own advantages and limitation.

Macros:
These are processed during preprocessing time i.e. before compilation. These are based on the principle of text replacement i.e. at preprocessing time the macro value is replaced wherever the macro name is referenced. Therefore the working of macros is faster. But as it is processed before compilation, it is not type safe. It can work with any type, which is not feasible.

Functions:
These are processed during compilation time. Therefore it is type safe. Whenever the function call statement executes the control is transferred to the function definition. That increases the function call overheads, which increases the time complexity.

In CPP, there is concept of inline function that has the combination of macros and functions. Inline functions are based on inline substitution of function definition and are processed during compilation therefore are type safe. The keyword used is **inline**.Any change to an inline function could require all clients of the function to be recompiled because compiler would need to replace all the code once again otherwise it will continue with old functionality.

Dynamic memory allocation in CPP

Memory allocated at runtime or during execution is referred as dynamic memory allocation.
It is needed when we don't know the array size/or number of records to be stored during programming time/compile time. In C-Programming it is achieved using malloc/calloc/realloc/free functions. Although CPP has backward compatibility with these functions, but still in CPP we have special operators for same. Operators take care of dynamic memory allocation and deallocation.

New operator:
This operator is used for runtime memory allocation. It returns the base address of the block allocate on heap.

```
….
….
int * arr,noe;
cout<<"\n Enter the number of elements::";
cin>>noe;
arr=new int[noe];
…
…
```

As shown, we directly specify the data type of the blocks to be allocated, therefore there is no need of typecasting of base address like malloc().As well as there is no need to use the sizeof operator to calculate the total size of block.

Delete operator:
It used for deallocating the memory allocated by new operator. If we don't delete the memory then it leads to memory leakage.

```
…
…
int * arr=new int[5];
…
…
delete [] arr;
…
```

When subscript [] operator is used with new operator to allocate memory it is necessary to use subscript [] operator with delete operator as well, otherwise it leads to memory leakage.

```
int * arr=new int[5];
…
…
delete arr;
…
```

The above code, the new operator allocates a memory of 20 bytes. But delete arr, deletes only 4 bytes from base address and there is a memory leakage of rest 16 bytes.

Classes & Objects

Class:
It is a template for creating similar kind of objects.It defines the structure and behavior of an object. When we define a class two pillars of OOPs are implemented i.e Abstraction and Encapsulation. It's a user-defined data-type to create objects.

Syntax:	**Example:**
classclass_name **{** **access_specifier :** **data_member declaration;** **member_function definition;** **};** **Three core components of class** **1. Access Specifier** **2. Data Members** **3. Member Functions**	**class Employee** **{** **private:** **intempid;** **char * name;** **…** **public:** **printdetails(){…}** **computesalary(){…}** **…** **};**

When defining a class there are important components as follows:

1. Access Specifier:
   It is a keyword that specifies the scope of the component. There are three specifiers.

public	Accessible within class and outside class.
private	Accessible only within class.
protected	Accessible within class and its next derived class.

2. Data Members:
   It is definition of data/attributes of corresponding object. To achieve encapsulation the data is made private.
3. Member functions:
   It is the behavior/action/operations of an object. To achieve Encapsulation some functions are made public. There are three types of functions,
   a. Constructor
   b. Destructor
   c. Ordinary functions.

   Object is an instance of a class. The process of creating objects is called as instantiation. When a object is created following happens:
   a. Memory is allocated
   b. Constructor is called.
   c. Memory is initialized.

Constructor:
It is a special member function that is used for initializing the object. It is called implicitly by compile when an object is created. As it is called implicitly, the name of constructor has to be same as that of class name. It doesn't have any return type. It can have parameters, and therefore we can have multiple constructors in a class with different signature(number of parameters / sequence of parameters / type of parameters). Constructor without any parameter is called as default parameter. If we don't provide default constructor in a class, then compiler provides the default constructor.

Example:

```
class Employee
{
private:
        intempid;
        char * name;
        …
public:
         Employee()
         {
                empid=1;
                strcpy(name,"abc");
         }
          Employee(inteid, char * nm)
         {
                empid=eid;
                strcpy(name,nm);
         }

};

int main()
{
        Employee e1; //default constructor called
        Employee e2(101,"King"); //Parameterized constructor
        call ….
}
```

Ordinary functions:
These are normal encapsulated functions that accesses or modifies the objects data. Usually the member functions are public interface to outside world. These functions can be categorized into following:

a. Accessor/Inspector- these are the functions that only access the individual data member's value without modification. Its purpose is to get the state of the data member. Usually these functions are named as get() methods.
b. Mutator- these are the functions that modifies the data member's state. These functions are usually named as set() methods.
c. Facilitator- these are the functions that display the objects state.

These functions are called through an objects using dot(.) operator. The object invoking the function is known as current object.

'this' pointer

It is a pointer that is created and maintained by compiler. It stores the address of the current object i.e. the address of the object that invokes the function for current instruction. 'this' keyword is used for accessing this pointer in program. The operator used with this pointer is '->' arrow operator to access the current object. This pointer is used in program for:

a. Accessing current object
b. Remove variable shadowing effect.

Example:

```
class Employee
{
private:
        intempid;
        char * name;
        …
public:
          ….
          Employee(intempid, char * name)
          {
                empid=empid;
                strcpy(name,name); //logical error
          }
          ….
};
```

As shown in above snippet, the parameterized constructor is having the parameters whose name is same as that of data members name. that leads to a variable shadowing problem i.e. the desired output is parameter value should be assigned to data member, but due to same names the preference is given to parameter name.and hence the object doesn't gets initialized.

Example:

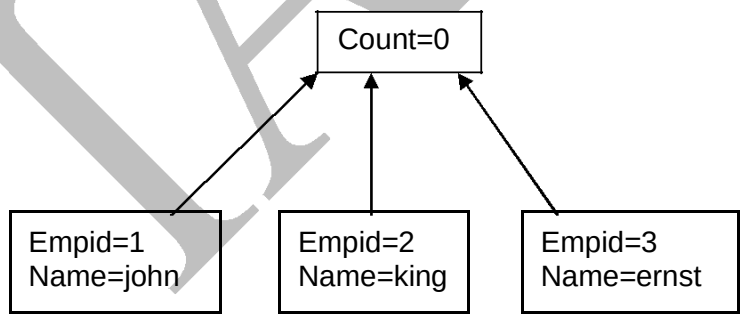
```
class Employee
{
private:
        intempid;
        char * name;
        …
public:
          ….
          Employee(intempid, char * name)
          {
              this->empid=empid;
strcpy(this->name,name);  //solution
          }

};
```

Static member and functions

There are some attributes that belong to the class level, instead of the object level.
E.g. If there is SavingAccount class then acccountNo, acholdername,balance etc. are the attributes that belong to individual object. But interestRate attribute and CalculateInterestRate function is common for all. Any change in it is shared by all objects.Such common attributes and functions are declared as static. 'static' keyword s used to declare static components. For static members only single copy exists that is shared by all objects. Static members are initialized outside the class using scope resolution operator(::).

Example:

```
class Employee
{
private:
        static int count;
        intempid;
        char * name;
        …
public:
          ….
static intgetCount(){ return count;}

};
int Employee::count=0;
int main()
{
Employee e1(1,"john"), e2(2,"king"),e3(3,"ernst");
    ….
cout<<"total employee objects are "<<Employee::getCount();
}
```



We cannot use 'this' pointer with static members as they are not a part of object. tatic members can be accessed in non-static as well as static functions, but static functions can access only static members. Static functions can't be invoked on objects. As they belong to class level, it has to be called using class name with scope resolution operator.

Function Overloading

Function overloading is a class having multiple functions with same name but different signature. The purpose of function overloading is need not remember multiple function names for same task.

```
class Math
{
    public:
        static int add(int n1,int n2)
        {
                return n1+n2;
        }
        static float add(float n1,float n2)
        {
                return n1+n2;
        }
        static char* add(char* n1,char*n2)
        {
                return strcat(n1,n2);
        }
         …
};


int main()
{
        cout<<"\n The addition of two int is"<<Math::add(10,12); cout<<"\n The
        addition of two float is"<<Math::add(10.5f,12.5f); cout<<"\n The addition
        of two strings is"<<Math::add("Hello", "World");


}
```

The names of all functions in class are mangled. Name Mangling is the process in which the name of functions is prefixed or suffixed with certain characters that make all names unique for compiler. The name mangling algorithms of all compilers are different. Therefore the object code of one compiler cannot be executed on another compiler, it has to be re-compiled.
The function calls are resolved at compile time i.e. the binding of the function calls with the objects is performed at compile time by checking the signature of actual parameters and that of formal parameters.

Destructors

These are the special member functions that are written to release the resources held by an object. E.g. if an object contains a pointer that points to heap memory block, then the heap block has to be released before the object is released.
Destructors are invoked implicitly when an object ceases to exist i.e. when an object goes out of scope or when a delete operator is used to delete an object. Asthe destructor is invoked implicitly the name of destructor is same as that of class name prefixed with ~. If we don't define a destructor in a class then it leads to situation of memory leakage. Memory leakage is situation in which memory remains allocated even if the program terminates.

```
cString::cString()
{
	this->len=0;
	this->name=newchar;
	this->name="\0";
}
cString::~cString()
{
	if(this->name!=NULL)
	{
		delete[] this->name;
		this->name=NULL;
	}
}
```

In above example, when an object is created, the constructor allocates a memory and stores the address in pointer name and when object will cease to exist the destructor is invoked and the memory pointed by name is deallocated.

Copy Constructor

Copy constructor is called when a new object is created based on another object. A new object should be copy of another object.

```
cString s1("Hello");
cString s2(s1); //copy constructor called
or
cString s2=s1; //copy constructor called
```

If we don't define the copy constructor in a class then the compilers copy constructor is invoked. Compilers copy constructor performs a shallow copying i.e. member wise copying, that leads to dangling pointer situation.
To overcome this situation class should contain its own copy constructor that has logic of deep copying.

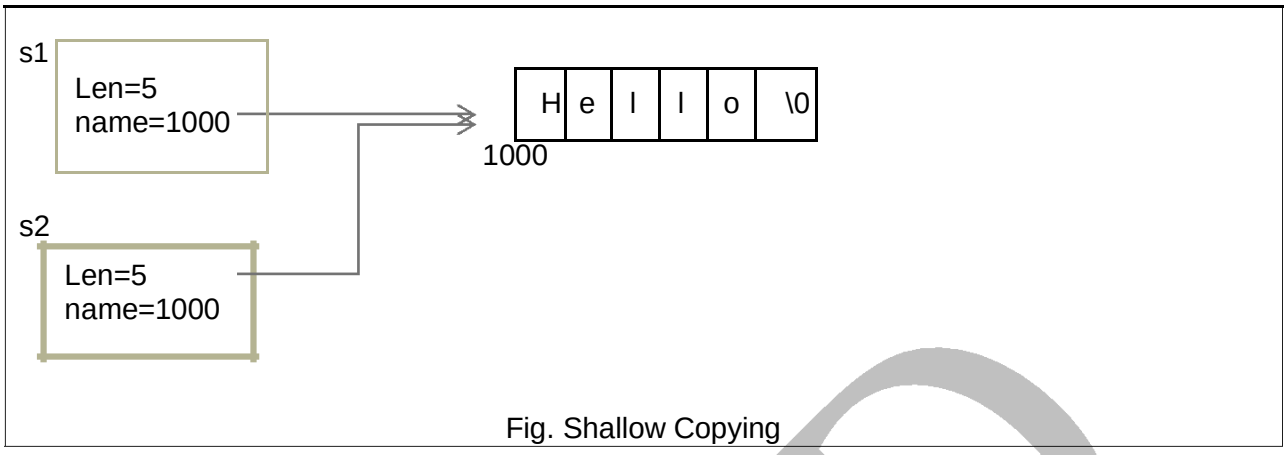


Fig. Shallow Copying

As shown in diagram, for s2 the compilers copy constructor was called that performed shallow copying. Therefore pointer name in s1 and s2 both are pointing to same memory block on heap. If any one object lets assume s1 goes out scope then a destructor is called for it which releases block on heap and then pointer name in s2 object become a dangling pointer i.e. it points to the memory that is already deallocated.To avoid the dangling pointer situation a class should define the copy constructor that has a logic for **deep copying** i.e a separate heap block is allocated for s2 and its name contains address of new block. When we define our own copy constructor the diagram would be.

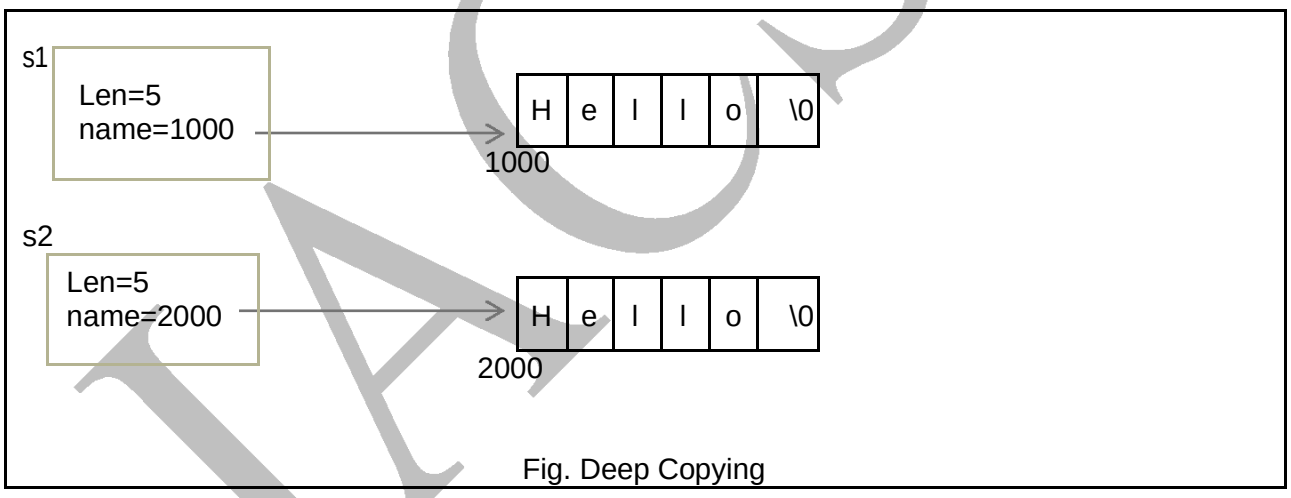


Fig. Deep Copying

Chapter 3

Operator Overloading

Operator overloading is best example of power of extensibility. All operators work with the operands of basic types. E g. + operator performs addition of all basic types, but it cant add the user defined types. Therefore CPP provides a functionality of extending the meaning of the operators, which is known as Operator Overloading, also known as operator adhoc polymorphism. The keyword used for overloading is operator suffixed with the symbol that is being overloaded.Some operators cant be overloaded such as .,.*,::,?:

Syntax:

return_type class_name::operator#(arguments)
{
Logic block
}

```
classComplex
{
private:
	float real,imag;
public:
	Complex();
	Complex(float,float);

	void accept();
	void display();

	float getReal();
	void setReal(float);

	float getImag();
	void setImag(float);

	Complex& operator+(Complex&);
	Complex& operator-(Complex&);
	Complex& operator-();

	Complex& operator++();
	Complex& operator++(int);
};

Complex::Complex()
{
	this->real=0.0f;
	this->imag=0.0f;
}

Complex::Complex(floatreal,floatimag)
{
	this->real=real;
	this->imag=imag;
}
```

```
voidComplex::accept()
{
	cout<<"\n Enter real part::";
	cin>>this->real;
	cout<<"\n Enter imag part::";
	cin>>this->imag;
}

voidComplex::display()
{
	cout<<"\n The complex number is "<<this->real<<"+"<<this->imag<<"i";
}

floatComplex::getReal()
{
	returnthis->real;
}

voidComplex::setReal(floatreal)
{
	this->real=real;
}
floatComplex::getImag()
{
	returnthis->imag;
}
voidComplex::setImag(floatimag)
{
	this->imag=imag;
}

Complex&Complex::operator+(Complex&c)
{
	Complex temp;
	temp.real=this->real+c.real;
	temp.imag=this->imag+c.imag;
	return temp;
}
Complex&Complex::operator-(Complex&c)
{
	Complex temp;
	temp.real=this->real-c.real;
	temp.imag=this->imag-c.imag;
	return temp;
}

Complex&Complex::operator-()
{
	Complex temp;
	temp.real=-this->real;
	temp.imag=-this->imag;
	return temp;
}
```

```
Complex&Complex::operator++()
{
	this->real=this->real+1;
	this->imag=this->imag+1;
	return *this;
}
Complex&Complex::operator++(int)
{
	Complex temp=*this;
	this->real=this->real+1;
	this->imag=this->imag+1;
	return temp;
}
int main()
{
	Complex c1(1,1);
	cout<<"\n C1 ----------------------     ";
	c1.display();

	Complex c2(2,2);
	cout<<"\n C2 ----------------------     ";

	c2.display();

	Complex c3;
	cout<<"\n C3 ----------------------     ";
	c3.display();

	c3=c1+c2;   //c3=c1.operator+(c2);
	cout<<"\n C3=C1+C2----------------------          ";
	c3.display();

	Complex c4;
	cout<<"\n C4 ----------------------     ";
	c4.display();

	c4=c2-c1; //c4=c2.operator-(c1);
	cout<<"\n C4=C2-C1----------------------          ";
	c4.display();

	Complex c5;
	cout<<"\n C5 ----------------------     ";
	c5.display();

	c5=-c4; //c5=c4.operator-();
	cout<<"\n C5=-C4----------------------       ";
	c5.display();

	Complex c6(1,1),c7;
	cout<<"\n C6 ----------------------     ";
	c6.display();

	cout<<"\n C7 ----------------------     ";
	c7.display();
```

```
        c7=++c6;  //c7=c6.operator++();   c6=c6+1  and c7=c6


cout<<"\n C6 after pre increment----------------------                 ";
        c6.display();

        cout<<"\n C7=++c6----------------------         ";
        c7.display();

        Complex c8(1,1),c9;
        cout<<"\n C8-------------------------    ";
        c8.display();

        cout<<"\n C9-------------------------    ";
        c9.display();

        c9=c8++; //c9=c8.operator++();  c9=c8 and c8=c8+1

        cout<<"\n C8 after post increment----------------------                 ";
        c8.display();

        cout<<"\n C9=c8++----------------------         ";
        c9.display();

        getch();
        return 0;
}
```

Chapter 4

Containment & Inheritance

Containment

It is a feature where one object is a part of another object as its attribute. The container object contains the contained object as its attribute. There is has-a relationship. The advantage of containment is reusability. E.g. Computer has harddisk,ram,mouse, etc. Car has an engine, wheels,etc.

```
classEmployee
{
protected:
	static int count;
	int eid;
	cString ename; //contained object of cSTring
	double salary;
	MyDate doj; //contained object of Mydate
public:
	Employee();
	Employee(char*,double,int,int,int);

};
```

If a class is implementing containment then the constructor invocation is in the given sequence, first the constructor of contained objects and then constructor of container object. When an object of Employee class is created the constructor invocation is as follows:
Default constructor of cString
Default constructor of MyDate
Default constructor of Employee

While defining the parameterized constructor of Employee class we have to implement member initializer list otherwise it leads to un-necessary call to default constructors. Member initialize list forces to invoke parameterized constructor directly.

```
Employee::Employee()
{
	count++;
	this->eid=count;
	this->salary=0.0;
}

Employee::Employee(char * name,doublesalary,intday,intmonth,intyear)
	:ename(name),doj(day,month,year)  //member initializer list
{
	count++;
	this->eid=count;
	this->salary=salary;
}
```

Inheritance

It is a feature that gives classification of all objects. There exists 'is-a' relationship.

One object takes the form of another objects i.e. one object acquires all properties and functionality of another object. A hierarchy is formed where a generalized broad category is created. And all its derived specialized subcategories are created. It implements two concepts:

c. Generalisation:
   It is factoring out all the common elements from all subcategories and assigning it as properties to the most super class/ base class. Advantage is reusability.
   Eg All employees has empid,salary,designation,departmentid in common.
d. Specialization:
   It is process of identifying the special attributes of derived classes. Child class possesses the general as well as special attributes. Advantage is extensibilty.

Eg. Manager has special attribute as incentives. SalesPerson has number_of_sales, commission_pct.

Modes of Inheritance

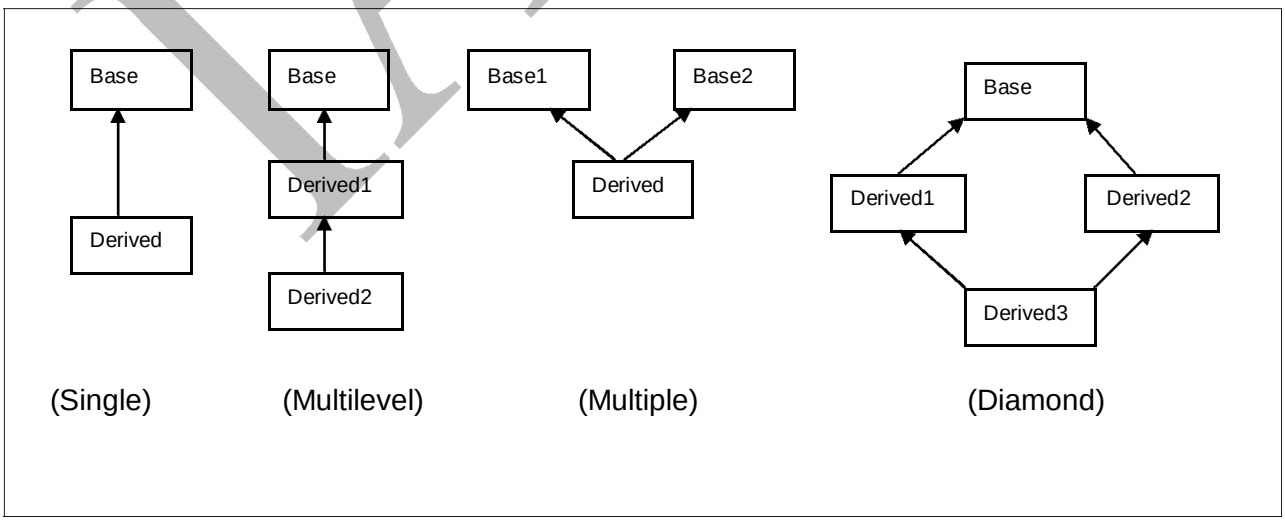
Mode of inheritance ↓	Access specifier in base class		
	public	**private**	**protected**
public	public	Inaccessible	protected
private	private	Inaccessible	private
protected	protected	Inaccessible	protected

Access specifier in derived class

Access specifier in a base class and the mode of inheritance both defines the accessibility of inherited members in derived class. As shown in a table private members of base class are always inaccessible in derived irrespective of mode of inheritance. Public inheritance is a true inheritance. Private inheritance is referred as a last stage of inheritance.

Types of Inheritance

There are different types of inheritance.



Single Inheritance: One base class and having one child class.
Multi level Inheritance: As shown in diagram derived class is in turn a base class of another derived class. One base class have multiple derived classes and form a hierarchical inheritance.
Multiple Inheritance: Child class derived from multiple base classes.
Diamond Inheritance: It is a hybrid inheritance that forms a diamond pattern of all classes.

Hierarchical Inheritance :

```
classEmployee
{
protected:
	staticint count;
	int eid;
	cString ename; //contained object of cSTring
	double salary;
	MyDate doj; //contained object of Mydate
public:
	Employee();
	Employee(char*,double,int,int,int);
};
classSalesPerson:publicEmployee
{
protected:
	int nos;
	double comm;
public:
	SalesPerson();
	SalesPerson(char*,double,int,int,int,int,double);

};
classManager:publicEmployee
{
protected:
	double fa,pa;
public:
	Manager();
	Manager(char*,double,int,int,int,double,double);
};
```

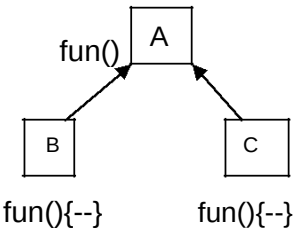
When an object of derived class is created the sequence of constructor call :
First the base class constructor and then the constructor of derived class. For defining the parameterized constructor of derived class the member initialiser list has to be implemented for avoiding the unnecessary call to default constructor of base class.

```
SalesPerson::SalesPerson(char* name,double salary,int day,int month,int year,int nos,double
comm):Employee(name,salary,day,month,year)
{
	this->nos=nos;
	this->comm=comm;
}
```

When a parameterized object of derived class is created the sequence of constructor call is the parameterized constructor of base class and then derived class.

Compile time binding

Binding is an association of a function call with an object. When a binding takes place at compile time it is known as compile time binding or static binding or early binding. For static binding the compile time type of an object is checked and the function from same class is bound to an object.



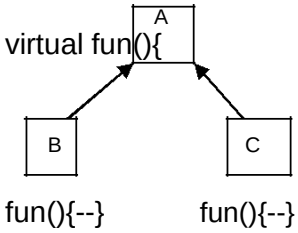
As shown base class A contains function fun(). And its derived class also contains function with the same name. this is called as method hiding.

```
A aobj;  //compile time type of aobj is class A
aobj.fun(); //invokes from class A

B bobj;     //compile time type of bobj is class B
bobj.fun();    //invokes from class B

C cobj;      //compile time type of cobj is class C
cobj.fun();    //invokes from class C

A *aptr=new A();     //compile time type of aptr is class A
aptr->fun();   //fun invoked from class A

A * aptr=new B();   // compile time type of aptr is class A
aptr->fun();   //fun invoked from class A

A * aptr=new C();   // compile time type of aptr is class A
aptr->fun();   //fun invoked from class A
```

Base class pointer is also known as generic pointer. It can point to object of base class as well as derived class type .But during compile time binding the compile time type is checked . In case of dynamic objects the compile time type of an object is the pointer type. And runtime type of an object is the actual object type. E g. A * aptr=new C(); the compile time type is class A and the runtime type is class C.

Polymorphism

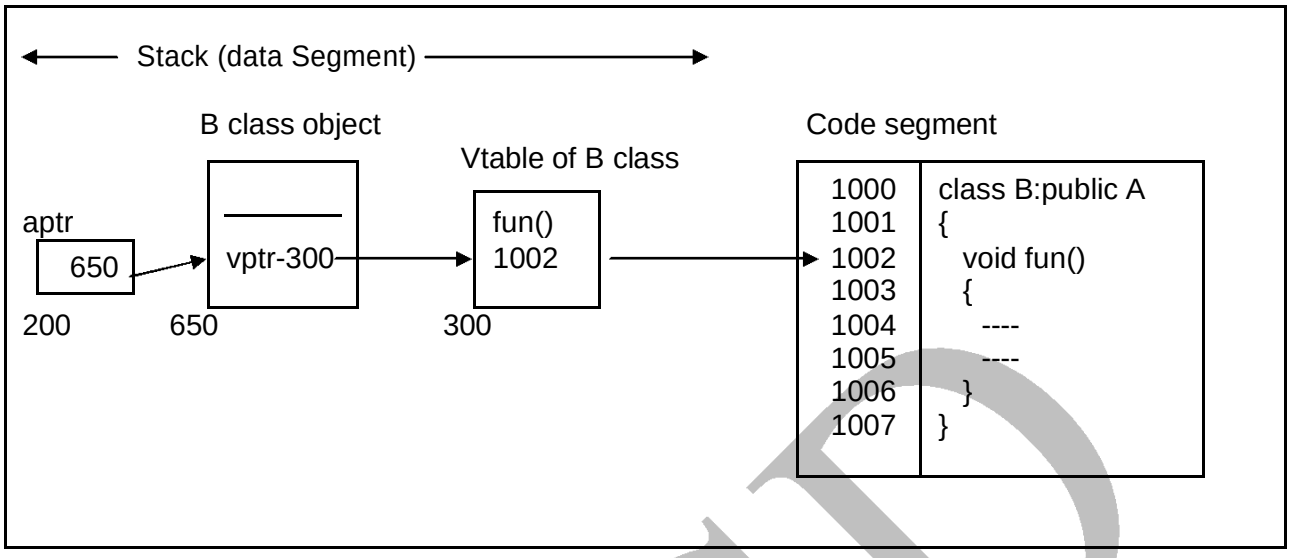
It is a mechanism in which all related objects respond to a generalized message in different manner. It is also known as runtime binging/late binding/dynamic binding. Programmatically, invoking the derived class functions using base class pointer, to make code compact, reusable, and extensible. Polymorphism is achieved through virtual functions. It is achieved only in inheritance. By default compile time binding is performed. But to achieve runtime binding virtual binding is implemented The polymorphism is implemented to design reusable and highly extensible software.



As shown base class A contains function virtual fun(). And its derived class also contains function with the same name. this is called as method overriding.marking the fun() virtual enable run time binding in hierarchy.runtime binding makes it possible to invoke derived class functions through base class pointer.

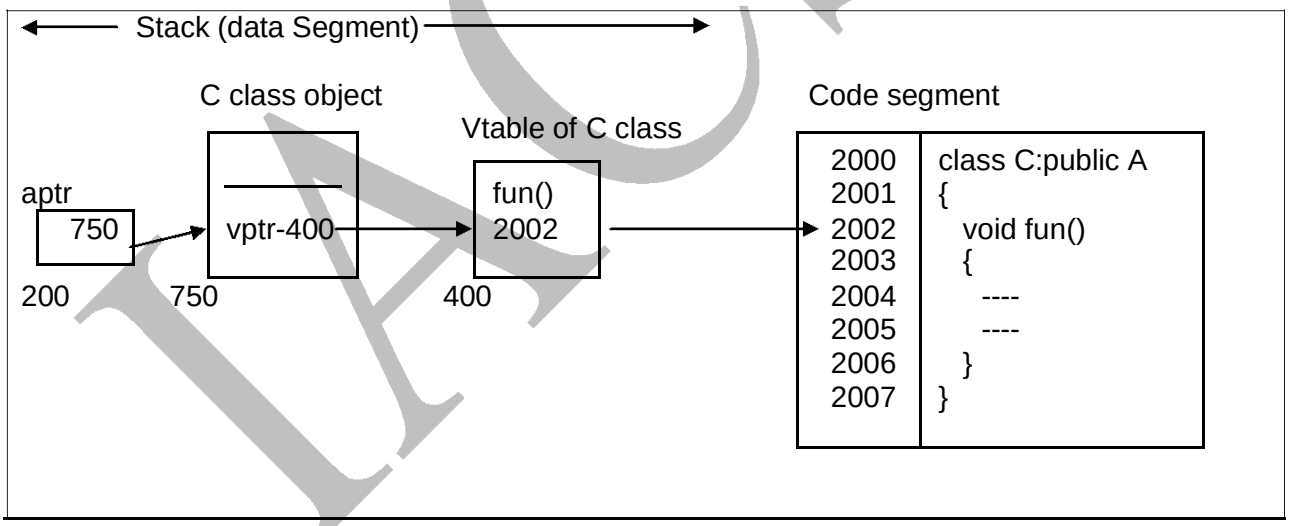
```
A aobj;  //compile time type of aobj is class A
aobj.fun(); //invokes from class A

B bobj;     //compile time type of bobj is class B
bobj.fun();    //invokes from class B

C cobj;      //compile time type of cobj is class C
cobj.fun();    //invokes from class C

A *aptr=new A();     //compile time type of aptr is class A
aptr->fun();   //fun invoked from class A

A * aptr=new B();   // compile time type of aptr is class A
aptr->fun();   //fun invoked from class B

A * aptr=new C();   // compile time type of aptr is class A
aptr->fun();   //fun invoked from class C
```

When a class contains virtual function the hidden data member is added by the compiler to a class i.e. vptr (virtual pointer). Due to this member the size of an object is four bytes more than any other class containing non-virtual functions. This vptr is is a pointer created and maintained by compiler. It points to the vtable i.e. static array of function pointers of corresponding class. i.e. if its an object of B class then the vptr of B class object will point to the static array of function pointers of B class. Same is applied to all derived classes. The vptr plays an important role in runtime biding. The vtable is for every derived class. The function pointers point to the virtual overridden functions on code segment of that particular class.



The above diagram shows how the runtime binding takes place for following statement
A * aptr=new B();
aptr->fun();

The same happens for all derived class and its overridden functions.
Reading the vptr and reading the vtable is the responsibility of runtime. The entire mechanism takes place only due to virtual keyword.



The above diagram shows how the runtime binding takes place for following statement
A * aptr=new C();
aptr->fun();

Note: Polymorphism enhances the extensibility and reusability. Overriding is known as runtime polymorphism and method overloading/operator overloading/method hiding is known asstatic polymorphism.

```
classPrintToScreen
{
public :
      staticvoid PrintEmployeeDetails(Employee *e)
      {
            e->display();
            cout<<"\n The computed salary is ::"<<e->computesalary();
      }
};

int main()
{
      SalesPerson sp1;
      PrintToScreen::PrintEmployeeDetails(&sp1);
      cout<<"\n\n\n";

      SalesPerson sp2("King",45000,12,12,2012,1000,0.10);
      PrintToScreen::PrintEmployeeDetails(&sp2);

      Manager m1;
      PrintToScreen::PrintEmployeeDetails(&m1);
      cout<<"\n\n\n";

      Manager m2("Steven",56000,3,2,2017,1000,1000);
      PrintToScreen::PrintEmployeeDetails(&m2);


      getch();
      return 0;
}
```

The static function`PrintEmployeeDetails()` in the class PrintToScreen is polymorphic in nature as it invokes the function computesalary which is polymorphic.

Run Time Type Identification

RTTI is a process that exposes the information of an object data type at run time during program execution.To implement RTTI there are certain operators that gives the runtime type of an object, performs dynamic casting of an object.

1. typeid
2. dynamic_cast
3. reinterpret_cast

1. typeid operator: this operator is used for fetching the runtime type/dynamic type of a polymorphic object and it can also get the static type.

2. dynamic_cast: It is an operator that is used specifically for down casting. i.e. when the base class pointer need to be type casted to derived class pointer. Down casting is required when we need to invoke special functions of derived class using base class pointer.

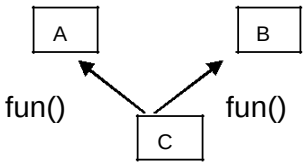
```
class PrintToScreen
{
public :
        static void PrintEmployeeDetails(Employee *e)
        {
                e->display();
                cout<<"\n The computed salary is ::"<<e->computesalary();
                if(typeid(*e)==typeid(SalesPerson)) {

                        SalesPerson * sp=dynamic_cast<SalesPerson*>(e);
                        sp->PayBonus();
                }
        }
};
```

In above example, PayBonus() is special function of class SalesPerson. Base class/generic pointer can invoke only generalized functions. Therefore the special functions cannot be invoked using base class pointer. For reusability and extensibility we need to downcast the base class pointer to derived class pointer. That is done using dynamic_cast operator as shown in above example. dynamic_cast operator is used only to typecast base class pointer.It cannot be used to downcast base class objects.

3. reinterpret_cast: this operator is used for converting one pointer to another pointer of any type. It doesn't check/match the type of pointer. The use of this casting is avoided.

Multiple Inheritance and its ambiguity

In multiple inheritance the derived class possesses the properties and functions of multiple base classes.




```
C obj;
obj.fun(); //compile time error or ambiguity error
```

As class C is inheriting features of A and B. A and B are non related but contains functions with same name fun(). Therefore it becomes ambiguous for above statement, C contains fun() from A as well as B.

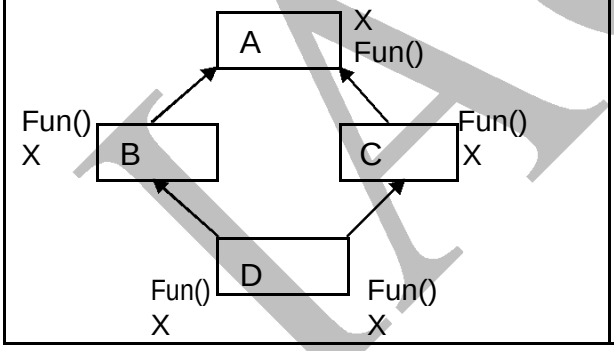
The above ambiguity can be overcome by following ways:

1. Make use of scope resolution operator i.e.

```
C obj;
obj.fun(); // will give ambiguity error at compile time

solution
obj.A::fun();   //invokes fun() from A class
obj.B::fun();   //invokes fun() from B class
```

2. Override the fun() in class C. Hence it will be called from class C instead of calling it from base class.

Diamond Inheritance and ambiguities



The hybrid inheritance that makes a pattern of diamond is diamond inheritance. There are two ambiguities.

1. Function call ambiguity
2. Duplicate data members

As shown in a diagram, the inheritance hierarchy is class D is derived from class B and class C, both of which is derived from a single parent class A. the ambiguities raised due to this pattern are as follows:

1. Function call ambiguity: class B and class C both contains the overridden function fun(), therefore class D contains the two both.

```
D obj;
obj.fun(); //compile time error or ambiguity error
```

As class D is inheriting features of B and C. B and C are contains overridden functions fun() derived from same base class.Therefore it becomes ambiguous for above statement.

The above ambiguity can be overcome by following ways:

a. Make use of scope resolution operator i.e.
   D obj;
   obj.fun(); // will give ambiguity error at compile time

   solution
   obj.B::fun(); //invokes fun() from A class
   obj.C::fun(); //invokes fun() from B class
b. Override the fun() in class D. Hence it will be called from class D instead of calling it from base class.

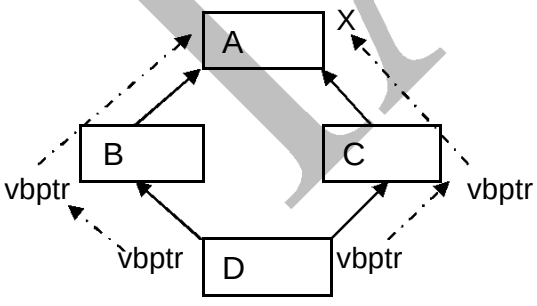
2. Duplicate data member copies: As shown the data member of class A i.e. X is inherited in class B and class C. class D receives X from class B and class C, therefore D contains duplicate copies of X.

   D obj;

   For above statement, the total memory allocated will be the attributes of class B(it contains attributes of A as well as B) , attributes of class C(contains attributes of A as well as C) and special attributes of C. Therefore, there are duplicate copies of class A attributes in the object of class D.

To avoid the above mentioned ambiguities we need to implement virtual base class. When a base class is made virtual , necessary care is taken so that the duplication is avoided regardless of the number of paths that exists to child class. The class B and class C are inherited virtual from class A. Hence the class A is marked as virtual base class in the hierarchy.

```
class Employee{----};
class SalesPerson:public virtual Employee{----};
class Manager:public virtual Employee{----};
class SalesManager:public SalesPerson,public Manager{----};
```



Vbptr is a virtual base class pointer that points to the data members of base class. When A class is virtual base class, the vbptr is created in B and C and D. Hence duplicate copies are avoided. While defining the parameterized constructor of most derived class D the member initialiser list is implemented. In that list we need to invoke parameterized constructor of A,B,C.

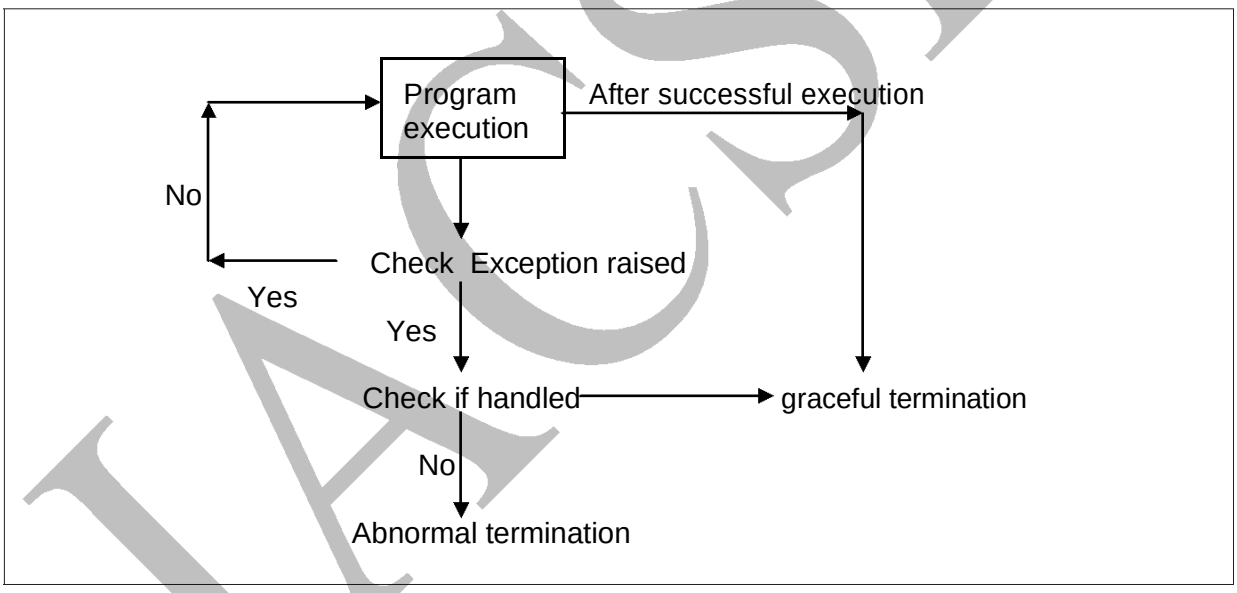
Chapter 5

Exception Handling

Excpetion Handling

Error is defined as any mistake in a program that may prevent the compilation or execution of a program.Errors is broadly categorized as follows:

1. Syntax Errors: These are the errors caused due to incorrect syntax in a program. These errors occur during the compile time and hence it prevents the compilation. These errors can be removed by correcting the syntax.
2. Logical errors: These errors occur due to incorrect logic of program. Undesired/ unexpected output during execution is logical error, it is identified by programmer. These errors can be fixed by giving the correct logic.
3. Linker errors: These errors occur during linking of the program. Caused due to undefined functions, inability to link the function calls. These errors can be fixed by providing the proper definitions, including the necessary library files.
4. Runtime errors: Any exceptional circumstance that causes a program to terminate while executions are runtime errors.These are also known as exceptions. These errors cannot be fixed,but program should contain a code forhandling the exception. Handling the exceptions doesn't give you the desired output, but prevents the abnormal termination of the program, and therefore prevents the degradation of system.



In C-Programming, the exception handling is implemented using if-else. But its limitation are that the business logic and exception handling logic is not separated. Exception is not treated as an object. To overcome these limitations OOPs gives a exception handling mechanism. In this mechanism the actual logic is separated from the exception handling logic. An exception is treated as a bundle of unit containing information of error.

Following are the steps for exception handling:

1. Identify the block of code that is likely to throw an exception. And put that code into try block. Try block is meant for the code that throws an exception.
2. An exception that is raised in try block need to be handled. There is a catch block to provide the exception handling logic.
3. There are multiple possible errors that can be raised in try block, hence there can be multiple catch blocks for single try block.

4. There is general catch to handle all exceptions that are not otherwise handled.

```
#include <iostream>
using namespace std; // Function prototype
double divide(double, double);
int main()
{
int num1, num2;
double quotient;
cout << "Enter two numbers: ";
cin >> num1 >> num2;
try
{
quotient = divide(num1, num2);
cout << "The quotient is " << quotient << endl;
}
catch (char *exceptionString)
{ cout << exceptionString; } cout
<< "End of the program.\n";
return 0;
}
double divide(double numerator, double denominator)
{
if (denominator == 0)
throw "ERROR: Cannot divide by zero.\n";
else return numerator / denominator;
}
```

User Define Exceptions

User defined exceptions or custom exceptions are classes whose objects are treated as run time error to be handled. You can define your own exception classes by inheriting and overriding **exception** class functionality. There is virtual method what() in exception class that we override in our derived class. It Gets string identifying exception.

```
virtual const char* what() const throw();
```

Following is the example, which shows how you can use std::exception class to implement your own exception in standard way

```
class StackException:public exception
{
private:
      char  errorMsg[20];
public:
      StackException(char*);
      const char* what() const throw();  //public method of
```

```
exception class

};

//The objects of StackException class is used in Stack class as
per requirements.

void stack::push(int x)
{
      if (isFull())
      {
            char str[20] = "Stack is Full";
            throw new StackException(str);
      }

      cout << "Inserting " << x << endl;
      arr[++top] = x;
}

//The StackException objects are handled in main.

int main()
{
      stack1 pt(3);
      int ele;
      char choice;
      try
      {
            do
            {
                  cout << "\nEnter element::";
                  cin >> ele;
                  pt.push(ele);
                  cout << "\n Do you wish to enter more elements::";
                  cin >> choice;
            } while (choice == 'y' || choice == 'Y');
      }
      catch (StackException ex)
      {
            cout<<ex.what();
      }
      try
      {

            do
            {
```

```
                cout << "\n peek element is " << pt.peek();
                cout << "\n Popped element is " << pt.pop();
                cout << "\n Do you wish to pop more elements::";
                cin >> choice;
            } while (choice == 'y' || choice == 'Y');
        }
        catch (StackException ex)
        {
            ex.what();
        }
        _getch();
        return 0;
}
```

Chapter 6

Templates

```
void swap(int n1,int n2)
{
    int temp;
    temp=n1;
    n1=n2;
    n2=temp;
}
```

```
void swap(char ch1,char ch2)
{
    char temp;
    temp=ch1;
    ch1=ch2;
    ch2=temp;
}
```

```
void swap(double n1,double n2)
{
    float temp;
    temp=n1;
    n1=n2;
    n2=temp;

}
```

```
void swap(Complex  n1,Complex n2)
{
    Complex temp;
    temp=n1;
    n1=n2;
    n2=temp;

}
```

As shown above, there is function overloading done for swapping the values. The logic of swapping remains same, but just for different data types we need to overload the functions. But overloading the functions unnecessarily increases the length of code, which leads to low maintenance. Therefore we need to write a single generic function that can work for all types.Such kind of programming is known Generic programming.We can have function templates and class templates.

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

template <class T>
void swap(T&a,T&b)      //Function Template
{
   T temp=a;
   a=b;
   b=temp;
}

int main()
{
   int x1=4,y1=7;
   float x2=4.5,y2=7.5;
   cout<<"Before Swap:";
   cout<<"\nx1="<<x1<<"\n y1="<<y1;
   cout<<"\nx2="<<x2<<"\n y2="<<y2;
   swap(x1,y1);
   swap(x2,y2);
```

```
   cout<<“nnAfter Swap:”;
  cout<<“\nx1=”<<x1<<“\ty1=”<<y1;
   cout<<“\nx2=”<<x2<<“\ty2=”<<y2;

   return 0;
}
```

We can also have class templates. Class templates are generally used to implement containers. A class template is instantiated by passing a given set of types to it as template arguments. The member functions in a template class are treated as template functions. Templates enhances the reusability of code for various data types.

```
template <class T>
class Stack
{
     private:
          int size;
          int top;
          T *arr;
     public:
          Stack(int);
          void push(T);
          T pop();
          T peek(); bool
          isFull(); bool
          isEmpty();

};

int main()
{
      Stack<int> s1(5);
      ----
      ----
}
```

Chapter 7

Advanced Operator Overloading

Advanced operator overloading

We can also overload some advance operators.

Subscript [] overloading

```
// Overloading operators for Array
class #include<iostream>
#include<cstdlib>

usingnamespacestd;

// A class to represent an integer
array classArray
{
private:
    int*ptr;
    intsize;
public:
    Array(int*, int);

    // Overloading [] operator to access elements in array
    style int&operator[] (int);

    // Utility function to print contents
    voidprint() const;
};


// Implementation of [] operator.  This function must return a
// refernce as array element can be put on left
side int&Array::operator[](intindex)
{
    if(index >= size)
    {
        cout << "Array index out of bound,
        exiting"; exit(0);
    }
    returnptr[index];
}

// constructor for array class
Array::Array(int*p = NULL, ints = 0)
{
    size = s;
    ptr = NULL;
    if(s != 0)
    {
        ptr = newint[s];
        for(inti = 0; i < s; i++)
            ptr[i] = p[i];
    }
}
```

```
voidArray::print() const
{
    for(inti = 0; i < size; i++)
        cout<<ptr[i]<<" ";
    cout<<endl;
}


Int main()
{
    inta[] = {1, 2, 4, 5};
    Array arr1(a, 4);
    arr1[2] = 6;
    arr1.print();
    arr1[8] = 6;
    return0;
}
```

Friend functions

As we have already performed the operator overloading, it has made the following operations possible.

C3=c1+c2; //c3=c1.operator+(c2);

In the above statement all are objects of complex class. Therefore the operator overloading function for + operator is defined as a member function of complex class.

C3=c1+5;

In above statement, as seen the invocation of the function is on c1 object. Therefore the operator overloading function for operator + is defined as a member function of complex class.

C3=5+c1; //error

In the above statement, the function cannot be called on 5(int) therefore it will give a compile time error. But the operations are performed on int and complex object,hence we need to define the operator overloading function as a non-member function but although it is a non-member function it should have the access of private data members of complex class. This can be achieved with the help of friend functions.
A C++ **friend functions** are special functions which can access the private members of a class. They are considered to be a loophole in the Object Oriented Programming concepts, but logical use of them can make them useful in certain cases. For instance: when it is not possible to implement some function, without making private members accessible in them. This situation arises mostly in case of operator overloading. We need friend functions when operator works with two dissimilar types of operands and the first operands happens to be of basic type.

```
class Complex
{
private:
	float real,imag;
public:
	Complex();
	Complex(float,float);

	friend Complex& operator+(int, Complex&); friend
	ostream& operator<<(ostream&, Complex&); friend
	istream& operator>>(istream&, Complex&);
};
Complex& operator+(int num, Complex& c)
{
	Complex temp;
	temp.real = num + c.real;
	temp.imag = num + c.imag;
	return temp;
}

ostream& operator<<(ostream& out, Complex& c)
{
	out << "\n The complex number is :: " << c.real << "+" <<c.imag << "i";
	return out;
}
istream& operator>>(istream& in, Complex&  c)
{
	cout << "\n Enter real part::";
	in >> c.real;
	cout << "\n Enter imag part::";
	in >> c.imag;

}

int main()
{
	Complex c10(1, 1), c11;
	c11.display();
	c11 = 1 + c10;
	c11.display();

	Complex c12;
	cout<<"\n Enter data for c12-----";
	cin >> c12;
	cout << c12;

	getch();
	return 0;
}
```

Chapter 8

File Handling

File represents storage medium for storing data or information. Streams refer to sequence of bytes. In Files we store data i.e. text or binary data permanently and use these data to read or write in the form of input output operations by transferring bytes of data. So we use the term File Streams/File handling. We use the class `<fstream>`

ofstream: It represents output Stream and this is used for writing in files.
ifstream: It represents input Stream and this is used for reading from files.
fstream: It represents both output Stream and input Stream. So it can read from files and write to files. fstream is parent class of ifstream and ofstream classes.

Operations in File Handling:
Creating a file: `open()`
Reading data: `read()`
Writing new data: `write()`
Closing a file: `close()`

Open a file
The first operation generally performed on an object of one of these classes is to associate it to a real file. This procedure is known as to open a file. An open file is represented within a program by a stream (i.e., an object of one of these classes; in the previous example, this was myfile) and any input or output operation performed on this stream object will be applied to the physical file associated to it.

In order to open a file with a stream object we use its member function open:

open (filename, mode);

Where filename is a string representing the name of the file to be opened, and mode is an optional parameter with a combination of the following flags:

`ios::in`	Open for input operations.
`ios::out`	Open for output operations.
`ios::binary`	Open in binary mode.
`ios::ate`	Set the initial position at the end of the file. If this flag is not set, the initial position is the beginning of the file.
`ios::app`	All output operations are performed at the end of the file, appending the content to the current content of the file.
`ios::trunc`	If the file is opened for output operations and it already existed, its previous content is deleted and replaced by the new one.

All these flags can be combined using the bitwise operator OR (|). For example, if we want to open the file example.bin in binary mode to add data we could do it by the following call to member function open:

```
ofstream myfile;
myfile.open ("example.bin", ios::out | ios::app | ios::binary);
```

Each of the open member functions of classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

For ifstream and ofstream classes, ios::in and ios::out are automatically and respectively assumed, even if a mode that does not include them is passed as second argument to the open member function (the flags are combined).

For fstream, the default value is only applied if the function is called without specifying any value for the mode parameter. If the function is called with any value in that parameter the default mode is overridden, not combined.

File streams opened in binary mode perform input and output operations independently of any format considerations. Non-binary files are known as text files, and some translations may occur due to formatting of some special characters (like newline and carriage return characters).

Since the first task that is performed on a file stream is generally to open a file, these three classes include a constructor that automatically calls the open member function and has the exact same parameters as this member. Therefore, we could also have declared the previous myfile object and conduct the same opening operation in our previous example by writing:

ofstream myfile ("example.bin", ios::out | ios::app | ios::binary);

Combining object construction and stream opening in a single statement. Both forms to open a file are valid and equivalent.

To check if a file stream was successful opening a file, you can do it by calling to member is_open. This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise: if (myfile.is_open()) { /* ok, proceed with output */ }

Closing a file

When we are finished with our input and output operations on a file we shall close it so that the operating system is notified and its resources become available again. For that, we call the stream's member function close. This member function takes flushes the associated buffers and closes the file:

myfile.close();

Once this member function is called, the stream object can be re-used to open another file, and the file is available again to be opened by other processes.

In case that an object is destroyed while still associated with an open file, the destructor automatically calls the member function close.

Reading and Writing

Following are the functions for reading and writing the data from file.

get()	Read a single character from a file
put()	write a single character in file.
read()	Read data from file
write()	Write data into file.

Checking state flags

The following member functions exist to check for specific states of a stream (all of them return a `bool` value):

`bad()`
Returns `true` if a reading or writing operation fails. For example, in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.
`fail()`
Returns `true` in the same cases as `bad()`, but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.
`eof()`
Returns `true` if a file open for reading has reached the end.
`good()`
It is the most generic state flag: it returns `false` in the same cases in which calling any of the previous functions would return `true`. Note that `good` and `bad` are not exact opposites (`good` checks more state flags at once).

The member function `clear()` can be used to reset the state flags.

Random Access of File

get and put stream positioning
All i/o streams objects keep internally -at least- one internal position:

`ifstream`, like `istream`, keeps an internal *get position* with the location of the element to be read in the next input operation.

`ofstream`, like `ostream`, keeps an internal *put position* with the location where the next element has to be written.

Finally, `fstream`, keeps both, the *get* and the *put position*, like `iostream`.

These internal stream positions point to the locations within the stream where the next reading or writing operation is performed. These positions can be observed and modified using the following member functions:

tellg() and tellp():
These two member functions with no parameters return a value of the member type `streampos`, which is a type representing the current get position (in the case of `tellg`) or the put position (in the case of `tellp`).

seekg() and seekp()
These functions allow to change the location of the get and put positions. Both functions are overloaded with two different prototypes. The first form is:

```
seekg ( position );
seekp ( position );
```

Using this prototype, the stream pointer is changed to the absolute position `position` (counting from the beginning of the file). The type for this parameter is `streampos`, which is the same type as returned by functions `tellg` and `tellp`.

The other form for these functions is:

```
seekg ( offset, direction );
seekp ( offset, direction );
```

Using this prototype, the get or put position is set to an offset value relative to some specific point determined by the parameter `direction`. `offset` is of type `streamoff`. And `direction` is of type `seekdir`, which is an enumerated type that determines the point from where offset is counted from, and that can take any of the following values:

`ios::beg`	offset counted from the beginning of the stream
`ios::cur`	offset counted from the current position
`ios::end`	offset counted from the end of the stream

```
#include"Student.h"
#include<conio.h>
#include<fstream>
class FileIO
{
public:
        static void WriteRecords()
        {
                char choice;
                fstream fs;
                fs.open("StudentData.dat", ios::out);
                do
                {
                        Student s1;
```

```
                        s1.accept();
                        fs.write((char*)&s1, sizeof(Student));
                        cout << "\n Do you want to insert more records::";

                        cin >> choice;
                } while (choice == 'y' || choice == 'Y');
                fs.close();
        }
        static void ReadRecords()
        {
                fstream fs;
                fs.open("StudentData.dat", ios::in);
                Student s2;
                while (fs.read((char*)&s2, sizeof(Student)))
                {
                        s2.display();
                }
                fs.close();
        }
        static bool SearchRecord(int rno)
        {
                Student s;
                fstream fs;
                fs.open("StudentData.dat", ios::in);
                while (fs.read((char*)&s, sizeof(Student)))
                {
                        if (s.getRollNo() == rno)
                        {
                                s.display();
                                return true;
                        }
                }
                fs.close();
                return false;

        }
};
int main()
{
        char wish;
        int menuchoice;
        do
        {
                cout << "\n 1. Add records in file"
                        << "\n 2. Read all records in file"
                        << "\n 3. Search Record in file"
                        << "\n 4. exit";
                cout << "\n Enter the choice";
                cin >> menuchoice;
                switch (menuchoice)
                {
```

```
                case 1:
                        FileIO::WriteRecords();
                        break;

               case 2:
                        FileIO::ReadRecords();
                        break;
                case 3:
                        int rollno;
                        cout << "\n Enter the roll number to search::";
                        cin >> rollno;
                        bool flag = FileIO::SearchRecord(rollno);
                        if (flag == true)
                        {
                                cout << "\n Record present";
                        }
                        else
                        {
                                cout << "\n No record found!!!";
                        }
                }
                cout << "\n Do u wish to perform operations::";
                cin >> wish;
        } while (wish == 'y' || wish ==
        'Y'); _getch();
        return 0;
}
```

Chapter 9

Advanced Polymorphism

Types of Classes

1. **Concrete Class:** It is an ordinary class that contains the relevant data members and functions. This class can be instantiated(object can be created) and we can execute the given functionalities to access and modify data.
2. **Polymorphic class:** it is a class that contains generalized data members and member functions amongst which at least one function is virtual. Polymorphic class is created when we want achieve inheritance along with runtime binding. This class can be instantiated if needed.
3. **Abstract class**: It is the class that contains all generalized data members and member functions. Amongst these member functions at least one pure virtual function. A pure virtual function is any virtual function that doesn't have implementation (that is denoted by giving pure specifier i.e. =0). Any concrete class that is inherited from abstract class will provide the implementation to this pure virtual function i.e. override the pure virtual functions. If the derived class doesn't override the function then the derived class also becomes abstract class. Abstract classes act as expressions of general concepts from which more specific classes can be derived. You cannot create an object of an abstract class type; however, you can use pointers and references to abstract class types. The main purpose of abstract class is to create a strong platform for inheritance and polymorphism.
4. **Pure abstract class**: When a class contains data members and all pure virtual functions it becomes pure abstract class. By creating pure abstract class we achieve high level of abstraction and encapsulation. It enhances high reusability and extensibility.

```
#include<iostream>
using namespace std;
#include<conio.h>
class Shape
{
protected:
	double area;
public:
	Shape()
	{
		this->area = 0.0;
	}
	virtual double CalculateArea() = 0;
};

class Circle : public Shape
{
private:
	double radius;
public:
	Circle()
	{
		this->radius = 0.0;
	}
	Circle(double radius)
	{
		this->radius = radius;
```

```
        }
        double CalculateArea()
        {
                this->area = 3.14f*this->radius*this->radius;
                return this->area;
        }
};

class Square :public Shape
{
private:
        double side;
public:
        Square()
        {
                this->side = 0.0;
        }
        Square(double side)
        {
                this->side = side;
        }

        double CalculateArea()
        {
                this->area = this->side*this->side;
                return this->area;
        }
};

class PrintToScreen
{
public:
        static void getAreaofShape(Shape *shape)
        {
                cout<<"\n The area is ::"<<shape->CalculateArea();
        }
};

int main()
{
        Circle c1(3);
        Square s1(2);

        PrintToScreen::getAreaofShape(&c1);
        PrintToScreen::getAreaofShape(&s1);

        _getch();
}
```

Why don't we have virtual constructors?

A virtual call is a mechanism to get work done given partial information. In particular, "virtual" allows us to call a function knowing only an interfaces and not the exact type of the object. To create an object you need complete information. In particular, you need to know the exact type of what you want to create. Consequently, a "call to a constructor" cannot be virtual.

Why are destructors not virtual by default?

Because many classes are not designed to be used as base classes. Virtual functions make sense only in classes meant to act as interfaces to objects of derived classes (typically allocated on a heap and accessed through pointers or references).

So when should I declare a destructor virtual? Whenever the class has at least one virtual function. Having virtual functions indicate that a class is meant to act as an interface to derived classes, and when it is, an object of a derived class may be destroyed through a pointer to the base. For example:

```
class Base {
        // ...
        virtual ~Base();
};
class Derived : public Base {
        // ...
        ~Derived();
};

void f()
{
        Base* p = new Derived;
        delete p;       // virtual destructor used to ensure that ~Derived is called
}
```

Had Base's destructor not been virtual, Derived's destructor would not have been called - with likely bad effects, such as resources owned by Derived not being freed.

What is object slicing?

In C++, a derived class object can be assigned to a base class object, but the other way is not possible.

```
class Base { int x, y; };

class Derived : public Base { int z, w; };

int main()
{
   Derived d;
```

```
    Base b = d; // Object Slicing,  z and w of d are sliced off
}
```

Object slicing happens when a derived class object is assigned to a base class object, additional attributes of a derived class object are sliced off to form the base class object.

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

class Base
{
protected:
    int i;
public:
    Base(int a)     { i = a; }
    virtual void display()
    { cout << "I am Base class object, i = " << i << endl; }
};

class Derived : public Base
{
    int j;
public:
    Derived(int a, int b) : Base(a) { j = b; }
    virtual void display()
    { cout << "I am Derived class object, i = "
          << i << ", j = " << j << endl;  }
};

// Global method, Base class object is passed by value
void somefunc (Base obj)
{
    obj.display();
}

int main()
{
    Base b(33);
    Derived d(45, 54);
    somefunc(b);
    somefunc(d);  // Object Slicing, the member j of d is sliced off
    return 0;
}
```

Output:

```
I am Base class object, i = 33
```

I am Base class object, i = 45

We can avoid above unexpected behavior with the use of pointers or references. Object slicing doesn't occur when pointers or references to objects are passed as function arguments since a pointer or reference of any type takes same amount of memory. For example, if we change the global method myfunc() in the above program to following, object slicing doesn't happen.

```
// rest of code is similar to above
void somefunc (Base &obj)
{
   obj.display();
}
// rest of code is similar to above
```

Output:
I am Base class object, i = 33
I am Derived class object, i = 45, j = 54
We get the same output if we use pointers and change the program to following.

```
// rest of code is similar to above
void somefunc (Base *objp)
{
   objp->display();
}

int main()
{
   Base *bp = new Base(33) ;
   Derived *dp = new Derived(45, 54);
   somefunc(bp);
   somefunc(dp);  // No Object Slicing
   return 0;
}
```

Output:
I am Base class object, i = 33
I am Derived class object, i = 45, j = 54

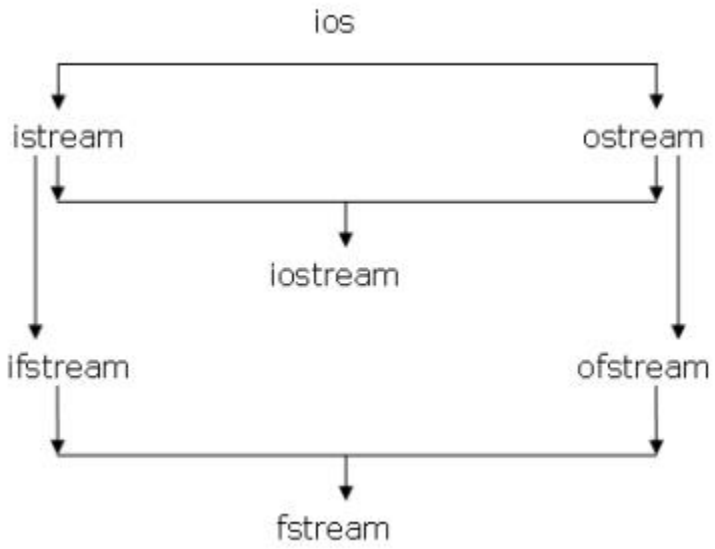
Object slicing can be prevented by making the base class function pure virtual there by disallowing object creation. It is not possible to create the object of a class which contains a pure virtual method.

Chapter 10

Console I/O

C++ I/O Features

Its important to understand the word stream before we actually begin. Stream indicates a flow from source to destination. In case of computers it's a buffer where the flow of data(bits) from one device to another takes place. C++ has its own Object Oriented way of handling data input and output. All the input/output operations are carried out using different stream classes and their related functions. A stream is a logical device that produces and consumes information.



Stream class for Console I/O

Class Hierarchy

ios is the topmost base class in the hierarchy of stream classes. This class contains features that are common to all streams viz. flags for formatting the stream data, error status flags, file operations mode etc.. Even though we don't use ios flags directly, we use many of its inherited member functions and data members. The classes istream and ostream are derived from this class to support input and output respectively.

C++ language provides us console input/output functions. As the name says, the console input/output functions allow us to -

- Read the input from the keyboard, entered by the user at the console.
- Display the output to the user at the console.

Note : These input and output values could be of *any primitive data type*.

There are two kinds of console input/output functions :

No.	Functions
1	*Formatted input/output functions.*
2	*Unformatted input/output functions.*

Unformatted console input output operations

These input / output operations are in unformatted mode. The following are operations of unformatted console input / output operations:

A) void get()
It is a method of cin object used to input a single character from keyboard. But its main property is that it allows wide spaces and newline character.

Syntax:
char c=cin.get();
Example:

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

int main()
{
        char c=cin.get();
        cout<<c<<endl;

        return 0;
}
```

Output
I
I

B) void put()
It is a method of cout object and it is used to print the specified character on the screen or monitor.

Syntax:
cout.put(variable / character);

Example:

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

int main()
{
        char c=cin.get();
        cout.put(c); //Here it prints the value of variable c;
        cout.put('c'); //Here it prints the character 'c';

        return 0;
}
```

Output

```
I
Ic
```

C) getline(char *buffer,int size)

This is a method of cin object and it is used to input a string with multiple spaces.

Syntax:

```
char x[30];
cin.getline(x,30);
```

Example:

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

int main()
{
        cout<<"Enter name :";
        char c[10];
        cin.getline(c,10); //It takes 10 charcters as input;
        cout<<c<<endl;

        return 0;
}
```

Output

```
Enter name :Divyanshu
Divyanshu
```

D) write(char * buffer, int n)

It is a method of cout object. This method is used to read n character from buffer variable.

Syntax:

cout.write(x,2);
Example:

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

int main()
{
        cout<<"Enter name : ";
        char c[10];
        cin.getline(c,10); //It takes 10 charcters as input;
        cout.write(c,9); //It reads only 9 character from buffer c;

        return 0;
}
```

Output
Enter name : Divyanshux
Divyanshu

E) cin

It is the method to take input any variable / character / string.

Syntax:
cin>>variable / character / String / ;
Example:

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

int main()
{
   int num;
   char ch;
   string str;
```

```
    cout<<"Enter Number"<<endl;
    cin>>num; //Inputs a variable;
    cout<<"Enter Character"<<endl;
    cin>>ch; //Inputs a character;
    cout<<"Enter String"<<endl;
    cin>>str; //Inputs a string;

    return 0;
}
```

Output

```
Enter Number
07
Enter Character
h
Enter String
Deepak
```

F) cout

This method is used to print variable / string / character.

Syntax:

```
cout<< variable / charcter / string;
```

Example:

```
#include<iostream>
using namespace std;
int main()
{
    int num=100;
    char ch='X';
    string str="Deepak";

    cout<<"Number is "<<num<<endl; //Prints value of variable;
    cout<<"Character is "<<ch<<endl; //Prints character;
    cout<<"String is "<<str<<endl; //Prints string;

    return 0;
}
```

Output

```
Number is 100
Character is X
String is Deepak
```

Formatted console input output operations

In formatted console input output operations we uses following functions to make output in perfect alignment. In industrial programming all the output should be perfectly formatted due to this reason C++ provides many function to convert any file into perfect aligned format. These functions are available in header file <iomanip>. iomanip refers input output manipulations.

A) width(n)
This function is used to set width of the output.
Syntax:
cout<<setw(int n);
Example:

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

int main()
{
        int x=10;
        cout<<setw(20)<<variable;

        return 0;
}
```

Output

```
          10
```

B) fill(char)
This function is used to fill specified character at unused space.
Syntax:
cout<<setfill('character')<<variable;
Example:

```
#include<iostream>
#include<iomanip>
using namespace std;
int main()
{
        int x=10;
        cout<<setw(20);
        cout<<setfill('#')<<x;

        return 0;
}
```

Output

```
##################10
```

C) precison(n)
This method is used for setting floating point of the output.
Syntax:
cout<<setprecision('int n')<<variable;
Example:

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

int main()
{
        float x=10.12345;
        cout<<setprecision(5)<<x;

        return 0;
}
```

Output
10.123

D) setflag(arg 1, arg,2)
This function is used for setting format flags for output.
Syntax:
setiosflags(argument 1, argument 2);

E) unsetflag(arg 2)
This function is used to reset set flags for output.
Syntax:
resetiosflags(argument 2);

C++ Manipulators

- Manipulators are operators used in C++ for **formatting output**. The data is manipulated by the programmer's choice of display.
- **endl** manipulator, **setw** manipulator, **setfill** manipulator and setprecision manipulator are all explained along with syntax and examples.
-

endl Manipulator:

- This manipulator has the same functionality as the 'n' newline character.
  For example:

```
cout<<"Exforsys"<<endl;
cout << "Training";
```

setw Manipulator:

- This manipulator sets the minimum field width on output.

Syntax:

setw(x)

- Here setw causes the number or string that follows it to be printed within a field of x characters wide and x is the argument set in setw manipulator.
- The header file that must be included while using setw manipulator is .

Example:

```
#include <iostream>
#include <iomanip>

void main( )
{
int x1=123,x2= 234, x3=789;
cout << setw(8) << "Exforsys" << setw(20) << "Values" << endl
<< setw(8) << "test123" << setw(20)<< x1 << endl
<< setw(8) << "exam234" << setw(20)<< x2 << endl
<< setw(8) << "result789" << setw(20)<< x3 << endl;
}
```

Output:

```
test          123
exam              234
result            789
```

setfill Manipulator:

- This is used after setw manipulator.
- If a value does not entirely fill a field, then the character specified in the setfill argument of the manipulator is used for filling the fields.

Example:

```
#include <iostream>
#include <iomanip>
void main()
{
cout << setw(15) << setfill('*') << 99 << 97 << endl;
}
```

Output:

```
********9997
```

setprecision Manipulator:

- The setprecision Manipulator is used with floating point numbers.
- It is used to set the number of digits printed to the right of the decimal point.
- This may be used in two forms:

1. fixed
2. scientific

- These two forms are used when the keywords fixed or scientific are appropriately used before the setprecision manipulator.
- The keyword fixed before the setprecision manipulator prints the floating point number in fixed notation.
- The keyword scientific, before the setprecision manipulator, prints the floating point number in scientific notation.

Example:

```
#include <iostream>
#include <iomanip>
void main( )
{
float x = 0.1;
cout << fixed << setprecision(3) << x << endl;
cout << scientific << x << endl;
}
```

Output:
0.100
1.000e-001

- The first cout statement contains fixed notation and the setprecision contains argument 3.
- This means that three digits after the decimal point and in fixed notation will output the first cout statement as 0.100. The second cout produces the output in scientific notation.
- The default value is used since no setprecision value is provided.

Chapter 11

Miscellaneous

Executing CPP Program on Linux/Ubuntu

Run a C/C++ program on terminal using gcc compiler

Follow these steps to run programs on terminal:
Step 1. Open terminal.

Step 2. Type command to install gcc or g++ complier:

$ sudo apt-get install build-essential

This will install the necessary C/C++ development libraries for your Ubuntu to create C/C++ programs.
To check gcc version type this command:

$ gcc –version or gcc –v

Step 3. Now go to that folder where you will create C/C++ programs. I am creating my programs in Documents directory. Type these commands:

$ cd Documents/
$ sudo mkdir programs
$ cd programs/

Step 4. Open a file using any editor.

$ sudo gedit first.c (for C programs)
$ sudo gedit hello.cpp (for C++ prgrams)

Step 5. Add this code in the file:

(i). C program code:

```
#include<stdio.h>
int main()
{
        printf("\n\nWelcome to my Homepage!!\n\n");
        return 0;
}
```

(i). C++ program code:

```
#include<iostream>
using namespace std;
int main()
{
        cout<<"\n\nHello World,\nWelcome to my first C ++ program on Ubuntu
        Linux\n\n"<<endl;
      return 0;
}
```

Step 6. Save the file and exit.

Step 7. Compile the program using any of the following command:

(i). Compiling C program.

$ sudo gcc first.c
It will create an executable file with ".out" extension named as "a.out".
Or
$ sudo gcc –o first first.c

Where **first** is the executable or object file of **first.c** program.

(ii). Compiling C++ program.

$ sudo g++ hello.cpp (or)
$ sudo g++ -o hello hello.cpp

[Note: Make sure you are in the same directory where you have created your program before compiling it.]

Step 8. To run this program type this command:

(i). For running C program
$ ./a.out (If you compiled using first command)
Or
$ ./first (If you compiled using second command)

(ii). For running C++ program
$ ./a.out (If you compiled using first command)
Or
$ ./hello (If you compiled using second command)
It will show output on the terminal.

C++ Programming Default Arguments (Parameters)

In C++ programming, you can provide default values for function parameters.
The idea behind default argument is simple. If a function is called by passing argument/s, those arguments are used by the function.
But if the argument/s are not passed while invoking a function then, the default values are used.
Default value/s are passed to argument/s in the function prototype.

Working of default arguments

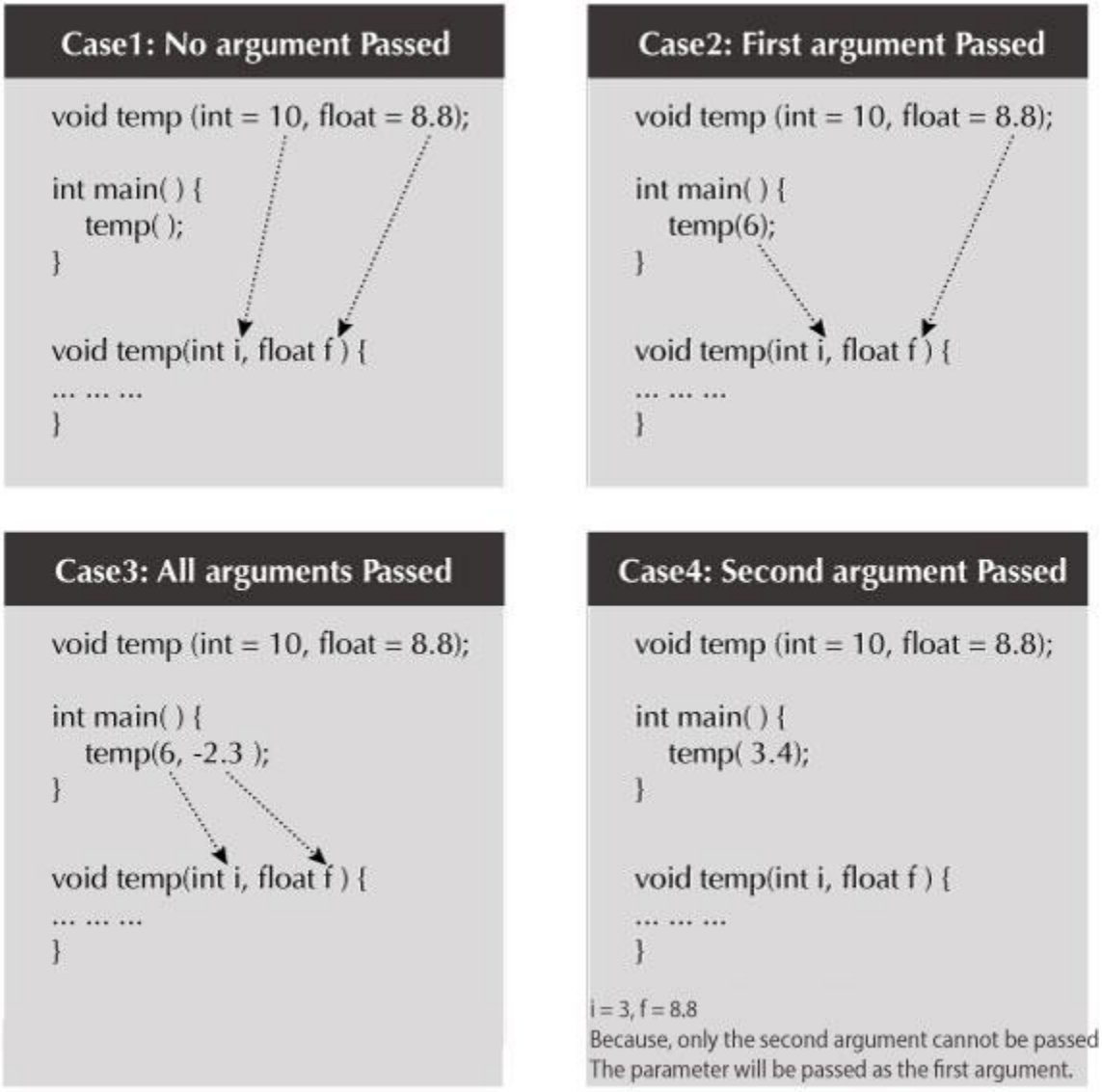


Figure: Working of Default Argument in C++

Example: Default Argument

```
1.  // C++ Program to demonstrate working of default argument
2.
3.  #include <iostream>
4.  using namespace std;
5.
6.  void display(char = '*', int = 1);
7.
8.  int main()
9.  {
10.     cout << "No argument passed:\n";
11.     display();
12.
13.     cout << "\nFirst argument passed:\n";
14.     display('#');
15.
16.     cout << "\nBoth argument passed:\n";
17.     display('$', 5);
18.
19.     return 0;
20. }
21.
22. void display(char c, int n)
23. {
24.     for(int i = 1; i <= n; ++i)
25.     {
26.         cout << c;
27.     }
28.     cout << endl;
29. }
```

Output

```
No argument passed:
*

First argument passed:
#

Both argument passed:
$$$$$
```

In the above program, you can see the default value assigned to the arguments void display(char = '*', int = 1);.

At first, display() function is called without passing any arguments. In this case, display() function used both default arguments c = * and n = 1.

Then, only the first argument is passed using the function second time. In this case, function does not use first default value passed. It uses the actual parameter passed as the first argument c = # and takes default value n = 1 as its second argument.
When display() is invoked for the third time passing both arguments, default arguments are not used. So, the value of c = $ and n = 5.

Common mistakes when using Default argument

1. void add(int a, int b = 3, int c, int d = 4);

   The above function will not compile. You cannot miss a default argument in between two arguments.
   In this case, c should also be assigned a default value.

2. void add(int a, int b = 3, int c, int d);

   The above function will not compile as well. You must provide default values for each argument after b.
   In this case, c and d should also be assigned default values.
   If you want a single default argument, make sure the argument is the last one. void add(int a, int b, int c, int d = 4);

3. No matter how you use default arguments, a function should always be written so that it serves only one purpose.
   If your function does more than one thing or the logic seems too complicated, you can use Function overloading to separate the logic better.

Points to remember:
Constructors and destructors can be made inline.But it should be avoided.
Default access specifier of class members and function is private.
Default mode of inheritance is private.
Const members of class can be initialized through member iniltializer list.
C++ supports structure, which can contain methods too. Structure member can only be public. It can contain only parameterized constructor. No inheritance support.