

Object Oriented Programming

Objective

- What is Object?
- Object oriented Programming?
- Major pillars of Object Oriented programming.
- Real Life examples on OOP.

Need of Object Oriented Programming

Software is Inherently Complex.

- Impedance mismatch between user of a system and it's developer.
- Changing Requirements during development.
- Difficulty of managing software development process. It's a team effort.
- Easy User Interface.
- Clients want systems to be adaptable and Extensible

Object Oriented Approach

- **An object oriented software is composed of discrete objects interacting with each other to give rise to the overall (complex) behavior of the system.**
- **A Program is a bunch of Objects telling each other what to do by sending messages.**
- **Object oriented approach helps to handle the complexity of software development and aids in generation of adaptable and Extensible Systems.**

Object

- An Object is an entity which has well defined structure and behavior.
- An Object has certain characteristic as
 - 1.State
 - 2.behavior
 - 3.Identity
 - 4.Responsibility
- Objects can be tangible ,intangible or conceptual.

For e.g.

Harddisk,person,printer,bankaccount,signal,transaction,sound,air etc.

Car as Object



Attributes

Name : Lightning McQueen
Chasis No : A1B2C3X45
Passing No : MVV 5891
Engine Type : M10
Color : Red
Price : \$XXXXXX
Fuel Type : Petrol
Speed :100

Behavior

Start()
Stop()
Move()
ShiftGear()

Object Properties

➤ State

- Every Object has State
- State of an Object is defined by the value of it's attributes.

➤ Behavior

- Behavior are the functionalities provided by an Object.
- Ideally state of an object should only be changed through it's behavior.

➤ Identity

- One of the attribute of an Object will be used to distinguish an Object from other Objects.

➤ Responsibility

- The role an Object plays in the system.

Bank Account object

Attributes	Behavior	State	Identity	Responsibility
Balance	Open	Balance=10000	Account Number unique=1011	Keep track of stored money with the facility of deposits and withdrawal.
Interest Rate	Balance Enquiry	Interest Rate=2%		Keep locker safely for all users
	Withdraw			
Account Number	Generate Report	Account number=1011		

Abstraction

- Abstraction is the process of identifying key aspects of an entity and ignoring the rest.
- Only those aspects are selected that are important to current problem scenario.
- For e.g

Person Object can be used to develop various application like.

- Student attendance system
- Health care industry
- Payroll system

Encapsulation

- Encapsulation is the mechanism used to hide the data, internal structure and Implementation details of an object.
- Interaction with the object through public interface.
- The user know only about the interface,any changes made to the implementation does not affect the user.

For eg.TV set(buttons and Remote),fan(regulator)

Mobile ,set top box etc.

Inheritance

- Inheritance is the process by which one object can acquire properties of another object.
- “is-a” kind of relationship.

Polymorphism

- Ability of different types of related objects to respond to the same message differently is called polymorphism.

Class

- Using classes we define the prototype for Object.
- A class is a template using which we can create multiple objects of the same type.
- Encapsulates data (attributes) and functions (behavior) into packages called classes.

Class

- Classes are user-defined (programmer-defined) types.
 - Data (data members)
 - Functions (member functions or methods)
- Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an instance of a class.
- Objects of same class have identical structure and operations defined in the class.
- Map real world entities into classes through data members & member functions.

Class

```
class Atm
{
    int accountNumber;
    char name[20];
    float balance;
    public:
    insertCard();
    showInformation();
    withdraw();
    balanceEnquiry();
}
```

- You have created a class and now you can create its objects.
- You can perform different activities like balance enquiry, withdraw money only if you create object of this class.
- When you complete your transaction, you are destroying the object.

End

Introduction to c++

Objective

- History of C++
- Difference between C & C++
- Describe cout / cin
- Reference Variable
- Const Keyword
- Dynamic Memory Allocation
- Inline Functions
- Function Overloading and Default Arguments

History of C++

- C++ is Object Oriented Programming Language.
- C++ was invented by Bjarne Stroustrup in 1979 at Bell Labs.
- Initially he called the new Language as 'C' with classes.
- Later in 1983 the name was changed to C++. ("++" being the Increment operator in C). Rick Mascitti
- 'C' was used as the base language for inventing C++.
- New features like overloading, exception handling, RTTI, templates etc were added.
- STL introduced by Alexander Stepanov.



C++ Language Fundamentals

- C++ is based on C language
- It enhances C by adding new features, as well as correcting certain loopholes in the type checking.
- C++ is C-compatible.
- Provides smooth transition from C to C++.

C++ Language Fundamentals

- C Code can compile under the C++ compiler, with few minor changes.
- Supports modularity and code reusability.
- C++ incorporates OO methodologies as well as the generic programming paradigm.

Cin and cout

Multiple input output can be cascaded.

```
Cin>>x>>y>>z;
```

```
Cout<<x<<y<<z;
```

Role of endl: Adds a new line character

```
Cout<<x<<y<<z<<endl;
```

Bool type

- C++ bool type can have two pre defined constant values i.e.
- true- is predefined value 1
- false - is predefined value 0

```
int main()
{
    bool flag;
    cout<<"Enter true or false "<<endl;
    cin>>flag;
    if(flag==true)
        cout<<"flag is true"<<true<<endl;
    else
        cout<<"flag is false "<<false<<endl;
    Return 0
}
```

Reference Variable

- Concept
- Pass by reference
- Rules

Reference Variable

Reference is an alias created for the variable.

Like nickname of a person

```
int a = 10;
```

```
int & b = a;    // here & acts as a reference operator
```

```
cout << "value is" << a ; // outputs 10
```

```
cout << "value is" << b ; // outputs 10
```

```
cout << "Address is " << &a; // outputs 1000 (say)
```

```
cout << "Address is " << &b; // outputs 1000
```

- Variable a and ref b are exactly same.
- Any operation performed on one reflects to another.

Points to note....

- References have to be initialized.
- No memory is allocated to references.
- Local variables should never be returned by reference
- Array of reference variables can't be created.

```
int *ptr=&num1;  
int num2;  
Ptr=&num2
```

Allowed

```
int &ref=num;  
int num2;  
int &ref=num2;
```

Not Allowed

Reference and Pointer

Pointer	Reference
Flexible connection	Rigid connection
Memory allocated for pointers	No memory allocated
Array of pointers possible	Array of reference impossible

What Will be the output of below code?

```
int i= 5;
int & ri = i;
int j = 20;
ri = j;      // ???
//Does ri refer to j now? What are the values of i,ri & j?
j = 30;
// And what are the values of i, ri & j now?
```

Pass by Reference

```
void SwapRef( int& a, int& b)
{
    int temp;
    temp=a;
    SwapRef(x,y);
    a=b;
    b=temp;
}
```

```
int main ()
{
    int x=10,y=20;

    cout<<"X :"<<x;
    cout<<"y :"<<y;
}
```

const keyword

- #define allows to create constants

Not type-safe

- The const keyword specifies that a variable's value is constant and tells the compiler to prevent the programmer from modifying it.

```
const int num=5;  
num=10 ;  
Num++ } //Error
```

- The above statement defines **num** to be a constant variable initialized with the value of 5.
- Constant should be initialized .Constant can not appear on LHS of Assignment operator.

const keyword

While using const know ...

- Constants should be initialized
- Constant variable cannot be there on LHS of = operator once initialized.
- Use const keyword instead of
- # define preprocessor directive since it is type safe

Reference and Const

- Consider the following **const** variable definition

```
const int i = 5;
```

Are the following statements valid?

```
int* pi = &i;
```

```
int& ri = i;
```

Reference and Const

- We can assign a const address only to a pointer to a const

```
const int* pi = &i;
```

pi is a pointer to a constant integer.

- An attempt to change i through pi will result in a compile-time error:

```
*pi = 7;          // compile time error
```

Given a pointer to a constant object, you cannot modify the pointed object through it.

Reference to const

- Similarly, we can assign a **const** only to a **reference to a const variable**:

const int& ri = i;

ri is a reference to a constant integer.

- An attempt to change **ri** will result in a compile-time error:
ri = 7; Compile error: l-value specifies const
- You cannot modify a reference to a constant variable.
- Now the **const** variable is protected from modifications through pointer / reference as well.

Const in function prototype

```
f(int & id)
```

```
{
```

```
    id = 10;//should not happen
```

```
}
```

```
void main (void)
```

```
{
```

```
    int empid=1;
```

```
    f(empid);
```

```
}
```

- Whenever requirement is original contents should not be changed use const keyword

Inline Functions

```
#define MAX(a,b) (a>b ?a:b)
```

Not Type Safe

Function call Overheads

```
int max(int n1,int n2)
{
    return(n1>n2?n1:n2)
}
```

```
inline int max(int n1,int n2)
{
    return (n1>n2?n1:n2);
}
```

faster as well as
type safe

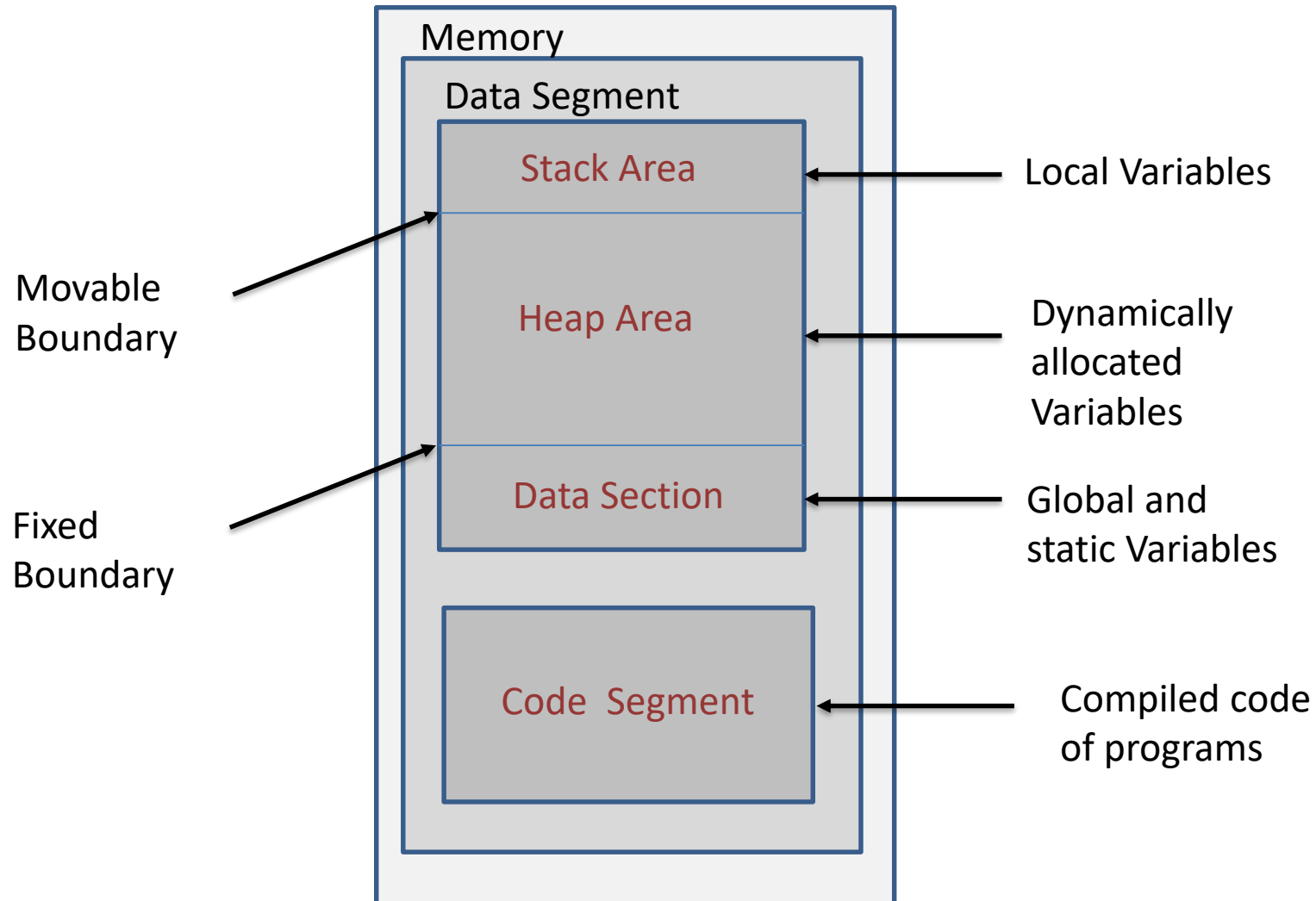
Inline Functions

While using inline function you should know

- inline is a keyword.
- For every instance of a function call the function. Definition gets substituted.
- inline is a request to compiler and not instruction.
- Function substitution occurs only at compiler's discretion.

Dynamic Memory Allocation

- Structure of a data segment of .exe file in memory



Dynamic Memory Allocation

While using new/delete you should know ...

- No need of sizeof operator.
- new returns pointer of specified data type .
- if [] is used in new, use [] in delete also.
- Otherwise there will be problem of memory leakage

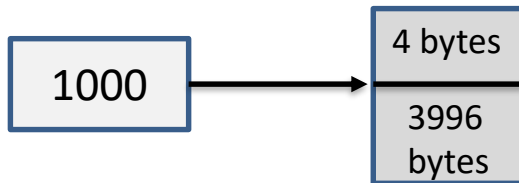
Dynamic Memory Allocation

Memory Leakage

```
int *p=new int[1000];
```

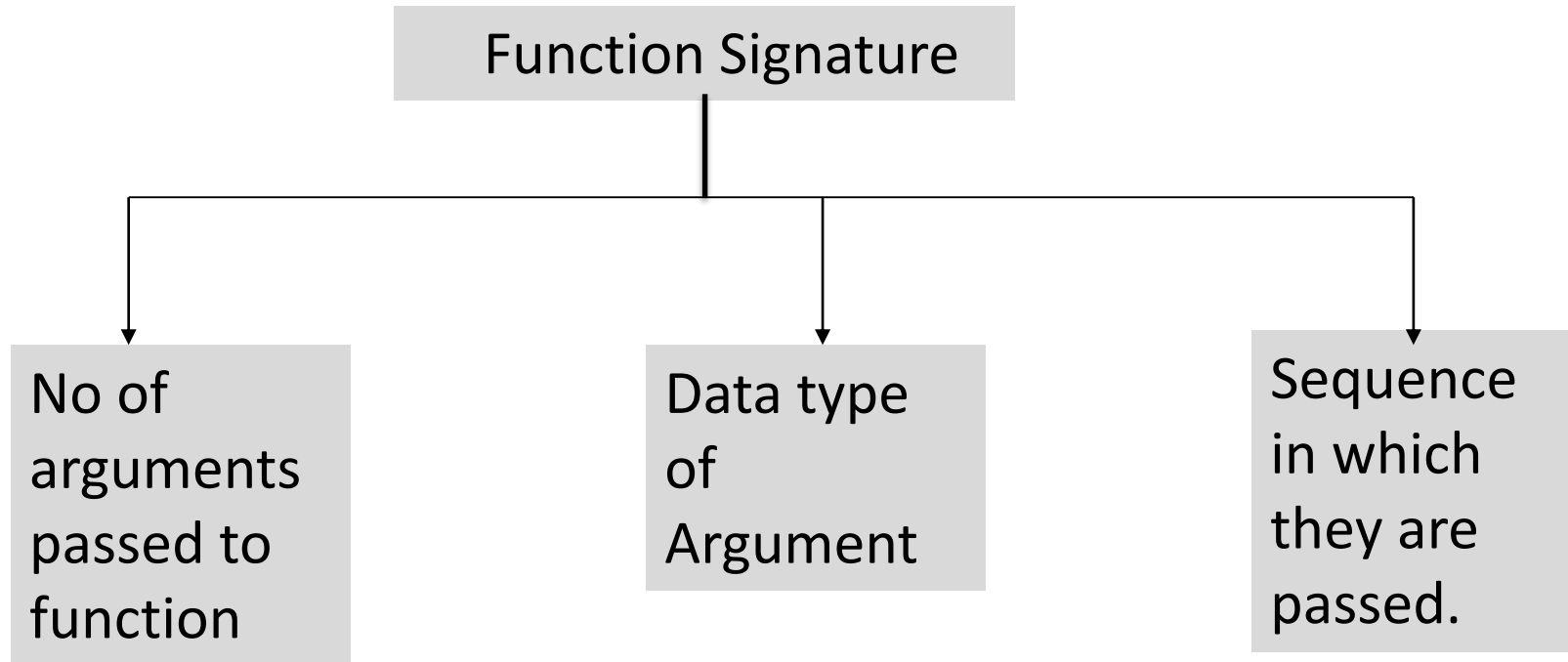
```
//..... Some code is  
there
```

```
delete [] ptr;
```



Memory leakage
occurs if delete ptr
Only first 4 bytes are
freed which results in
memory leakage of
3996 bytes

Function Overloading



Function Overloading

- Functions in C and C++ programs are known internally by their **decorated names**.
- A decorated name (also known as **mangled name**) is a string created by the compiler during compilation of the function definition or prototype.
- While in a C program, a function's decorated name includes the function's name only, **in C++ a function's decorated name also contains its parameters**.
- This way each overloaded version gets a unique internal name, enabling the compiler (and the linker) to handle few overloading versions of same function name.

int sum(int a, int b)

sum@1.....

float sum (float a, float b)

sum@2.....

float sum (int a, float b)

sum@3.....

float sum (float a, int b)

sum@4.....

Function Overloading

```
int      max(int, int);  
long     max(long, long) ;    // no collision  
double   max(double, double) ; //no collision
```

- The user can now simply specify the required operation and attach the required arguments:

```
int i = max (j, k);  
double d = max (d1, d2);
```

- Function overloading helps to generate clear and readable programs.

Function Match

- **Function match** (also known as function overload resolution) is the process in which a function call is associated with one function in a set of overloaded functions:

Given the following function definitions:

```
max (int, int) {//...}  
max (long, long) {//...}  
max (double, double) {//...}
```

- And the following function call in user's code:

```
max(5.5, 6);
```

- The function match process should “decide” which of the three **max()** function versions will be activated as a result of the above function call.
- For a call of a given function name, the compiler selects the best-matching function, based on the arguments specified in the function call.

What Does Match Compare?

- Obviously, the first thing compared is the **name** of the function that is called.
- If no function with this name has been defined, the compiler will return an error.
- If more than one function of the same name should have been defined, matching is done through **parameters' type**.

Returned-type function name (parameter type list) Match

- The name and the parameters list are the function's header components that play a role in the function match process.
- They are considered to be a function's signature.

Match Stages - cont'd

- At each stage the compiler checks for a match:
 - If there are **no** matches, it continues on to the next stage.
 - If there is **exactly one match**, the process is successful. Therefore, the compiler stops and enters a function call code.
 - If there is **more than one match**, the compiler stops and sets a compiler error: ambiguity.

- The function match process can result in either:
 - Success - One function matched the function call.
 - Failed - None of the functions matched the function call.
 - Failed - Ambiguous call: multiple functions match the function call equally well.

Default Arguments to a Function

In C++ function can be declared with default values.

- C++ allows to call a function without specifying all its arguments if a function is declared with default values.
- Only trailing arguments can have default values.
- The default values must be specified in the declaration & not in the definition.

Default Arguments to a Function

Default arguments

```
void f( int x, int y, int z)
{
    cout << x;
    cout << y;
    cout << z;
}
```

```
void main (void)
{
    void f(int, int = 5, int = 0) ;
    f ( ); // error
    f ( 10); //displays10 5 0
    f(10,20,30)//10 20 30
    f( 10, ,30)//error
}
```

End

Classes and Objects

Objective

- State what are classes , objects & constructors.
- Distinguish between structure in C & C++.
- Explain concept of class and object.
- Explain concept of constructor.
- Create object on heap.
- Create constant object & state rules for const objects
- Create static data member & state rules for using static data member.

Structure in C and C++

➤ Structure in C:

Supports only data abstraction.

➤ Structure in C++

Support data abstraction and procedural abstraction.

Therefore functions can also be added as members in structure

Structure in C and C++

Structure in c

```
struct student
{
    int rollNo;
    char name[20];
};
```

Structure in c++

```
Struct Student
{
    int rollNo;
    char name[20];
    void display()
    {
        cout<<name;
    }
};
```

Class

- struct keyword can be replaced by class keyword.
- Generally 'struct' is used in 'C' context while 'class' is used in C++ context.
- Class and object is C++ terminology
- Template for the creation of similar objects.
- A class in c++ is an encapsulation of data members and members.
- And member functions that manipulate the data.

Class

- A class is a template for the creation of similar objects.
- A class declaration defines a new type that links code and data. This new type is then used to declare objects of that class.
- Thus, a class is a logical abstraction, but an object has physical existence. In other words, an object is an *instance* of a class.
- Map real world entities into classes through data members & member functions.
- By writing class & creating objects of that class one can map two major pillars of object model i.e. abstraction & encapsulation into software domain.
- All members of a class data / methods are private by default.

Class Syntax

```
Class ClassName
```

```
{
```

```
    private:
```

```
        variable declaration;
```

```
        function declaration;
```

```
    public:
```

```
        variable declaration;
```

```
        function declaration;
```

```
};
```

If semicolon is missing compiler throws an error.

Class cDate

```
class cDate
```

```
{
```

```
private:
```

```
    int Day;
```

```
    int Month;
```

```
    int Year;
```

```
public:
```

```
    Void display();
```

```
    .....
```

```
};
```

Data members

Member Functions

```
int main()
```

```
{
```

```
    cDate d1;
```

```
    d1.display();
```

```
    return 0;
```

```
}
```


Class Components

- A class declaration consists of following components.
- Access Specifiers :restrict access to class members
 - Private
 - Protected
 - public
- Data members
- Members Functions
- Constrcutor
- Destructor
- Ordinary member functions

Types of member functions

Following are the types of member function of a class.

- Mutator- Changes contents of instance members;
- Accessor-Accesses instance members.
- Facilitator- Helps to view the values of attributes of object.
- Iterator –General method of Successively accessing each element of sequential or associative containers.
- Helper-A Private function, Accessed from public member function of same class to help in their implementation.

Instantiating a class

- Object is an instance of a class.

```
int main()
{
    cDate d1;
    cDate d2(12,6,1987);
}
```

Values need to be
intialized for the data
members of the
object

Constructor

- Constructor is a special function with same name as it's class name.
- Constructors can not have return type, not even void.
- Constructors are implicitly called when objects are created.
- A constructor without input parameter is called default constructor.

Constructor

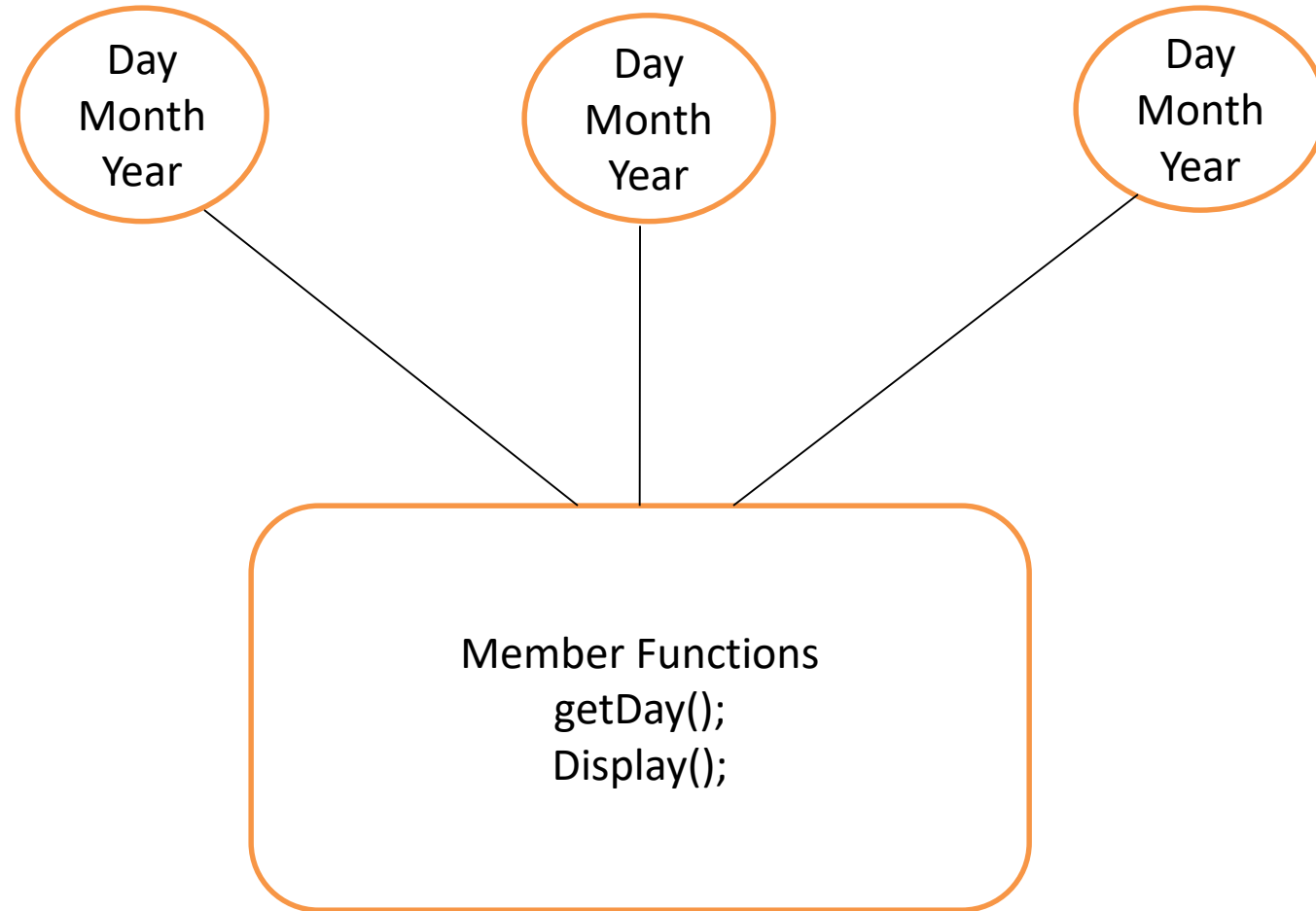
- Constructor can be overloaded.
- If your class does not have constructor, compiler provides a default constructor.
- An Object can not be created without an Constructor the constructor legalizes an Object.

Constructor in cDate class

```
Class cDate
{
    .....
    //no Argument constructor
    cDate()
    {
        Day=1;
        Month=1;
        Year=2000;
    }
    //Parameterized Constructor
    cDate(int d,int m,int y)
    {
        Day=d;
        Month=m;
        Year=y;
    }
};
```

```
int main()
{
    cDate d1;
    cDate d2(13,10,2016);
    return 0;
}
```

Memory Allocation to Objects



'this' pointer

- Address of an Object is passed implicitly to a member function, called on that Object.
- Every member function of class has hidden parameter : the **this** pointer.
- Pointer **this** holds the address of the Object invoking the member function.
- **this** is a keyword in C++

'this' pointer

- Pointer **this** is available only in member functions of the class.
- *Using the pointer this a member function knows on what Object it has to work.*
- *Using the pointer this a member function knows on what Object it has to work.*

Using 'this' Keyword

```
int cDate:: getDay()  
{  
    return this->Day;  
} //Or return Day is same;
```

```
Void cDate::setDay()  
{  
    this->Day=d;  
}
```

```
int main()  
{  
    cDate(3,5,2000);  
    int d=d1.getDay();  
    cout<<d;  
    d1.setDay(6);  
    return 0;  
}
```

class cDate

```
class cDate {  
    int day,month,year;  
public:  
    void setDate(int,int,int);  
};  
void cDate::setDate(int day,int month,int year)  
{  
    this->day=day;  
    this->month=month;  
    this->year=year;  
}
```

Object instance of a class

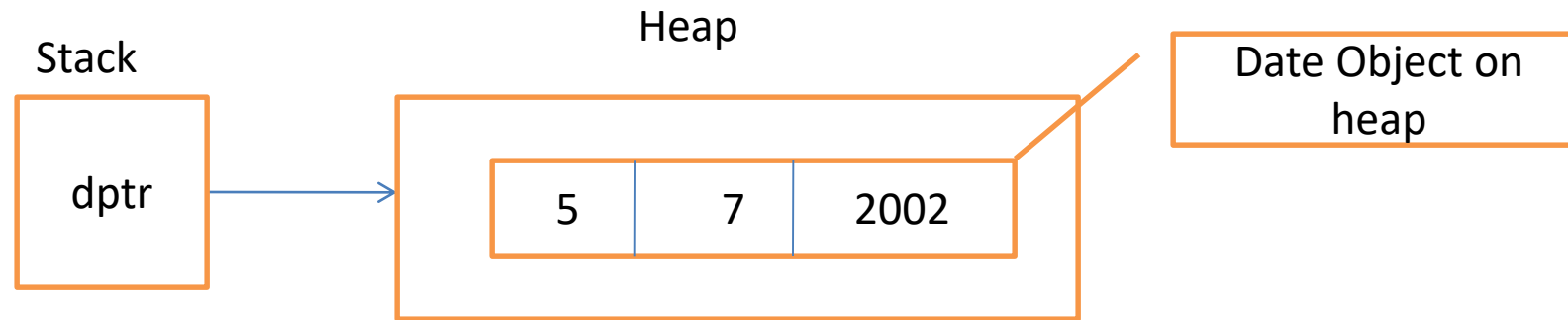
```
void main (void)      //client code
{
    cDate d1;          // d1 is an object of class cDate
}
```

Steps in creation of object.

- Memory is allocated.
- Constructor is called.
- Memory gets initialized

Creating Object on Heap

```
int main()
{
    cDate *dptr=new cDate(5,7,2002);
    ....
    .....
    retrun 0;
}
```



const in Classes

- Constant data members must be initialized through constructors using constructor initialization list.
- Use of constant inside a class means, “It will be constant for the lifetime of the Object”.
- Each object will have a different copy of the constant data member probably having different value.

Initializing const data members

```
class Data
{
    int val;
    Const int sconst;
Public:
    Data():sconst(0) //constructor initializer list
    {
        val=0;
    }
    Data( int val, int c ) : someConst(c)
    {
        this ->val=val;
    }
};
```

Const Member Functions

```
class cDate
{
    ....
public:
    ....
    int getDay(void) const;
    ...
}

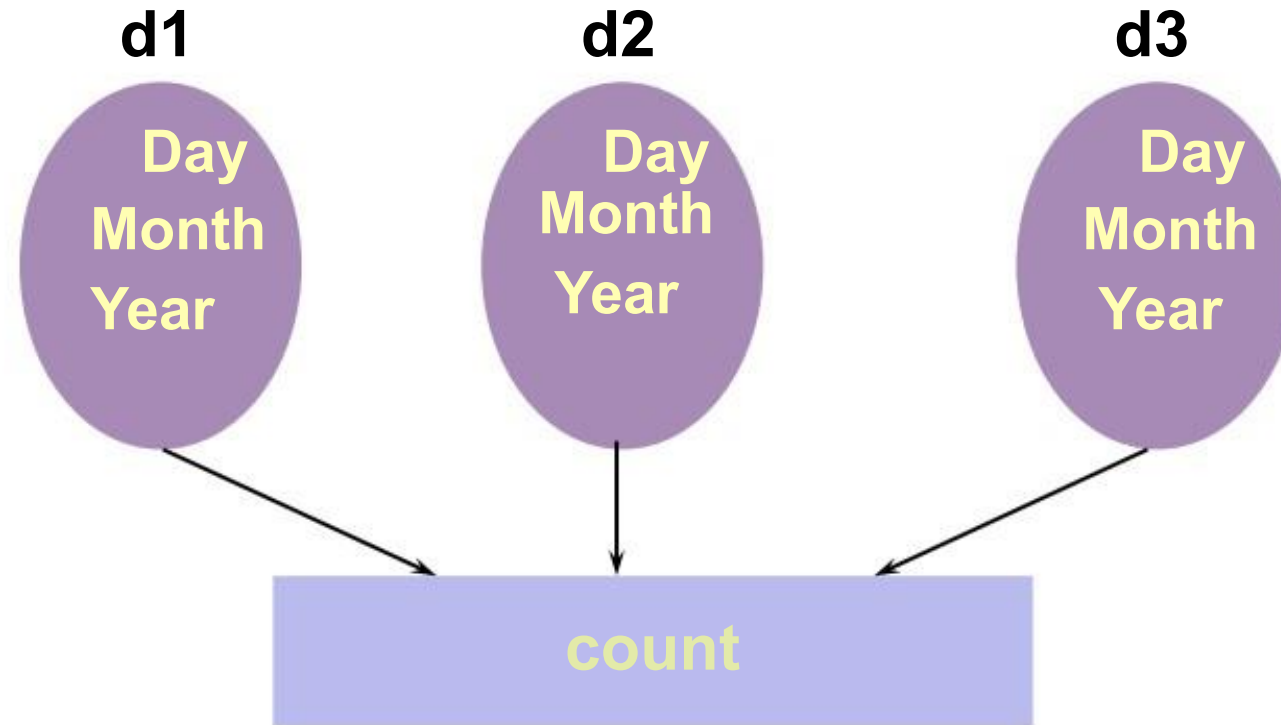
int cDate::getDay(void) const
{
    return this->Day;
}
```

```
int main()
{
    const cDate d1(5,7,2002);
    int day=d1.getDay();
    cout<<"Day is "<<day;
    return 0;
}
```


Static Data Members

- Useful when all objects of the same class must share a common item of information.
- Data to be shared by all objects is stored in static data members.
- There is single piece of storage for static data members.
- It's a class variable. Static data members could be made private, public or protected.
- Static data members are class members, they belong to class and not to any object.
- The static data member should be created and initialised before the `main()` program begins.

Objects with Static Member in Memory



Static Data Members - contd..

- As static data has single piece of memory, regardless of how many objects you have, Compiler will not allocate memory to static.
- If static data members are declared and not defined linker will report an error.
- Static data members must be defined outside the class. Only one definition for static data members is allowed.

Static Member Functions

- Static member functions can access static data members only.
- Static member function is invoked using class name.

class name :: function name()

- Pointer **this** is never passed to a static member function.
- Can be invoked before creation of any object.

Passing Object to Function

- Objects can be passed to functions in a similar manner like variables.
- If an object is “passed by value” mechanism Copy Constructor is called.
- A copy of an object is made i.e another object is created.
- To avoid creation of a copy of an object, objects should be passed by reference or by pointer.

Returning Object from function

- A function may return an object to the caller function.
- When an object is returned from a function.
- A temprary object is automatically created that holds the return value i.e temporary object is constructed by using returned object.
- Copy constructor is called for this temporary object.
- After the value has been returned to the caller function this temporary object is destroyed.

End

String Class and Copy Constructor

Objective

- Describe class String.
- Write constructors for class string.
- Need of Copy constructor
- Destructor for class string
- Object creation and destruction

cString class

```
class cString
{
    int s_len;
    char *s_buff;
public:
    cString();
    void display();
    .....
};
```

No-argument constructor

```
Class cString
```

```
{
```

```
    ...
```

```
    public:
```

```
    cString()
```

```
    {
```

```
        sLen=0;
```

```
        s_buff=new char;
```

```
        *s_buff='\0';
```

```
    }
```

```
};
```

```
int main()
```

```
{
```

```
    cString s1;
```

```
    return 0;
```

```
}
```

Parameterized constructor

```
Class cString
```

```
{
```

```
    ...
```

```
    public:
```

```
    cString(char ch,int len)
```

```
    {
```

```
        s_Len=len;
```

```
        s_buff=new
```

```
        char[s_len+1];
```

```
        memset(s_buff,ch,s_Len);
```

```
        *s_buff='\0';
```

```
    }
```

```
};
```

```
int main()
```

```
{
```

```
    cString s1('S',5);
```

```
    return 0;
```

```
}
```

Parameterized constructor

```
Class cString
```

```
{
```

```
    ...
```

```
    public:
```

```
    cString(const char *s)
```

```
    {
```

```
        s_Len=strlen(s);
```

```
        s_buff=new char[s_len+1];
```

```
        strcpy(s_buff,s);
```

```
    }
```

```
};
```

```
int main()
```

```
{
```

```
    cString s1("INFOWAY");
```

```
        return 0;
```

```
}
```

Destructor

- Destructor is also a special function with the same name as class name prefixed by ~ character. Eg ~Complex() ; or ~String();
- No need to specify return type or parameters to destructor function.
- Destructor cannot be overloaded. Therefore a class can have only one destructor.
- Destructor is implicitly called whenever an object ceases to exist.

Destructor

- Destructor function de-initializes the objects when they are destroyed.
- A destructor is automatically invoked when object goes out of scope or when the memory allocated to object is de-allocated using the delete operator.
- If a class contains pointer variable then it is mandatory on programmers part to write a destructor otherwise there is problem of memory leakage.

Destructor

Class cString

{

.....

public:

~cString()

{

if(s_buff)

{

delete [] s_buff;

}

}

};

Object Creation/Destruction

- Sequence of Object creation
 - 1) Memory is allocated
 - 2) Constructor is called
 - 3) Memory is initialized

- Sequence for Object Destruction
 - 1) Destructor is called
 - 2) If memory is allocated dynamically it is freed
 - 3) Memory allocated to objects is deallocated only when object goes out of scope.

Scope of an Object

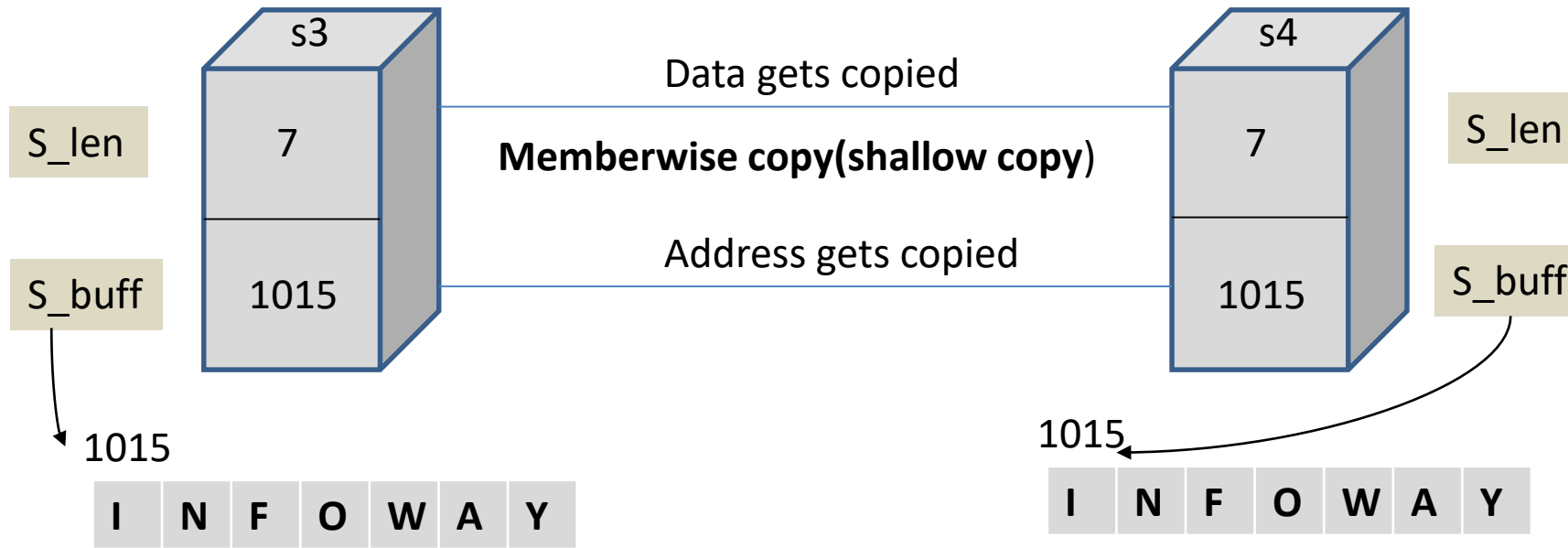
Example of Test class to show Construction and Destruction of an Object.

```
class Test
{
    ...
public:
    Test()
    {
        cout<<"Constructor is invoked";
    }
    ~Test()
    {
        cout<<"Destructor is invoked";
    }
};
```

```
Test t;
int main()
{
    Test t1;
    {
        Test t2;
        {}
    }
}
```

Need of Copy Constructor

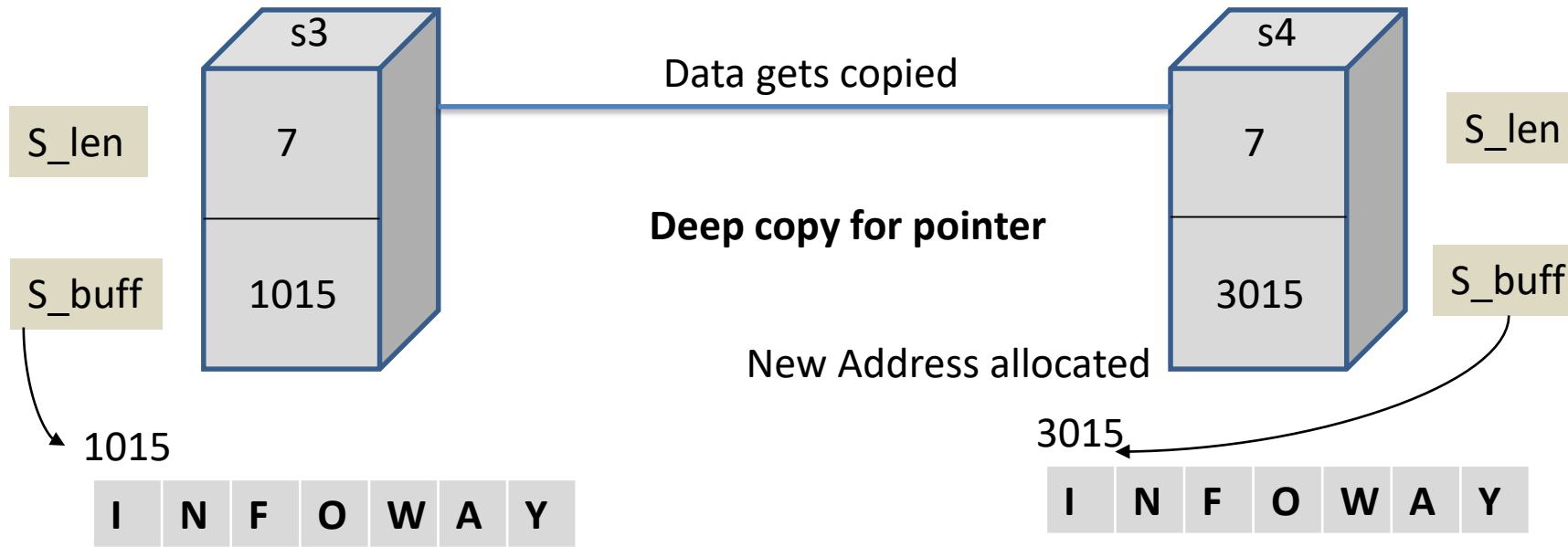
`cString s4(s3); //To create s4 as copy of s3`



If any one of the object goes out of scope .Pointer data member of other object becomes “**Dangling Pointer**”;

Need of Copy Constructor

`cString s4(s3);` //To create s4 as copy of s3



Copy constructor allocates separate memory on heap for string to be copied.

Copy Constructor for cString class

Class cString

{

.....

public:

cString(const cString &s)

{

s_len=s.s_len;

s_buff=new char[s_len+1];

strcpy(s_buff,s.s_buff);

}

};

int main()

{

cstring

s3("INFOWAY");

cString s4(s3);

}

Copy Constructor –Points to note

While creating copy of an Object

- Compiler Provides default copy constructor which does member wise copy.
- If a class contains one of it's data member as a pointer type ,it is mandatory on programmers part to write user defined copy constructor.
- User defined copy constructor should take care of dangling pointer situation.
- To avoid infinite recursion,pass the parameters by reference to the copy constructor.

End

Operator Overloading I

Objective

- Identify need of Operator Overloading.
- State Rules and restrictions on operator Overloading.
- Overload Arithmetic operators and pre/post increment operators for class.
- Overload Assignment operator .

Need of Operator Overloading

- Operator overloading is feature of c++ because of which additional meanings can be given to existing operators.
- Operator Overloading is an important technique that enhances power of extensibility.
- operator is keyword in C++ which is used to implement operator overloading.
- Operator Overloading feature makes UDT's more natural & closer to built in data types.

Restrictions on Operator Overloading constructor

You can not-

- Define new operators such as `**`.
- Change the precedence ,associativity or arity of an operator
- Meaning of operator when applied to built in types
- You can not overload
 - 1) The membership operator(`.`)
 - 2) The scope resolution operator(`::`).
 - 3) The ternary operator `?:`
 - 4) The sizeof operator

Format for Operator Function

```
Return_type class_name::operator#(argument List)
```

```
{
```

```
.....
```

```
}
```

Where

- # is a placeholder.substitute operator for the #
- Complex Complex::operator+(Complex &)
- Operator functions are non static member functions of class.

Operator class

Overload + operator to do addition of two Complex class objects that is

```
//in main
```

```
{
```

```
    Complex c1(3,4);
```

```
    Complex c2(1,2);
```

```
    Complex c3=c1+c2;
```

```
}
```

Internally the call is
resolved as

Comeplx

c3=c1.operator+(c2);

Operator class

Overload ++ operator to do pre/post increment of a Complex class objects

//in main

```
{  
    Complex c1(3,4);  
    Complex c2(1,2);  
    Complex c3=c1++;  
}
```

Overload Binary subtraction and unary '−' operator for Complex object.

Operator = for class cstring

Overload “=” assignment operator for cstring class to assign one string object to other.

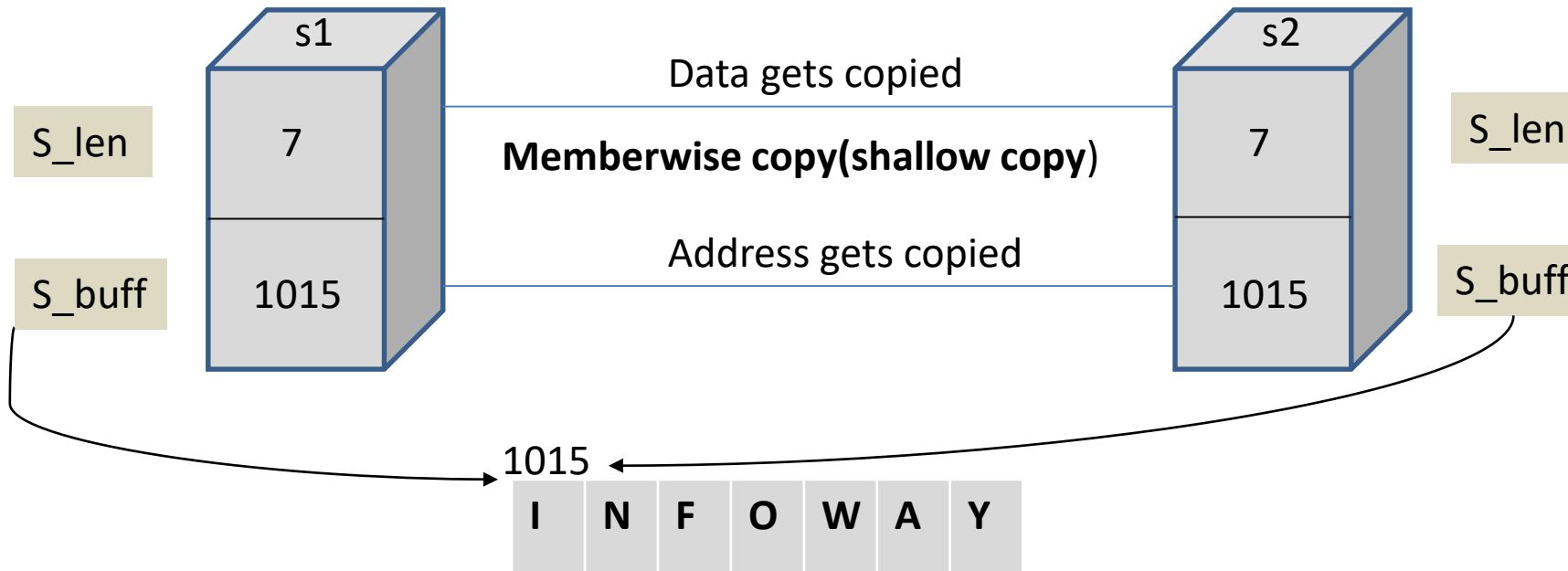
```
cString s1("Technologies");
```

```
cString s2("Infoway");
```

```
    s1=s2;
```

“=” operator overloading

`cString s1=s2 //To copy s2 as copy of s1`



This scenario leads to two problems.

- 1.Memory Leakage
- 2.Dangling Pointer

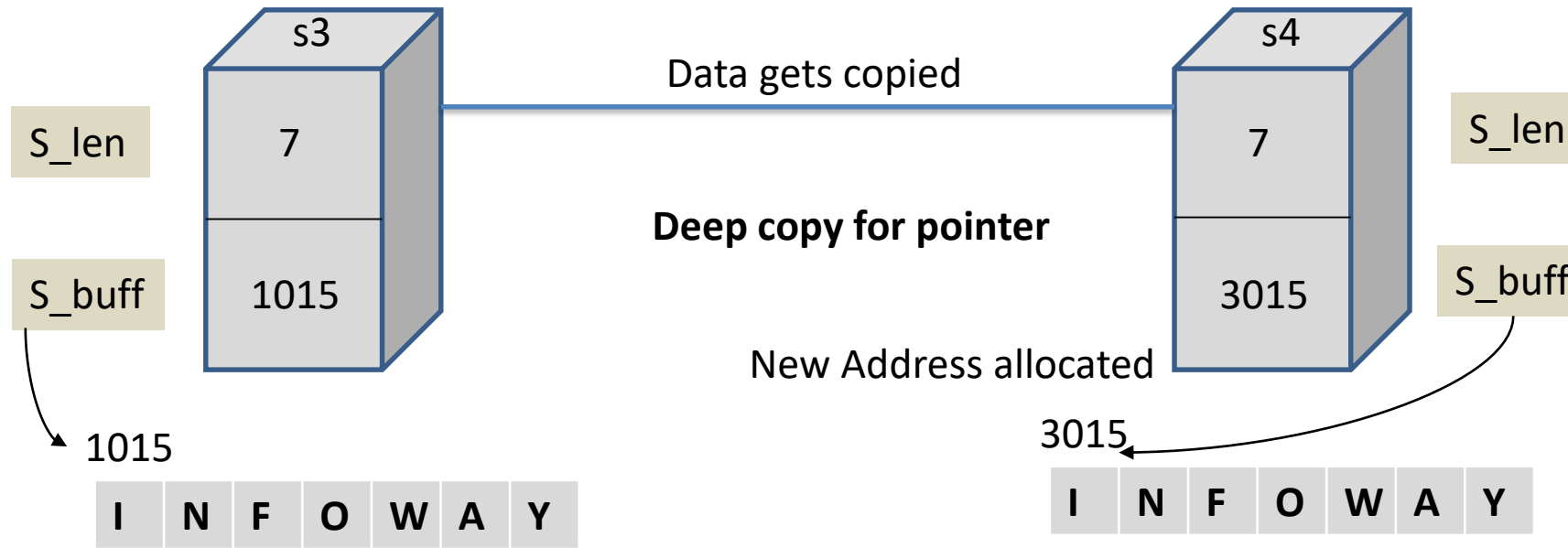
“=” operator overloading

Steps to overcome Memory Leakage and Dangling pointer.

- 1.Delete the old buffer.which prevents memory leakage.
- 2.Allocate enough new memory on heap to prevent dangling pointer problem.
- 3.Copy the string to the newly allocated memory.

"=" operator overloading

cString s1=s2 //To copy s2 as copy of s1



Operator Function “=” for cString class

```
Class cString
{
    cString & operator =(const cString &);
};
cString & cString::operator =(const cString &s)
{
    if(this==&s)
        return *this;
    else
    {
        s_len=s.s_len;
        delete [] s_buff;
        s_buff=new char[s_len+1];
        strcpy(s_buff,s.s_buff);
        return (*this);
    }
}
```

```
int main()
{
    cstring
    S1("Technologies");
    s2("INFOWAY");
}
```

Summary

- If class contains at least one data member as a pointer then to avoid memory related problems you must implement following
 - User defined copy constructor
 - Destructor
 - Assignment operator overloading.

End

Operator Overloading II

Objective

- Overload operators [],().
- Identify the Need of Friend Function
- Overload <<, >> operators for Complex class
- Restrictions on Friend Functions.

Operator Overloading

- Operator overloading is feature of c++ because of which additional meanings can be given to existing operators.
- Few More operators can be overloaded for Complex and CString class that we discussed earlier.
- Other operators are [],(),<<, >> etc.

[] Operator overload for cString class

```
//Part of main
```

```
{
```

```
    cString s1("Infoway");
```

```
    char ch=s1[0];
```

```
    s1[0]='T'
```

```
}
```

There are two cases here

1.ch=s1[0];

2.s1[0]='T'

Let's try to write code for first example.

[] Operator for cString Case I

```
class cString
{
    char operator[](int);
};

char cString::operator [](int index)
{
    if(index>=0 && index<s_Len)
        return (*(s_buff+index));
}
```

[] Operator cString class case II

```
class cString                                char & cString ::operator [](int index)
{                                           {
    char & operator[](int);                if(index>=0 && index<s_Len)
                                           return (*(s_buff+index));
};                                           }
                                           }
```

When any function returns by reference and appears on left hand side of assignment operator ,the location of that variable is considered instead of it's value.

Function call () operator Overloading

```
class cString                                     char *cString ::operator()(int index)
{                                                  {
    char *operator()(int);                        if(index>=0 && index<s_Len)
                                                    return (s_buff+index);
};

int main()                                       }
{
    cString s1("Infoway");
    Char *str=s1(2);
}
```

Need of Friend Function

```
//part of main
```

```
{
```

```
    complex c1(2,3),c2;
```

```
    c2=c1+5;    // c1.operator +(5);
```

```
    c2=5+c1;    // ?? Will it work.
```

```
}
```

For the Statement `c2=5+c1` ;operator + can not be overloaded as member function

Need of Friend Function

To do so ,it has to be overloaded as non member function.

- Friend functions are not members of a class.(i.e non member function)
- They are normal global functions and hence do not implicitly receive this pointer.
- Friend Function can access private members of a class.

Friend Function

- Friend Function is written as other normal function

`return_type functionName(argument_list) {}`

Only precede friend keyword before function declaration

- Friend functions can be declared in private or public section of class

Using Friend complex class

```
class complex
{
    friend comeplx operator+(int,const complex&);
};

complex operator+(int num,const complex &c)
{
    complex temp;
    temp.real=num+c.real;
    temp.img=num+c.img;
    return temp;
}

int main()
{
    complex c1(3,5),c2;
    c2=5+c1;
}
```


Overloading of << >> for comeplx class

- The << and >> are called stream operators and are binary operators.
- The operator >>() is an overloaded member function of class istream class.
- The operator <<() is an overloaded member function of class ostream class.
- These operators are overloaded for all built in types.

Example of << operator

- Implement complex class to overload << and >> operator

```
class complex
```

```
{
```

```
    ...
```

```
    friend ostream & operator <<(ostream &,const comeplx &);
```

```
    friend istream & operator >>(istream &,comeplx &);
```

```
};
```

Restriction of friend function

- =,[],(),-> these all operators can not be overloaded using friend function.
- All the above operators should be overloaded using non static member function of the class.

When to use Friend Function

- Whenever first operand happens to be built-in type, use friend function to overload.
- When operations on two dissimilar data types to be performed, friend function have to be used.
- When one of the argument to a friend function belongs to a class of which it is a friend.
- When it is absolutely necessary, since it defeats object oriented paradigm.

End

Containment and Inheritance

Objective

- Define containment and state its rules
- Construct program to implement containment
- Identify need of member initializer list.
- Define inheritance and state its rules
- Program to implement inheritance
- Define function overriding.

Containment

- Containment represents 'has-a' or 'is a part of' relationships.

- It depicts how one object is part of another object.

For example: engines, wheels, cd-players etc are parts of car.

- Container relationship enables reusability of code.

Engine is also used in an airplane.

Pan card is used for opening bank account as well as for IT returns.

Containment example

```
class cEmployee
{
    int empid;
    int sal;
    cString name;
    cDate bdate;
public:
    cEmployee(int e,int s,char *name,int d,int m,int y);

};
```

Containment example

```
cEmployee::cEmployee(int e,int s,char *nm,int d,int m,int y)
{
    empid=e;
    salary=s;
    name=cString(nm);
    bdate=cDate(d,m,y);
}
```

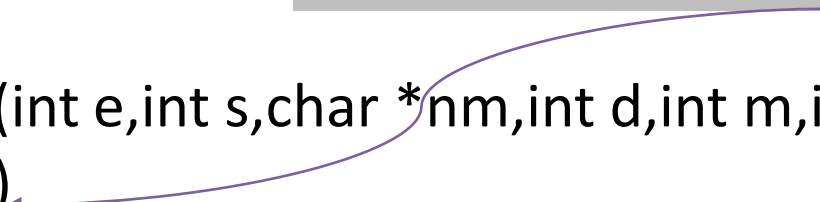
Execution sequence

- Default constructor for class cString
- Default constructor for class cDate
- Constructor for class cEmployee through which
 - Parameterized constructor for class cString
 - Parameterized constructor for class cDate.
- Total no of constructors invoked =5

Using member initializer list

Member Initializer List

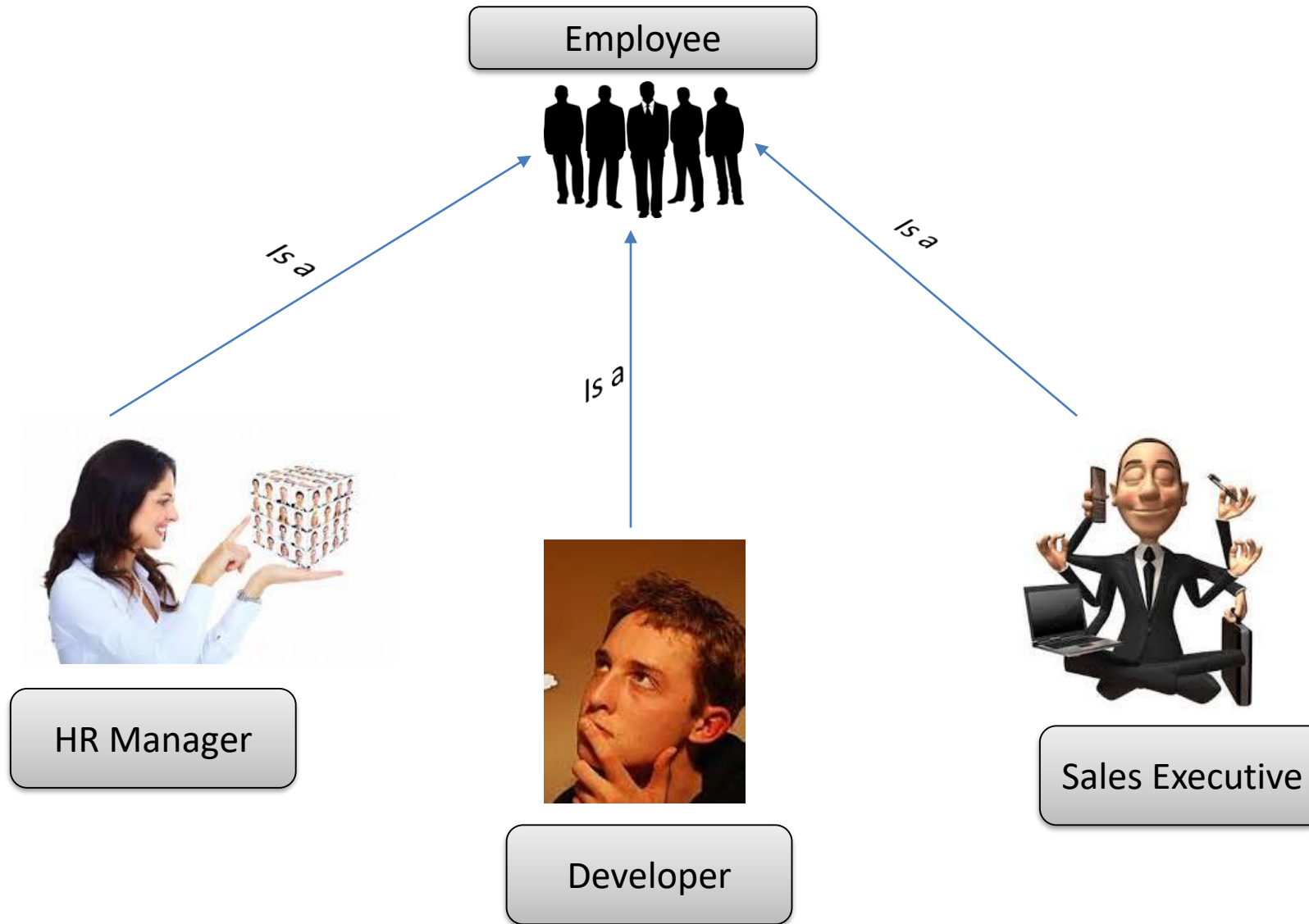
```
cEmployee::cEmployee(int e,int s,char *nm,int d,int m,int y):  
    name(nm),bdate(d,m,y)  
{  
    empid=e;  
    salary=s;  
}
```



Execution sequence

- While using Member initializer list ,constructors for contained objects are invoked first and then constructors for container objects.
- Parameterized constructors called directly.
- Constructor for class cEmployee through which
 - Parameterized constructor for class cString
 - Parameterized constructor for class cDate.
- Total no of constructors invoked =3

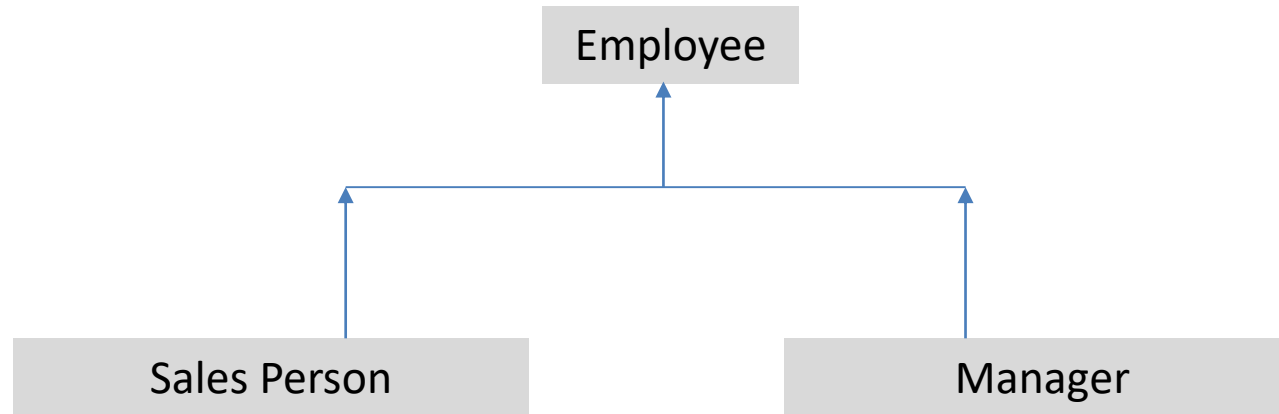
Relationship



Inheritance

- One of the key concepts of object oriented programming approach.
- Allows creation of hierarchical classification.
- Advantages are-
 - Reusability
 - Extensibility

Base and Derived class



- This is 'is a' kind of relationship.
- More than one class can inherit attributes from a single base class.
- Derived class can be base class to another class.

Inheritance syntax

```
class base_class  
{  
    //code for base class  
};
```

```
Class derived_class:access base_class name  
{  
    //code for derived class  
};
```

Access can be public or private or protected.

Class cSalesPerson

```
class cSalesPerson:public cEmployee
```

```
{
```

```
    float sales;
```

```
    float comm;
```

```
    public:
```

```
    void display();
```

```
    void compute_salary();
```

```
};
```

```
cSalesPerson::cSalesPerson(int e,int sal,const char *nm,int d,int  
m,int y,float s,float c):cEmployee(e,sal,nm,d,m,y)
```

```
{
```

```
    sale=s;
```

```
    comm=c;
```

```
}
```

Base initializer list



Derived class constructor and destructor

- Constructors are called in the sequence of Base->derived
- When Object of derived class is created the sequence of constructor calling is

cString->cDate->cEmployee->cSalesPerson

- Destructors are called in the sequence of Derived -> base

cSalesPerson->cEmployee->cDate->cString

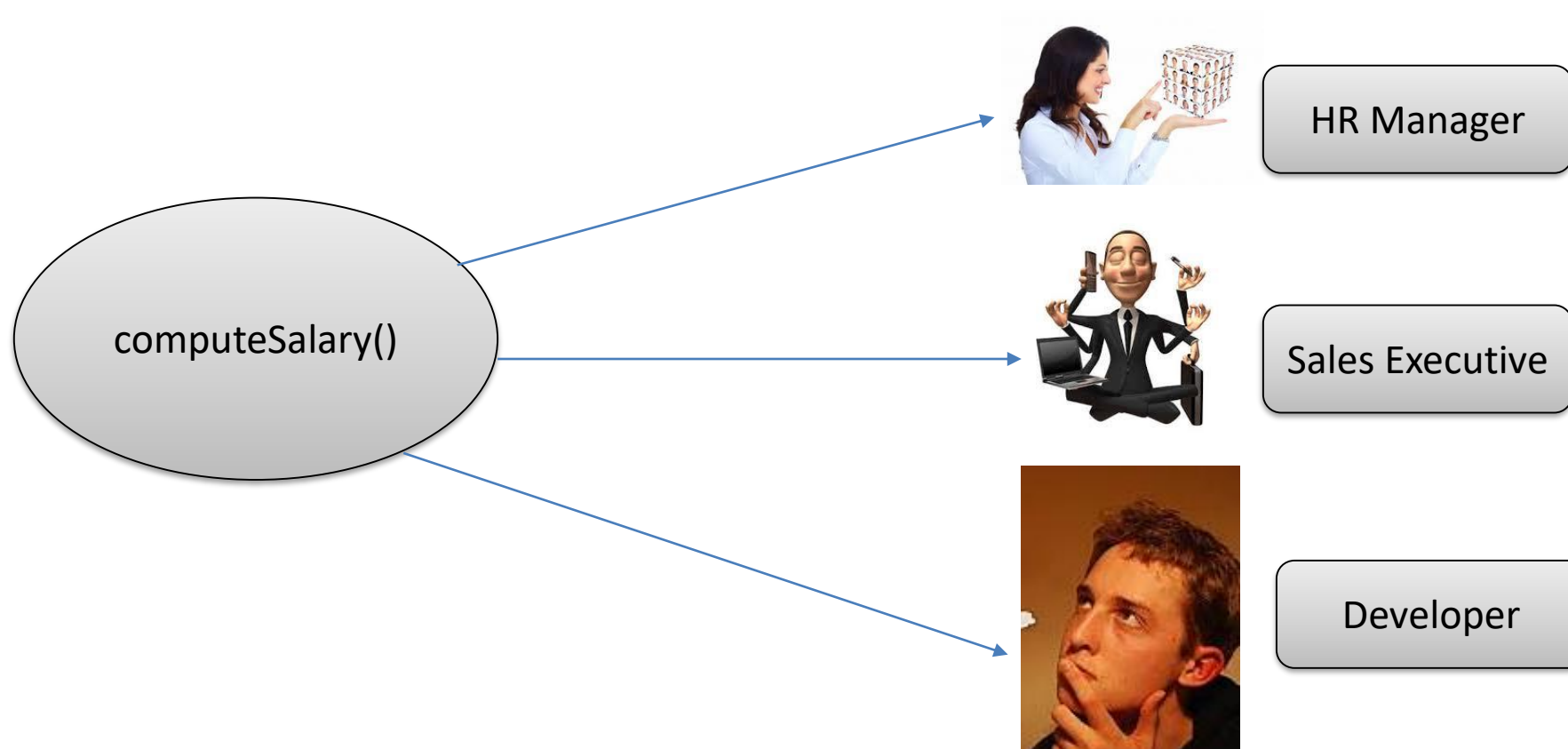
Polymorphism and Inheritance

Objective

- Distinguish between compile-time and run time binding.
- Use of generic pointers.
- Program for polymorphism using virtual function.
- Difference between overloading and overriding.
- Virtual destructor
- Define abstract class

Relationship

- Ability of different related objects to respond to the same message in different ways is called Polymorphism.



Compile time and Run time binding

- Binding is an association of function call to an object.
- Compile time binding
 - The binding of member function call with an object at compile time.
 - Also called as dynamic binding or late binding.
- Runtime binding
 - The binding of function call with an object at run time.
 - Also called as dynamic binding or late binding
 - Achieved using virtual functions and inheritance.

Generic pointer

```
int main()
{
    cEmployee e(...), *emp;
    cSalesPerson s1(...);
    emp=&e;
    emp=&s1;
}
```

Base class or generic pointer can point to objects of derived class.

- Assigning derived class objects to base class pointer or reference is always safe. But assigning derived class object to base class objects leads to “Object Slicing”.

Type casting

```
int main()
{
    cEmployee e(...),*emp;
    cSalesPerson s1(...);
    emp=&s1;
    ((cSalesPerson *)emp)->computeSalary()
}
```

- Based on which computeSalary() should be called ,cEmployee pointer is type casted to a particular type.
- Compiler always checks static data types.

Virtual Functions

- To implement late binding ,the function is declare with the keyword virtual in the base class.
- Virtual function is a member function of a class.
- Virtual functions can be redefined in the derived class as per the design of the class.
- Also consider virtual by compiler.

Generic pointer

- Declare computeSalary() as virtual in cEmployee class.

```
int main()
{
    cEmployee e(...), *emp;
    cSalesPerson s1(...);
    emp=&s1;
    emp->computeSalary()
}
```

- Dynamic data type of generic pointer will govern method invocation.

Virtual Functions..points to note

- It should be non static member function of base class.
- Can not be used as friend function.
- Constructor can not be virtual but destructors can be.
- If Function is declare virtual in base class then it is treated virtual in derived class too,even if virtual keyword is not used explicitly .

Overloading vs Overriding

	Overloading	Overriding
Scope	Within same class	In the inherited classes.
Purpose	Method names need not be remembered	Message is same but implementation is specific to particular class.
Signature	Different for each function	Has to be same in all derived classes.
Return Type	can be same or different for each function but it is not consider	Return type also needs to be same .

Pure Virtual Functions

- Virtual function without any executable code .
- Declare as follows,let say in cEmployee

```
virtual float computeSalary()=0;
```

- A class containing at least one pure virtual function is known as abstract class.

Virtual destructor

Class A

```
{  
  
    public  
    ~A()  
    {  
        cout<<"A destructor";  
    }  
};
```

class B:public A

```
{  
  
    public:  
    ~B()  
    {  
        cout<<"B destructor";  
    }  
};  
  
int main()  
{  
    A *a_ptr=new B();  
    .....  
    Delete a_ptr;  
}
```

Types of classes

- Concrete class
- Abstract class
- Pure abstract
- Polymorphic clas

Abstract class

- An object of an abstract class can not be created.
- Pointer or reference of abstract class can be created.
- Therefore ,abstract class supports Runtime polymorphism.
- Pure virtual functions must be overridden in derived classes;
Otherwise derived classes also treated as abstract.

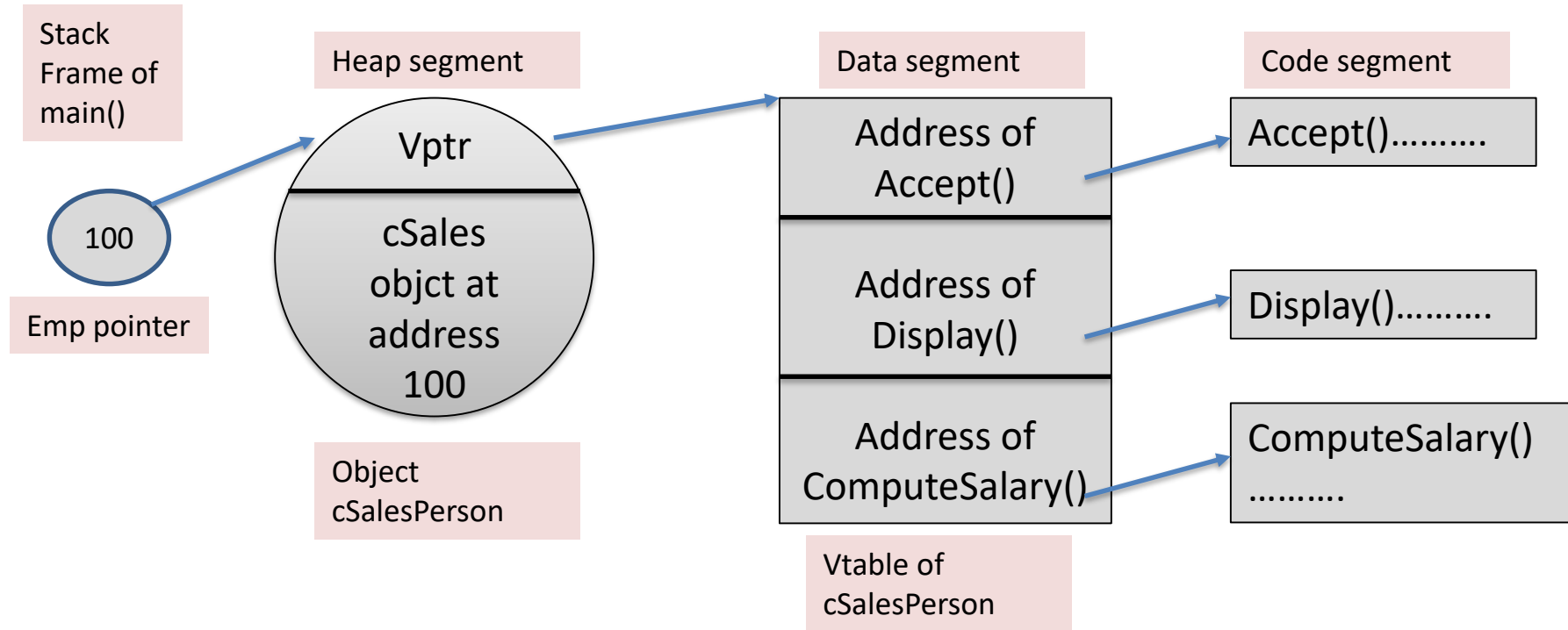
How virtual function work?

- For every polymorphic class compiler implicitly adds data member referred as vptr to every object.
- Virtual pointer is a hidden pointer
- Vptr is a pointer to static table of function pointers called Virtual table(vtable).

How virtual function work?

- Vptr is initialized to the starting address of the vtable in the constructor.
- Virtual table is an array of function pointers that contains pointers to all the virtual functions in the class.
- Virtual table is static member of a class since all objects of that class need reference to virtual table.

How virtual function work?



End

Multiple Inheritance

Objective

- Different types of inheritance
- Multiple inheritance and its problems.
- Identify problems with diamond inheritance
- Need of Virtual base class
- Implement diamond inheritance

Modes of inheritance

- Three Modes of inheritance
 - Access specifier:private
 - Access specifier:public
 - Access specifier:protected

- We discuss about private and public mode of inheritance earlier. Now let's discuss what is protected mode of inheritance.

Protected access Specifier

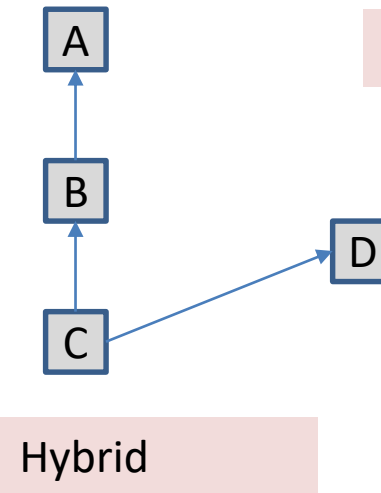
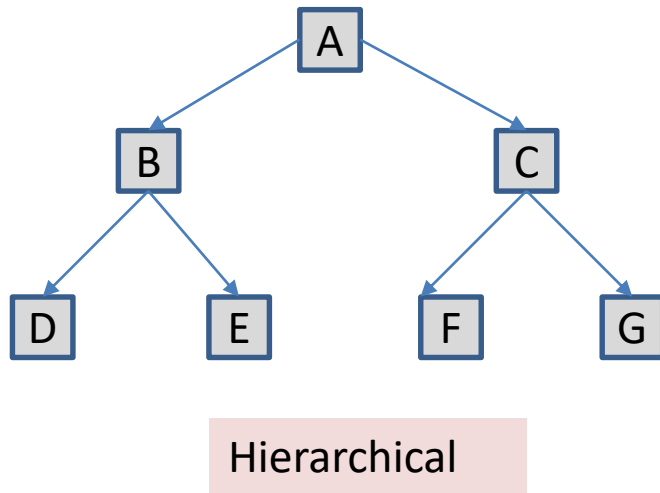
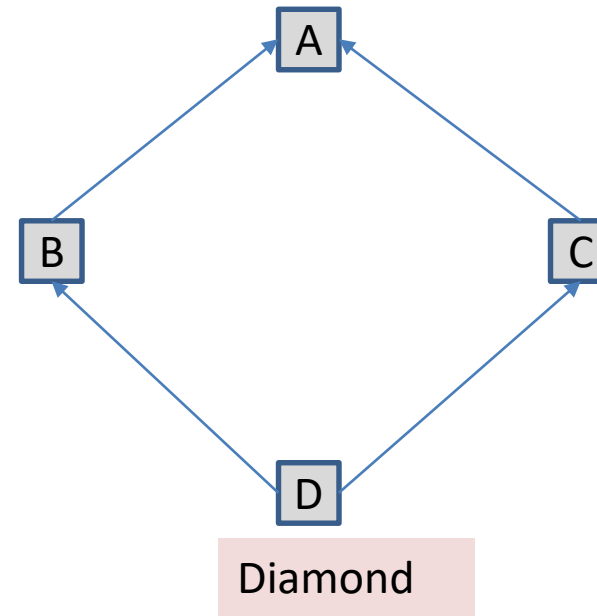
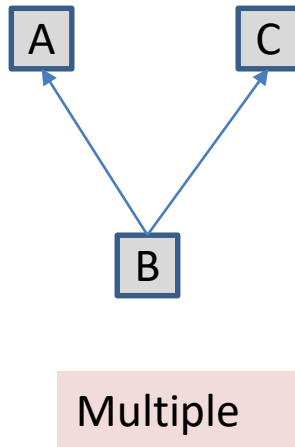
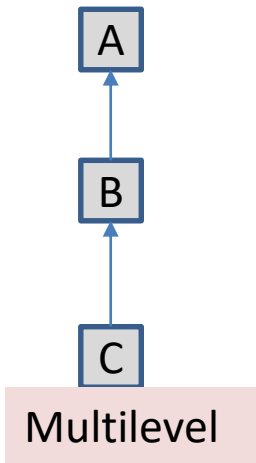
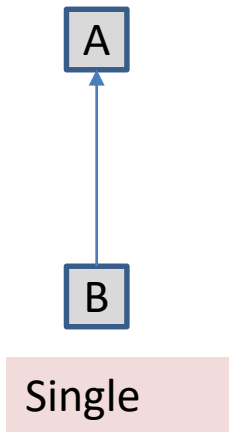
- The Protected keyword is added in c++ to provide flexibility to the inheritance mechanism.
- When class member is declared as protected, that member is not accessible by other **non-member** function of the class.
- Access to protected member is same as access to private member i.e can be accessed only by the members of the class.

Modes of Inheritance

Accessibility in derived class for all modes

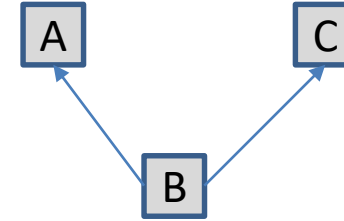
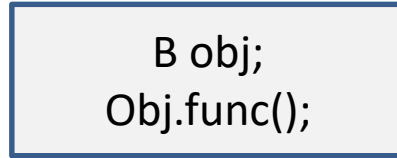
Type of Inheritance	Private	Public	Protected.
Private Mode	Private	Private	Private
Public Mode	Private	Public	Protected
Protected Mode	Private	Protected	protected

Types of inheritance



Problems with multiple inheritance

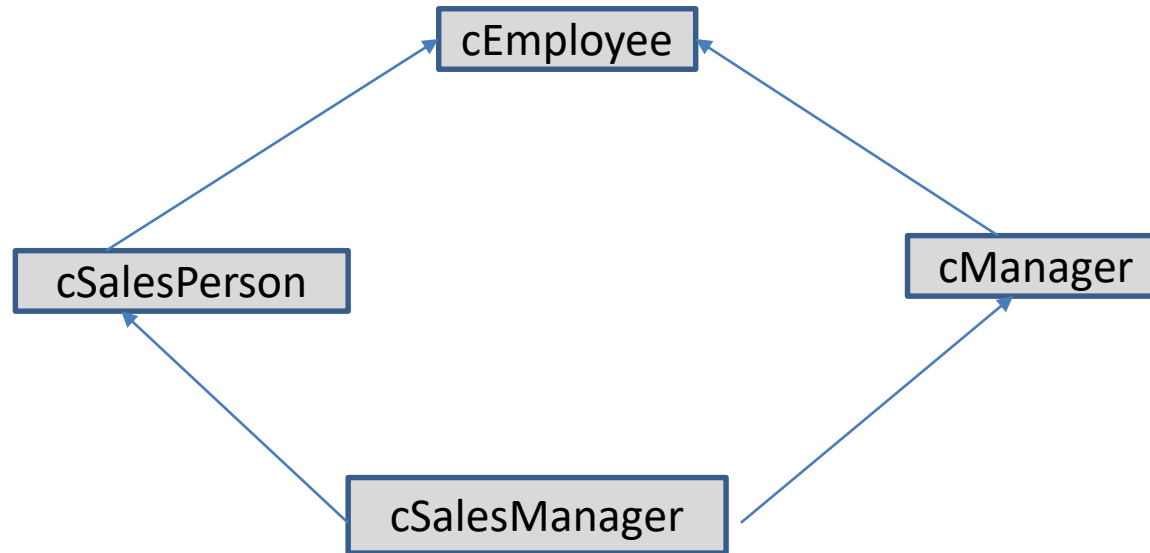
```
class B:public A,public C{ .....
```



- If multiple base class contains function with same name. Compiler throws ambiguity error ,as obj can see two copies of same functions from two classes.
- To resolve this ambiguity by two ways
 - `Obj.A::func()`
 - Override func in B class

Diamond Inheritance

- When a class inherits two classes, each of which inherits from a single base class, it leads to diamond inheritance pattern

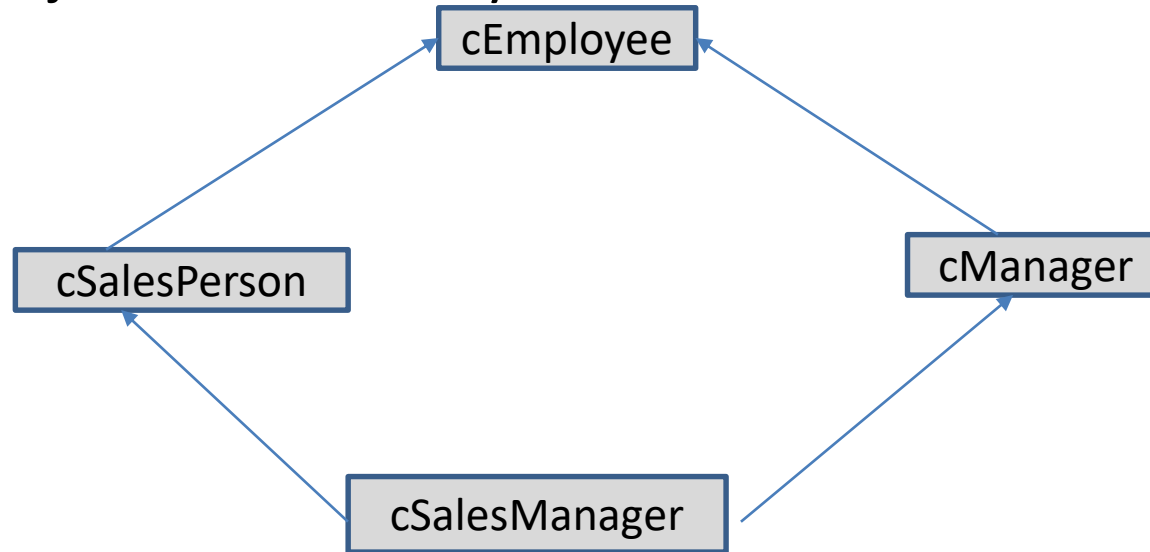


Ambiguities in Diamond Inheritance

- Ambiguity in calling the function.
 - It is similar problem discussed in multiple inheritance.
 - It can be resolved by same way
 - `csalObj.cEmployee::func()`
 - Override func in cSalesManager class.
- Data Duplication
 - It happens when the derived class gets multiple copies of the same base class.

Diamond Inheritance

- So if we consider below example and Create an object of cSalesManager class –then
- Object will have two copies of id and name from ,which increases size of object unnecessarily.



Virtual Base class

- Duplicate data member ambiguity can be resolved by declaring Virtual base class.
- Derive cSalesPerson and cManager class using virtual keyword from cEmployees
- Then derive cManager class from both of them.

Virtual Base class

- By declaring base class as virtual ,duplicate copies of base class data member is not created..
- There is only one copy of common base class in memory and it's pointer reference is there in derived class object.
- The meaning of virtual keyword is overloaded.

Virtual Base class

```
Class cSalesPerson:virtual public cEmployee
{
    .....
};
```

```
Class cManager:virtual public cManager
{
    .....
};
```

When inheritance is implemented using virtual base class it is termed as Virtual Inheritance.

End

Exception Handling

Objective

- Identify need of Exception
- Define exception
- Implement the mechanism of exception handling using try ,catch and throw.
- Construct code to handle exceptions and user defined exceptions.

Errors

- Writing program code and error generation go hand in hand.
- They can occur as a result of incorrect syntax, incorrect logic or can occur at run time.
- There are different types of errors.
 - Logical errors
 - Run-time errors

Exception

- The errors that occurs during program execution of a program are called as Runtime errors or **Exceptions**.
- For Emample
 - Division by 0
 - Access to an array outside its bounds
 - Stack overflow
 - File not found
 - Invalid Type casting.

Exception

- C++ exception handling is built upon three keywords: **try**, **catch**, and **throw**

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

```
catch (type2 arg) {  
    // catch block  
}  
catch (type3 arg) {  
    // catch block  
}...  
catch (typeN arg) {  
    // catch block  
}
```


Exception

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Start\n";
    try { // start a try block
        cout << "Inside try block\n";
        throw 100; // throw an error
        cout << "This will not execute";
    }
    catch (int i) { // catch an error
        cout << "Caught an exception -- value is: ";
        cout << i << "\n";
    }
    cout << "End";

}
```

Exception

```
// Catching class type exceptions.
#include <iostream>
#include <cstring>
using namespace std;
class MyException {
public:
    char str_what[80];
    int what;
    MyException() { *str_what = 0; what = 0; }
    MyException(char *s, int e)
    {
        strcpy(str_what, s);
        what = e;
    }
};
```

```
int main()
{
    int i;
    try
    {
        cout << "Enter a positive
number: ";
        cin >> i;
        if(i<0)
            throw MyException("Not
Positive", i);
    }
    catch (MyException e)
    { // catch an error
        cout << e.str_what << ": ";
        cout << e.what << "\n";
    }
    return 0;
}
```

Using Multiple catch Statements

```
#include <iostream>
using namespace std;
// Different types of exceptions can be caught.
void Xhandler(int test)
{
    Try
    {
        if(test) throw test;
        else throw "Value is zero";
    }
    catch(int i)
    {
        cout << "Caught Exception #: " << i << '\n';
    }
    catch(const char *str) {
        cout << "Caught a string: ";
        cout << str << '\n';
    }
}

int main()
{
    cout << "Start\n";
    Xhandler(1);
    Xhandler(2);
    Xhandler(0);
    Xhandler(3);
    cout << "End";
    return 0;
}
```

While using exception handling.....

Note that.

- When an exception is raised, program flow continues after catch block.
- Control never comes back to the point from where exception is thrown.
- Memory leakage in context with exception when object is created on heap.

If Exceptions Are Ignored...

Note that.

- If exceptions are ignored or not handled properly, the program is terminated.
- This will happen in cases where:
 - An exception is thrown out of a **try** block.
 - No appropriate catch block has been defined to handle an exception.
- In such a case, the standard termination function - **terminate()** is executed
- Memory leakage in context with exception when object is created on heap.
- The function executes the **abort()** function.

If Exceptions Are Ignored...

Note that.

- We can define our own `terminate()` function, using the `set_terminate()` function:

```
Void myTerminate()  
{  
    //    do    whatever    required  
}
```

```
Void main()  
{  
    set_terminate(myTerminate) ;  
    //regular execution  
}
```

- `set_terminate` is defined in `<exception>`

Catch block Definition

- Catch every exceptio.
- A **catch** can be defined to receive 3 points as a parameter:
catch (...).
- The meaning is **catch-all**:
- Any exception object can be caught by this **catch** block.
- The type of error thrown cannot be clarified at this entrance.
- **catch** blocks are scanned in order to find the **first** match (and not the **best** match).
- Therefore, the **catch-all** block must be the last in a list of
- **catch** blocks.
 - Adding a **catch-all** block as the last **catch** block in **main()** can be used to avoid unhandled exceptions to escape out of our program.
 - Here we can define the code to be activated when an
 - unhandled exception occurs.

End

File handling and Console i/o

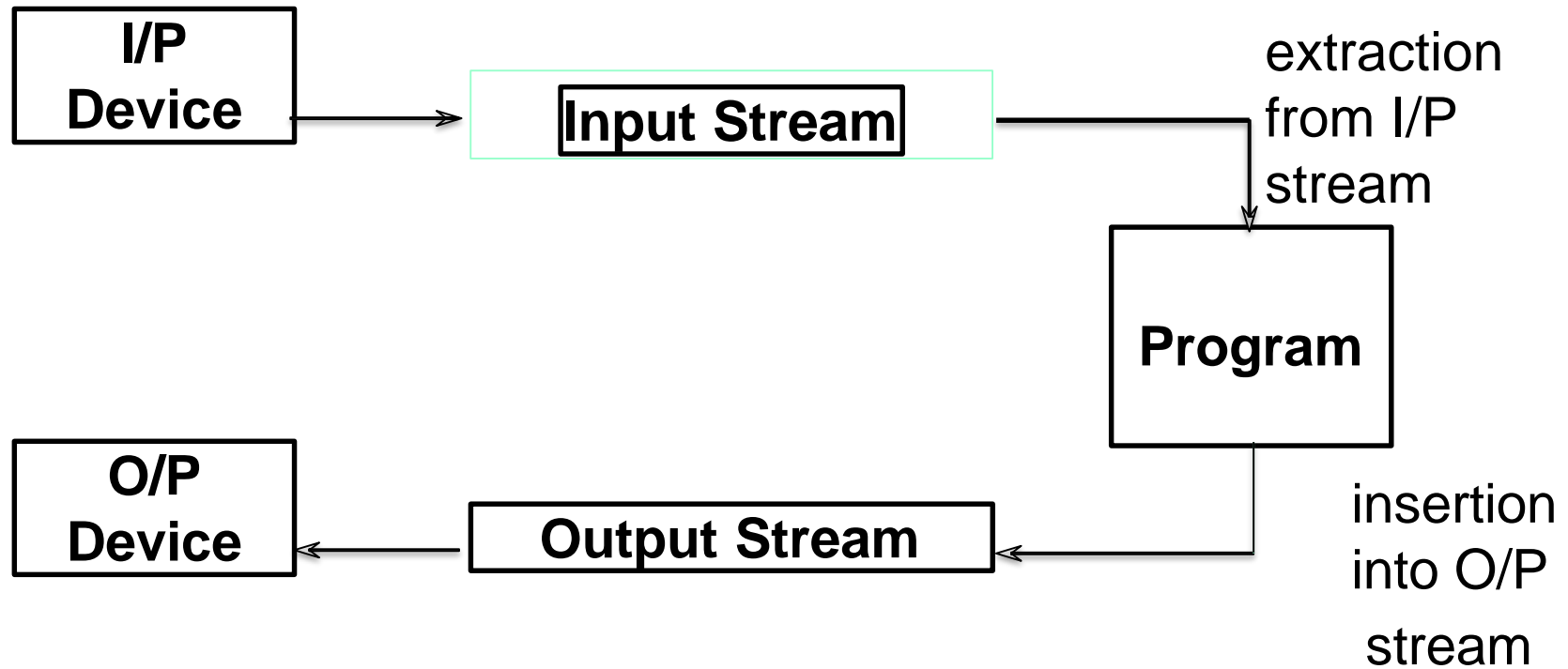
Objective

- List formatted and unformatted console I/O functions.
- Define Stream
- List classes required for performing file I/O in c++
- Random access operations on file
- `>>` and `<<` are overloaded for all built-in data types in

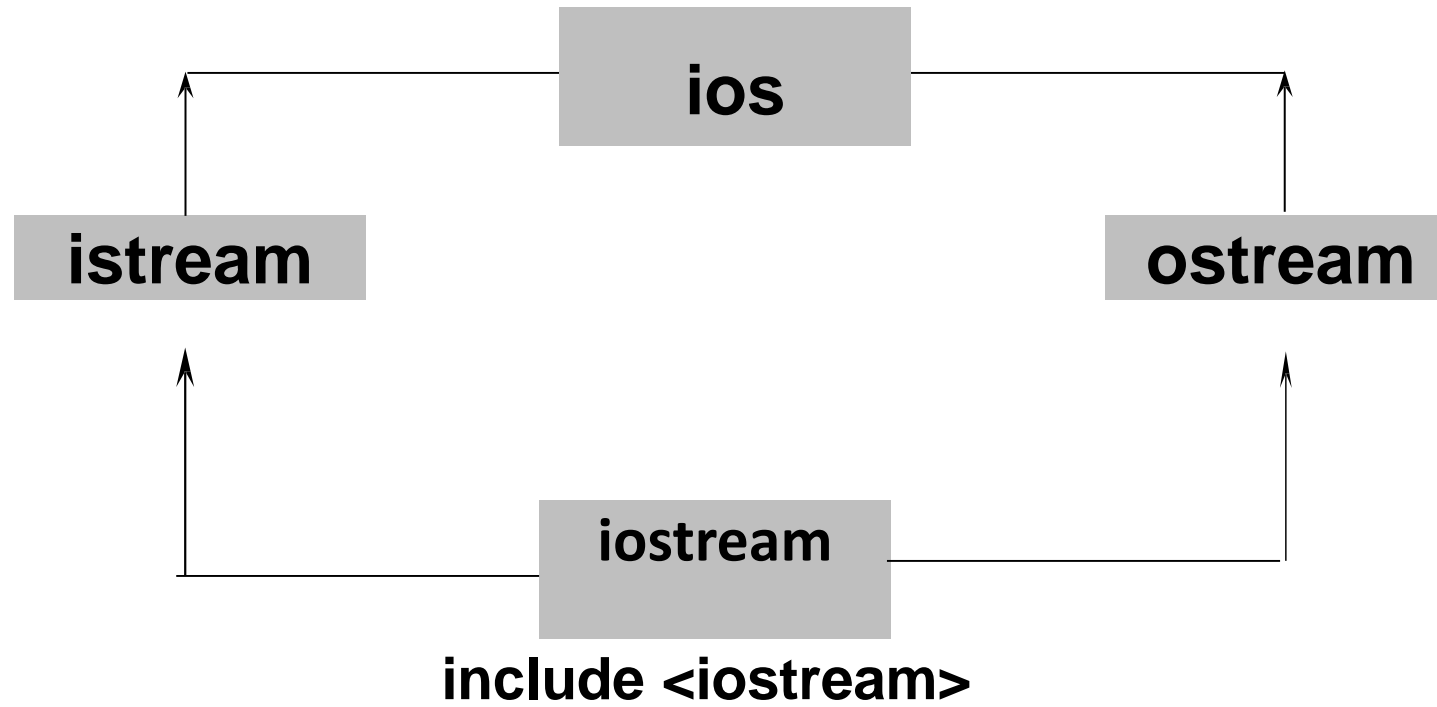
C++ I/O

- C++ defines its own object oriented I/O system
- C++ I/O system supplies consistent interface to the programmer independent of actual device.
- A stream is logical device that produces or consumes information.
- Two types of stream : Text stream, Binary stream.
 - Text stream is sequence of characters
 - Binary stream is sequence of bytes, no character translation

Data Stream



Stream class for console I/O



Unformatted I/O Operations

- >> and << are overloaded for all built-in data types in corresponding istream and ostream classes
- Use get()/put() to fetch/display a character including blank space, tab and newline character
 Use cin.get();
 Use cout.put(ch);
- Use getline() / write() to read / write a whole line of text
 Use cin.getline(line, size);
 Use cout.write(line, size);

Formatted I/O Operation

- Formatted I/O operations can be done
 - Using ios class functions and flags
 - Using manipulators

Using ios class

- `width()` → specify field width
- `precision()` → specify number of digits after decimal point
- `fill()` → specify character to be used to fill unused field
- `setf()` → used to set flags for say left justification / right justification clear
- `unsetf()` → the set flags

Using Manipulators

setw() →

- specify field width

setprecision() →

- specify number of digits after decimal point

setfill() →

- specify character to be used to fill unused field

setiosflags() →

- used to set flags for say left justification / right justification

resetiosflags() →

- clear the set flags

include <iomanip.h>

Using width(), precision(), fill()

```
cout.width(5);  
cout<<511;  
cout.width(5);  
cout<<15;
```

```
cout<<setw(5);  
cout<<511;  
cout<<setw(5);  
cout<<15;
```

		5	1	1				1	5
--	--	---	---	---	--	--	--	---	---

Using setf()

- Syntax : `cout.setf (arg1, arg2);`
- `arg1` is known as formatting flag of `ios` class
- `arg2` is known as bitfield, specifies the group to which formatting flag belongs

Using setf()

Formatting Flags

`ios::left ios::right`

`ios::scientific`

`ios::fixed ios::dec`

`ios::oct ios::hex`

Bit field

`adjustfield`

`adjustfield`

`floatfield`

`floatfield`

`basefield`

`basefield`

`basefield`

Using setf()

- few more formatted flags for which no bitfield is required

Formatted Flags	Description
<code>ios::showpos</code>	print + before positive numbers
<code>ios::showpoint</code>	show trailing decimal point and zeros
<code>ios::uppercase</code>	use uppercase letters for hex output

Example of Formatted I/O

```
int main() {  
    cout.setf(ios::showpoint);  
    cout.setf(ios::showpos);  
    cout.precision(3); cout.setf(ios::fixed,  
    ios::floatfield); cout.width(10);  
    cout<<275.5<<“\n”;  
    return;  
}
```

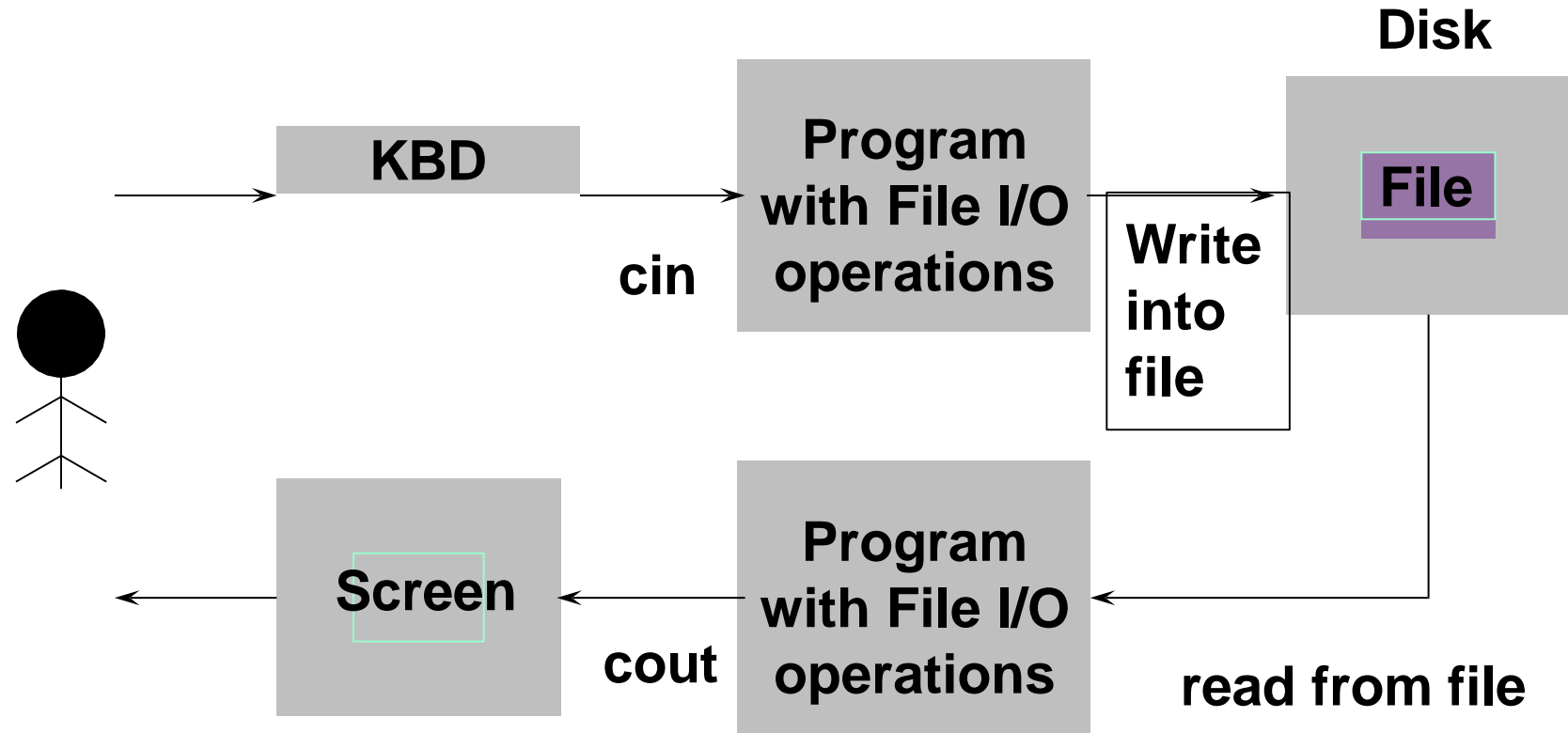
+			2	7	5	.	5	0	0
---	--	--	---	---	---	---	---	---	---

File I/O

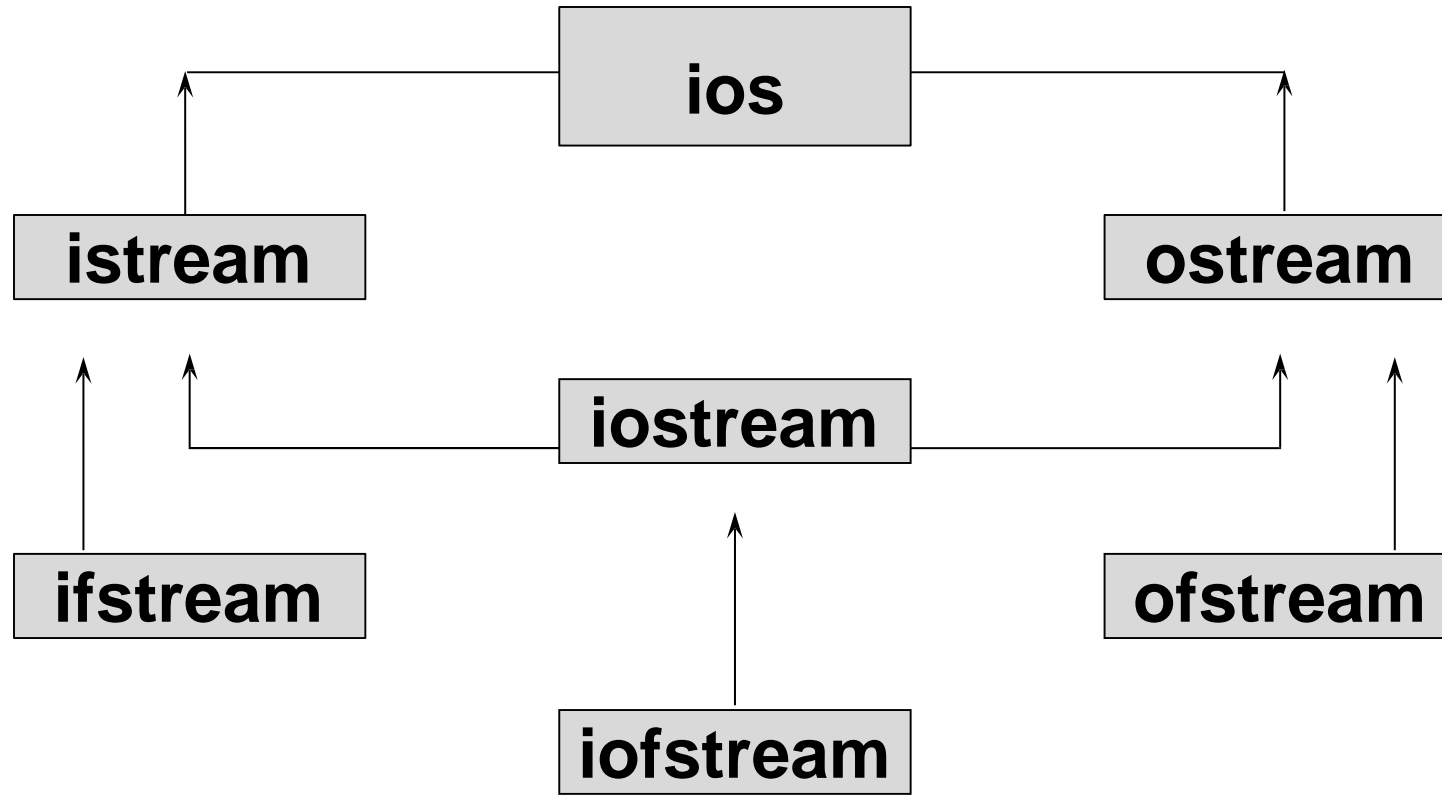
Data which outlives the program is stored on a disk in the form of files

A file is a collection of related data stored on a disk

Data Flow for File I/O



Stream Classes for File I/O



`include <fstream.h>`

Handling Files

- To read / write data to /from a file
 - Open a file
 - Do read write operations
 - Close file
- File can be opened in two ways
 - Using construction function of a class
 - Using member function open() of class

Using constructor to open a file

```
int main() {  
  
    ofstream outfile("result");  
    // outfile is object of class ofstream  
    // constructor will open it in write mode  
  
    char ch = ' ';  
    outfile.put( '1' ).put( ' ' ).put( ch );  
    outfile << "1 + 1 = " << (1 + 1) << endl;  
  
    return 0;  
}
```

inserts in result file 1) $1 + 1 = 2$

When file objects goes out of scope corresponding destructor is called which closes the file

Using open() to Open File

```
Void main {  
    ofstream outfile;           // create output stream  
    outfile.open("Result1");    // open a file  
  
    ...  
    outfile.close();           // to disconnect stream  
    outfile.open("Result2");  
  
    ...  
    outfile.close();  
}
```

To handle multiple files simultaneously create more streams

e.g. ifstream fin1, fin2;

More About open() : mode

Parameter	Meaning
<code>ios::app</code>	Append to an end of file
<code>ios::binary</code>	Binary file
<code>ios::in</code>	Open file for reading only (default arg)
<code>ios::nocreate</code>	Open fails if the file does not exist
<code>ios::noreplace</code>	Open fails if the file already exists
<code>ios::out</code>	Open file for writing only (default arg)
■ The mode can combine two or more parameters using bitwise OR	
■ e.g.: <code>fout.open("data", ios::app ios::nocreate)</code>	

Handling Files

- A pair of function `put()` and `get()` is used to access the file character by character
- A `getline()` is used to read a file line by line
- A pair of functions `write()` and `read()` are used to access the file in binary mode
- Objects with formatted i/os are handled in a binary file using `read()` / `write()` functions
- An `ifstream` object, such as `fin`, returns a value of 0 if error occurs in the file operation including the end-of-file condition

End

RTTI, Namespace

Objective

- typeid operator
- Casting Operator and RTTI
- Namespaces
- Introduction to templates.

typeid Operator

- Allows the type of an object to be determined at run time.
- It is an operator which returns reference to an object of `type_info` class.
 - `type_info` class describes type of object.
 - Include the standard template library header `<typeinfo>`
 - `typeid` returns type of built-in or user defined objects.

Run time type Identification

- Type checking is required at two points of time in a program
 - Compile time(known as static type checking).
 - Run time(known as dynamic type checking).
- RTTI enables identifying the type of an object during execution of a program.
- There are two types of RTTI operators:
 - typeid operator
 - dynamic_cast operator
- RTTI operators are runtime events for polymorphic classes and compile time events for all other types

Typeid in context with cEmployee class

```
int main()
{
    cEmployee *person;
    person=new cSalesPerson(.....);
    cout<<typeid (*person).name();    //cSalesPerson class

    if((typeid(*person)==typeid(cSalesPerson))
        .....
}
```

TypeCasting

- Explicit type conversion is referred to as cast.
- Casting operator is used to type cast
- Explicit type casting allows programmer to momentarily suspend type checking.
- Syntax

```
Cast_name<type>(expression);
```

static_cast

- Any Conversion which the compiler performs implicitly can be made explicit using static_cast.
- Programmer and Compiler are made aware of the loss of precision.
- Eg.

```
double d_val;  
int i_val;  
i_val=d_val; //implicit  
i_val=static_cast<int>(d_val); ///Explicit
```

Need of Dynamic Cast

```
Class A
{
    public:
    virtual void show()=0;
};
Class B :public A
{
    public:
    Void show()
    {
        .....
    }
    Void display()
    {
        .....
    }
}
```

```
Int main()
{
    A *Aptr=new B;
    Aptr->show();
    Aptr->display();
}
Aptr->display();---Gives Compile
time error.as display() is specific to
class B.
```

So what is the Solution?

Dynamic Cast

- Performs type conversion at run time.
 - Type-safe down casting
 - Guarantees the conversion of base class pointer to as derived class pointer.
- Works with pointer only and not with objects.

Dynamic Cast

```
int main()
{
    A *Aptr=new B();
    Aptr->show();
    B *bptr=dynamic_cast<B*>(Aptr);
    If(bptr)
        bptr->display();

    return 0;
}
```

Reinterpret_Cast

- Performs low level interpretation of bit pattern. It allows to edit individual bytes in memory by using bitwise/bitshift operator.
- Used to convert any data type to any other data type.
- It should not be used to cast down a class hierarchy or to remove the const or volatiles qualifiers.

```
Complex <double>*com;  
Char *pc=reinterpret_cast<char*>(com);
```

Reinterpret_Cast

- Performs low level interpretation of bit pattern. It allows to edit individual bytes in memory by using bitwise/bitshift operator.
- Used to convert any data type to any other data type.
- It should not be used to cast down a class hierarchy or to remove the const or volatiles qualifiers.

```
Complex <double>*com;  
Char *pc=reinterpret_cast<char*>(com);
```

Need of Namespace

- Software development is team effort.
- It is difficult to control names of variables, structures, classes, functions etc.
- Same variable names, structure names, class names lead towards re-declaration error.
- Same is the case when two or more functions with same signature are in the scope.
- The Solution is –Namespace.

Namespace

Namespace is a work area or a declarative region that attaches an additional identifier to all names declared inside it.

```
namespace space1
```

```
{
```

```
    void f();
```

```
    class A
```

```
    {
```

```
        ....
```

```
    }
```

```
}
```

```
namespace space2
```

```
{
```

```
    class B
```

```
    {
```

```
        ....
```

```
    };
```

```
    void f();
```

```
};
```

Using directive

- Using is resolved by compiler as unique

```
int main()
{
    using namespace space1;
    .....
    using namespace spac2;
}
```

End

Templates and STL

Objective

- Identify Need of templates.
- Describe function templates and class templates
- Distinguish between templates and macros.
- Name what is STL and its use.

Need of Templates

```
Void swap(int &a,int &b)
{
    int temp;
    temp=a;
    a=b;
    b=temp;
}
```

```
Void swap(Complex &c1,Comeplx &c2) }
{
    Complex temp;
    temp=c1;
    c1=c2;
    c2=temp;
}
```

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

Function



Function Overloading



Generic Function

Templates

- With function overloading same code needs to be repeated for different data types which leads towards waste of time & space.
- Templates enable us to define generic functions or classes which avoids above repetition of code for different data types .
- Generally templates are used if same algorithm works well for various data types eg sorting algorithms.
- There can be function templates or class templates.
- Function templates can be overloaded.

Function Templates

```
template <class type>ret type funName(parameter list)
```

- template is a keyword used to create generic functions.
- class type is a placeholder

```
template<class T>
Void swap(T &a,T &b)
{
    T temp;
    temp=a;
    a=b;
    b=temp;
}
```

```
int main()
{
    int i=10,j=20;
    swap(i,j);
    char ch1='a' ,ch2='z';
    swap(ch1,ch2);
}
```

Class template

Power of templates is reusability of code.

```
template<class T>
```

```
Class Demo
```

```
{
```

```
    int n;
```

```
    T *arr;
```

```
    public:
```

```
    Demo();
```

```
    void add_elements(T);
```

```
};
```

Example of Class template

Create a class calculator which should work to perform operations on Integer and Double types.

STL

- Standard Template Library.
- provides general-purpose, templated classes and functions that implement many popular and commonly used algorithms and data structures, including, for example, support for vectors, lists, queues, and stacks.
- Because the STL is constructed from template classes, the algorithms and data structures can be applied to nearly any type of data.
- At the core of the standard template library are three foundational items: *containers*, *algorithms*, and *iterators*.

Containers

- *Containers* are objects that hold other objects.
- Sequence Containers
 - E.g. vector, deque
- Associative Containers
 - E.g. map
- Each container class defines a set of functions that may be applied to the container.
- A list container includes functions that insert, delete, and merge elements. A stack includes functions that push and pop values.

Algorithms

- *Algorithms* act on containers.
- Their capabilities include initialization, sorting, searching, and transforming the contents of containers.
- They provide the means by which you will manipulate the contents of containers.

Iterators

- Iterators are objects that are, similar to pointers.

Iterators give you the ability to cycle through the contents of a container.

Types of Iterators:

Random Access

Bidirectional

Forward

Input

Output

You can increment and decrement Iterators.

You can apply the `*` operator to them. Iterators are declared using the **iterator** type defined by the various containers.

Vectors

The **vector** class supports a dynamic array.

you can use the standard array subscript notation to access its elements.

Any object that will be stored in a **vector** must define a default constructor.

Example:

```
vector<int> iv; // create zero-length int vector
```

```
vector<char> cv(5); // create 5-element char vector
```

```
vector<char> cv(5, 'x'); // initialize a 5-element char vector
```

```
vector<int> iv2(iv); // create int vector from an int vector
```

Vector

Vector Functions

`size()` - returns the current size of the vector

`begin()` - returns an iterator pointing to the beginning of vector

`end()` - returns an iterator pointing to the end of the vector

`push_back()` - inserts an element at the end of the vector

`Insert()` - used to insert an element in middle

`erase()` - used to remove an element from a vector

`Clear()` - removes all elements from the vector

List

The **list** class supports a bidirectional, linear list.

List can be accessed in a sequence only.

The following comparison operators are defined for **list**:

`==, <, <=, !=, >, >=`

Any data type that will be held in a list must define a default constructor. It must also define the various comparison operators.

List

Functions

Iterator begin()

Iterator end()

iterator erase(iterator *i*);

iterator erase(iterator *start*, iterator *end*);

iterator insert(iterator *i*, const T &val);

void pop_back();

void pop_front();

void push_back(const T &val);

void push_front(const T &val);

void remove(const T &val);

size_type size() const;

void sort();

Maps

The **map** class supports an associative container in which unique keys are mapped with values.

The power of a map is that you can look up a value given its key.

The following comparison operators are defined for **map**.

`==, <, <=, !=, >, >=`

Map

Functions

Iterator begin()

Iterator end()

iterator erase(iterator *i*);

iterator erase(iterator *start*, iterator *end*)

size_type erase(const key_type &k)

iterator find(const key_type &k);

iterator insert(iterator *i*, const value_type &val);

size_type size() const;

Map - Example

```
int main() { map<char, int> m;

// put pairs into map for(int i=0; i<26; i++) {
m.insert(pair<char, int>('A'+i, 65+i));
}

char ch;
cout << "Enter key: ";
cin >> ch;
map<char, int>::iterator p;

// find value given key p = m.find(ch);
if(p != m.end())
cout << "Its ASCII value is " << p->second;
else
cout << "Key not in map.\n";

return 0;
}
```

Algorithms

Algorithms act on containers.

Although each container provides support for its own basic operations, the standard algorithms provide more extended or complex actions.

They also allow you to work with two different types of containers at the same time.

To have access to the STL algorithms, you must include `<algorithm>` in your program.

Algorithms - Counting

binary_search

count

find

merge

max

Min

sort

Algorithm Example

```
int main()
{
    vector<bool> v;
    for(int i=0; i < 10; i++)
    {
        if(rand() % 2)
            v.push_back(true); else
            v.push_back(false);
    }
    cout << "Sequence:\n"; for(int i=0; i<v.size(); i++)
    cout << boolalpha << v[i] << " "; cout << endl;

    i = count(v.begin(), v.end(), true); cout << i << " elements are
    true.\n"; return 0;
}
```

Algorithms - Example

```
int main()
{
    vector<int> v;

    for(int i=0; i<10; i++) v.push_back(i);

    cout << "Initial: "; for(i=0; i<v.size(); i++)
    cout << v[i] << " "; cout << endl;

    reverse(v.begin(), v.end());

    cout << "Reversed: "; for(int i=0; i<v.size(); i++)
    cout << v[i] << " ";

    return 0;
}
```

Algorithm Examples

```
i = count(v.begin(), v.end(), true); cout << i << " elements are true.\n";
```

```
bool dividesBy3(int i)
{
    if( ( i % 3 ) == 0 )
        return true; return false;
}
```

```
i = count_if ( v.begin(), v.end(), dividesBy3); cout << i << " numbers are divisible by 3.\n";
```

```
// remove all spaces
remove_copy(v.begin(), v.end(), v2.begin(), ' ');
```

```
sort(v.begin(), v.end() );
```

Thank you