Polymorphism

- The fourth pillar of OOP -

Polymorphism – An Introduction

- The ability for objects of different classes related by inheritance to respond differently to the same message.
- Polymorphism is implemented in C++ via inheritance and virtual functions.
- This is also called as dynamic polymorphism. i.e. when a request is made through a base class pointer to use a virtual function, C++ chooses the correct overridden function in the appropriate derived class associated with the object
- Other types of polymorphism are obtained through function or operator overloading are called static polymorphism.
- This powerful feature of C++ helps in reducing complexity of the system.

Static Binding

- When the information is known to the compiler at the compiled time and therefore, compiler is able to select appropriate function for a particular call at the compile time itself, this is called early binding or static binding.
- Static binding is the compile-time determination of which function to call for a particular object based on the type of the formal parameter

Pointer to Objects

```
class A
{
      int x;
      public:
      void get()
      {
        cout<<"Enter the
value of x::";
            cin>>x;
      }

      void show()
      {
            cout<<x;
      }
};
```

```
int main()
{
  A *ptr,obj;
  ptr=&obj;
  ptr->get();
  ptr->show();
  return 0;
}
```

Example
(Pointer to Derived Class)

```
class A{
   public:
      int b;
      void show(){
      cout<<"b="<<b<<endl; }
};

class B: public A{
   public:
     int d;
     void show(){
      cout<<"b="<<b<<endl;
      cout<<"d="<<d<<endl; }
};
```

```
int main(){
  A *bptr, base;
  bptr=&base;
  bptr->b=100;
  bptr->show();
  B derived;
  bptr=&derived;
  bptr->b=200;
  bptr->d=300;
  bptr->show();
  B *dptr;
  dptr=&derived;
  dptr->d=300;
  dptr->show();
}
```

Dynamic Binding

- This is the run-time determination of which function to call for a particular object of a derived class based on the type of the argument
- Declaring a member function to be virtual instructs the compiler to generate code that guarantees dynamic binding
- Dynamic binding requires pass-by-reference

Virtual function

- If a function is declared as "virtual" in the base class then the implementation will not be decided at compile time but at the run time.
- Keyword virtual instructs the compiler to use late binding and delay the object interpretation
- Once a function is declared virtual, it remains virtual all the way down the inheritance hierarchy from that point, even if it is not declared virtual explicitly when a derived class overrides it
- But for the program clarity declare these functions as virtual explicitly at every level of hierarchy

Example- Dynamic Binding.

```
class Window{
   public:
     virtual void Create(){
      cout <<"Base Window"<<endl;
     }
};

class CommandButton : public
    Window{
  public:
    void Create(){
      cout<<"Derived Command
                    Button "<<endl;
    }
};
```

```
void main()
  {
      Window  *x, w;
       x = &w;
      x->Create();

      CommandButton cb;
      x=&cb;
      y->Create();
  }
```

OUTPUT

```
Base Window
Derived Command Button
```

Polymorphism

- When you use virtual functions, compiler store additional information about the types of object available and created
- Polymorphism is supported at this additional overhead
- Important :
  - virtual functions work only with pointers/references
  - Not with objects even if the function is virtual
  - If a class declares any virtual methods, the destructor of the class should be declared as virtual as well. Why??

Pure Virtual Functions

- It is a virtual function having no body

  syntax: `virtual void show()=0;`
- But this function can't be removed from the body. (compiler error)
- It is because the base class pointer is allowed to refer the base class members only.
- Pure virtual functions may be inherited/ overridden in derived classes.

Abstract Class

- A class never used for instantiation is called an abstract class.
- The sole purpose of an abstract class is to provide an appropriate base class for derivation of sub classes.
- A class will be treated as an abstract base class if at least one member function of the class is purely virtual
- Attempting to instantiate an object of an abstract is a syntax error

Abstract Class contd..

- If a class is derived from a class with a pure virtual function and if no definition for that function is supplied in the derived class also then …
  - That virtual function also remains purely virtual in the derived class.
  - The consequence is that the derived class again becomes an abstract class
  - i.e. It is not possible now to instantiate the derived class also!!!

Example of pure virtual Functions…

```
class Car
{
      public:
      virtual void brake()=0;
      virtual void steering()=0;
      virtual void regNo()=0;
};
```

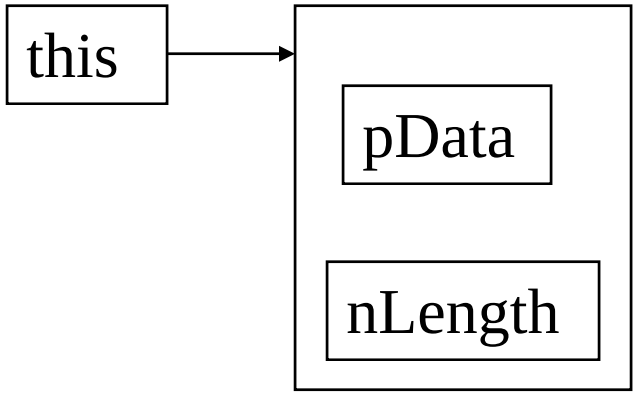
```
class Santro:public Car
{
     int regn_no;
     public:
     Santro(int reg)
     {
          regn_no=reg;
     }

 void brake()
     {
         cout<<"AIR BRAKE"<<endl;
     }

  void steering()
     {
         cout<<"POWER STEERING"<<endl;
     }
 void regNo()
     {
         cout<<"REGISTRATION:::\t"<<regn_no<<endl;
     }
};
```

```
class Hyundai: public Car
{
      int regn_no;
      public:
        Hyundai(int reg)
      {
           regn_no=reg;
      }
      void regNo()
      {
          cout<<"REGISTRATION:::"<<regn_no<<endl;
      }
      void brake()
      {
          cout<<"NORMAL BRAKE"<<endl;
      }
      void steering()
      {
          cout<<"ORDINARY STEERING"<<endl;
      }
};
```

```
int main()
{
    Car *ptr;
    Santro s(121);
    Hyundai h(234);

    ptr=&s;
    ptr->regNo();
    ptr->brake();
    ptr->steering();

    cout<<endl;

    ptr=&h;
    ptr->regNo();
    ptr->brake();
    ptr->steering();

    return 0;
}
```

"*this*" pointer

- Within a member function, the *this* keyword is a pointer to the current object, i.e. the object through which the function was called
- C++ passes a hidden *this* pointer whenever a member function is called
- Within a member function definition, there is an implicit use of *this* pointer for references to data members



CStr object
(*this)

Data member reference	Equivalent to
pData	this->pData
nLength	this->nLength

"this" Pointer

- Every object has access to its own address through a pointer called 'this'
- The this pointer is passed as an implicit first argument on every non-static member function call for an object.
- The this pointer is implicitly used to refer both the data and function members of an object
- It can also be used explicitly;

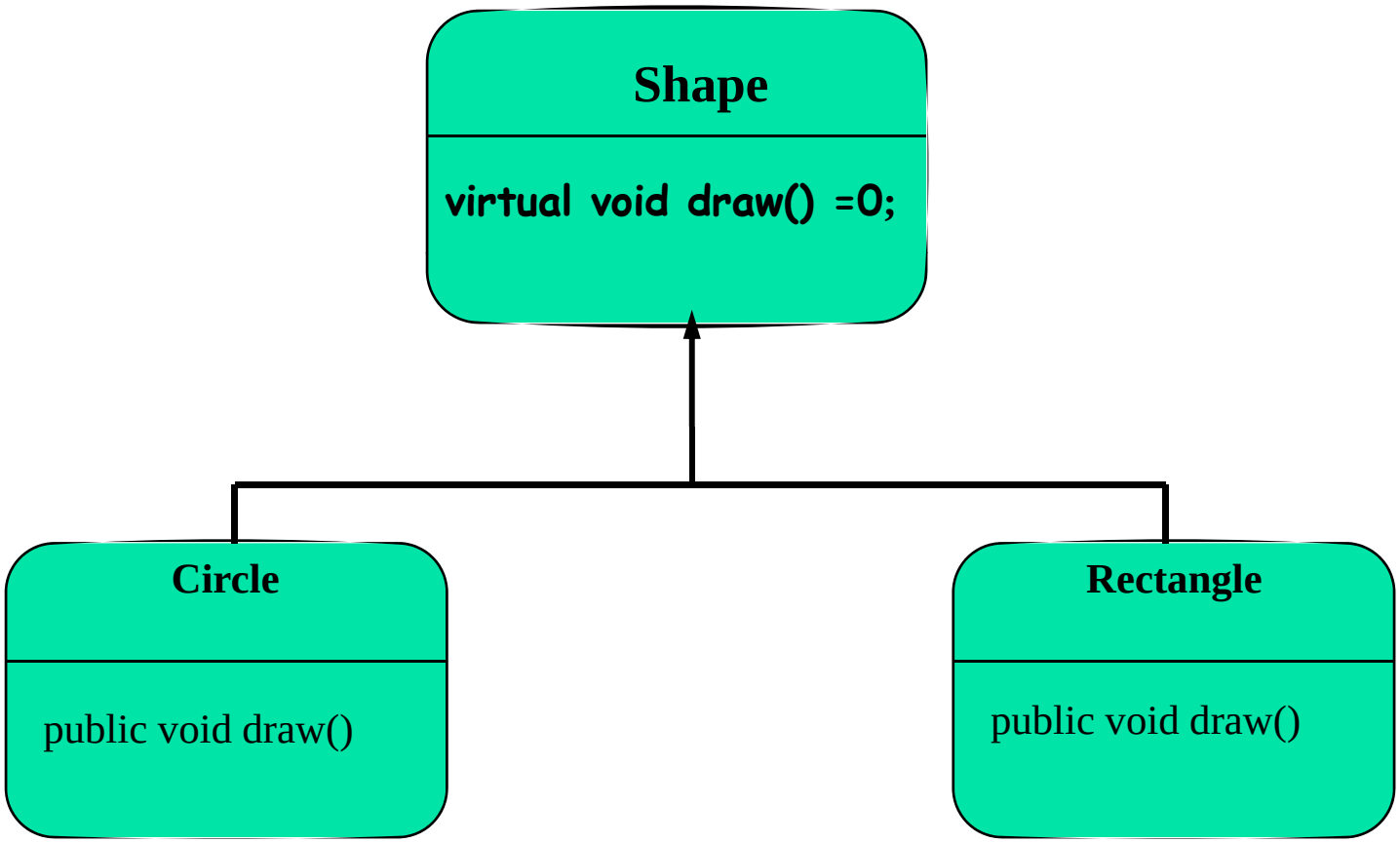
Example: (*this).x=5; or this->x=5;

EXAMPLE OF this POINTER

```
class A
{
      int i;
      float f;

      public:

      A(int i, float f)
      {
           this->i=i;
           this->f=f;
      }
```

```
void set()
      {
       cout<<"Value of i:"<<i<<endl;
       cout<<"Value of f:"<<f;
      }
};

int main()
{
    A obj(10,20);
    obj.set();
}
```

Example


Shape
virtual void draw() =0;
Circle
public void draw()
Rectangle
public void draw()

Example- contd..

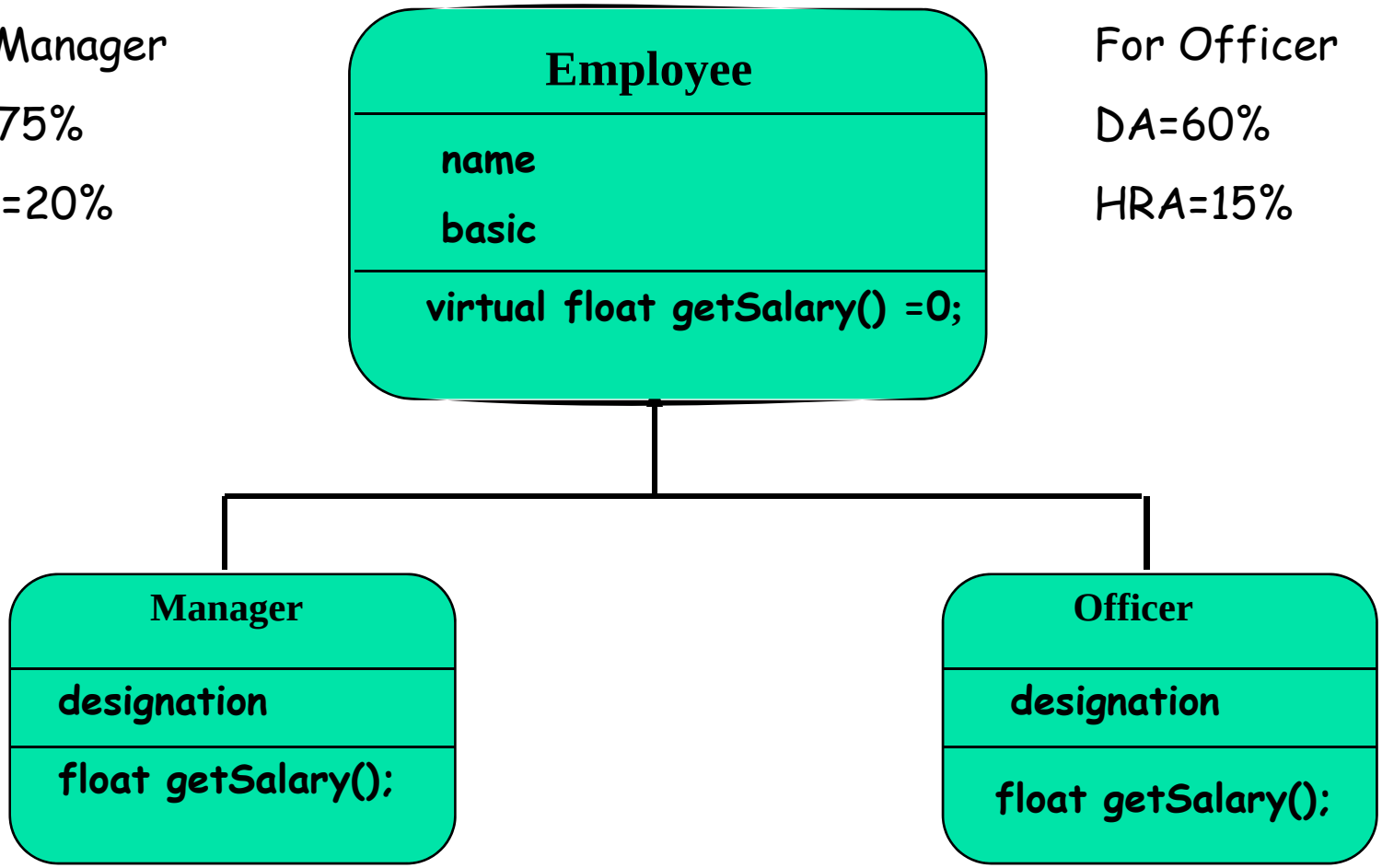
```
class Circle : public Shape {
   public :
   void draw(){ //Override Shape::draw()
     cout << “I am a Circle” << endl;
   }
class Rectangle : public Shape {
   public :
   void draw(){ // Override Shape::draw()
     cout << “I am a  Rectangle” << endl;
   }
```

ASSIGNMENT

For Manager

DA=75%

HRA=20%

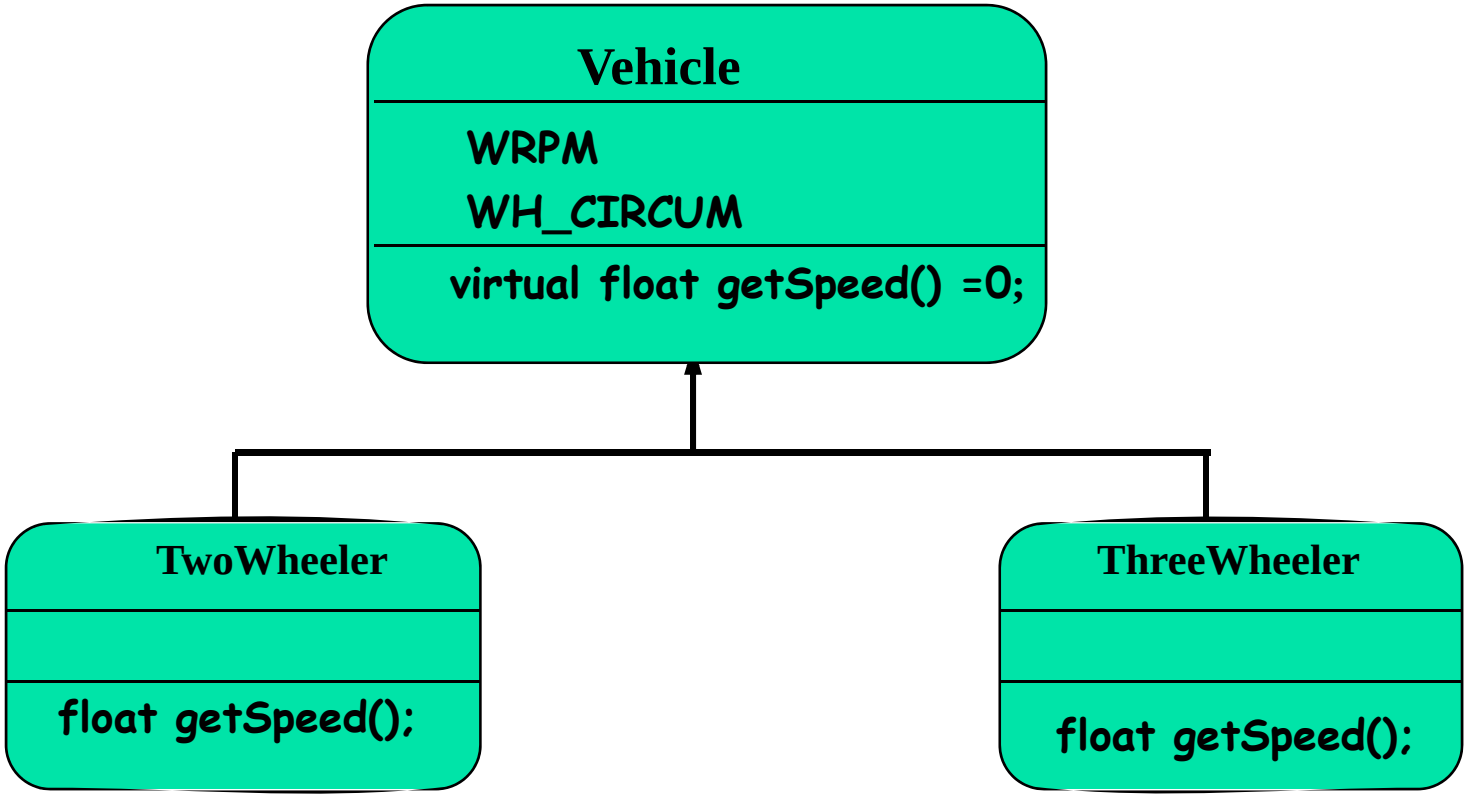


For Officer

DA=60%

HRA=15%

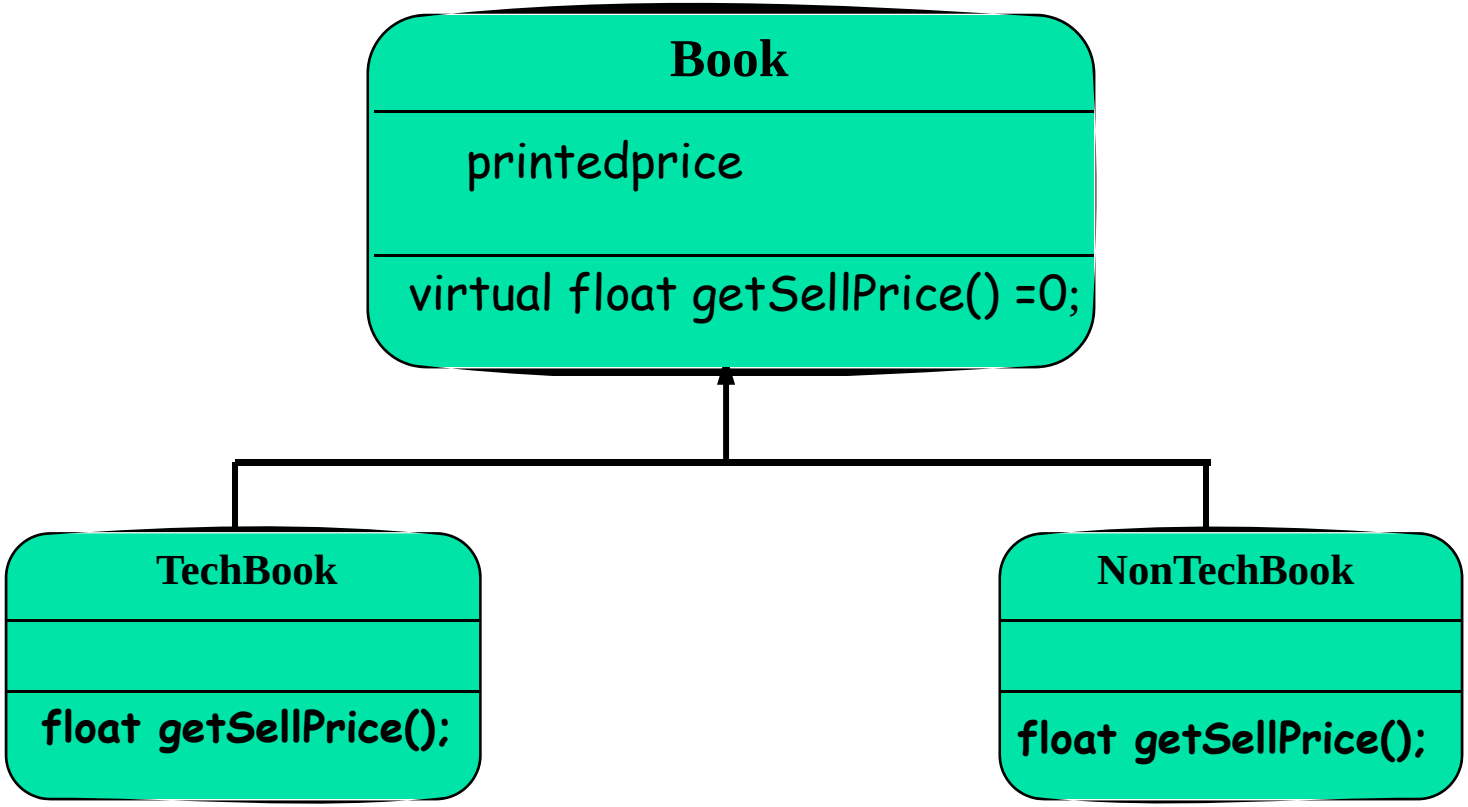
ASSIGNMENT



WRPM- Wheel rotation per minute,

WH_CIRCUM- Wheel circumference

ASSIGNMENT



Tech Book discount- 15%

Non Tech Book discount- 10%

ASSIGNMENT

- Use base pointer to call the overridden functions of derived class
- Handle Exception for each assignments