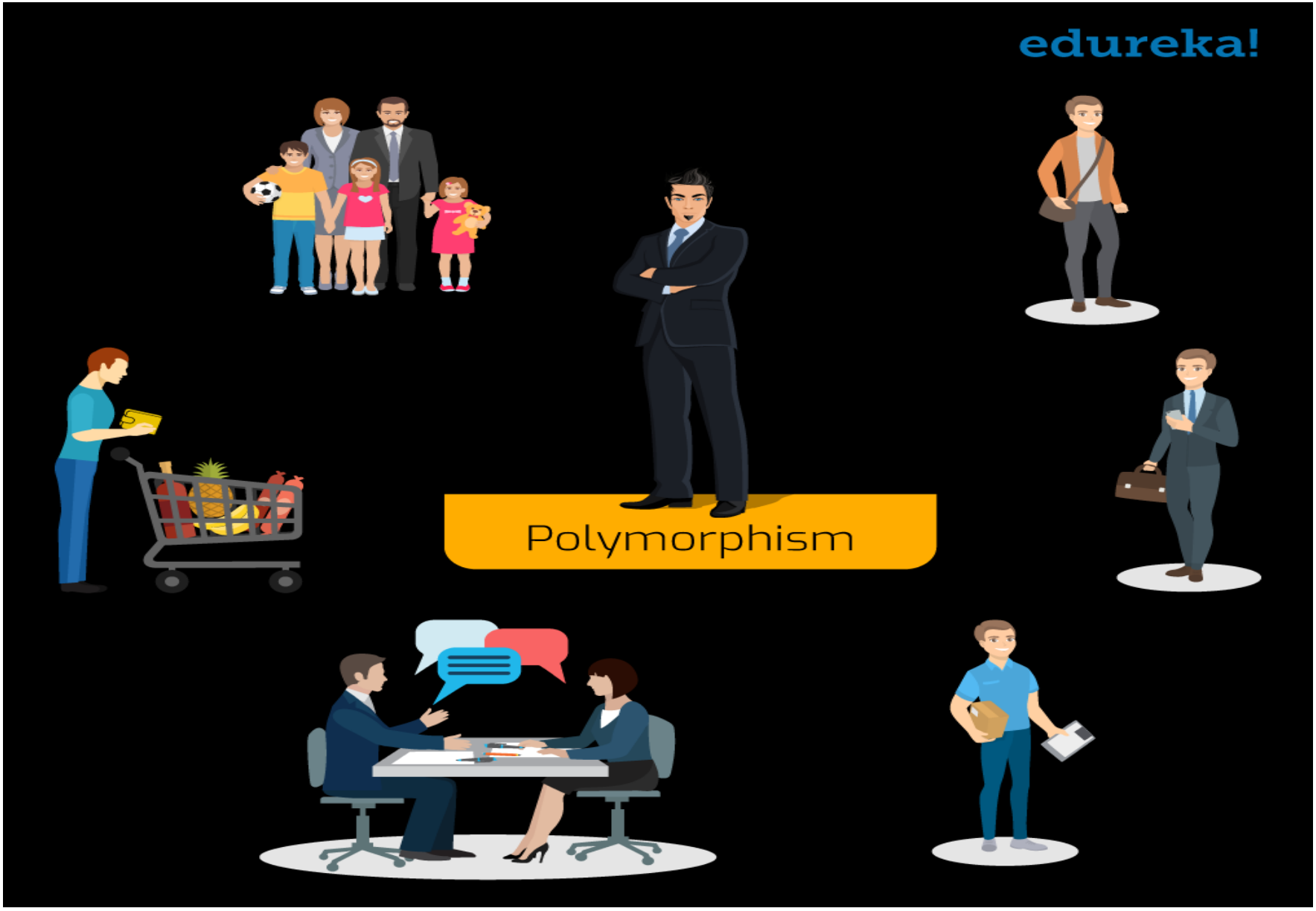
Unit III: Polymorphism

No	Hrs	Contents
1	**1**	Introduction to Polymorphism, Types of Polymorphism, Operator Overloading- concept of operator overloading.
2	**2**	Overloading Unary Operators, Overloading Binary Operators
3	**1**	Data Conversion, Type casting (implicit and explicit), Pitfalls of Operator Overloading and Conversion, Keywords explicit and mutable.
4	**2**	Function overloading Run Time Polymorphism- Pointers to Base class, virtual function and its significance in C++
5	**1**	pure virtual function and virtual table, virtual destructor, abstract base class.


edureka!
Polymorphism

Polymorphism (Unit III)

- If we take example of Human behavior than
  - In the class room you behave as a Teacher
  - In the shopping mall you behave as a Customer
  - At home you behave as a Father/Mother
  - In the train/bus you behave as a Passenger

  Thus the same person with different behavior based on the location.

  If behavior is a function then based on the parameter location its functionality changes.

What is Polymorphism?

Ability to take more than one form.

A function to exhibit different behavior with different instances.

Behavior dependent on type and number of parameters.


"Now Speak!"
"Quack"
"Woof"
"Meow"


POLYMORPHISM IN THE MALABAR PIT VIPER
GORGEOUS GREEN
ORNATE ORANGE
BEAUTEOUS BLUE
GLORIOUS GREY
PREPOSSESSING PURPLE
BORED-OF-ALLITERATION BROWN
www.greenhumour.com

Types of Polymorphism

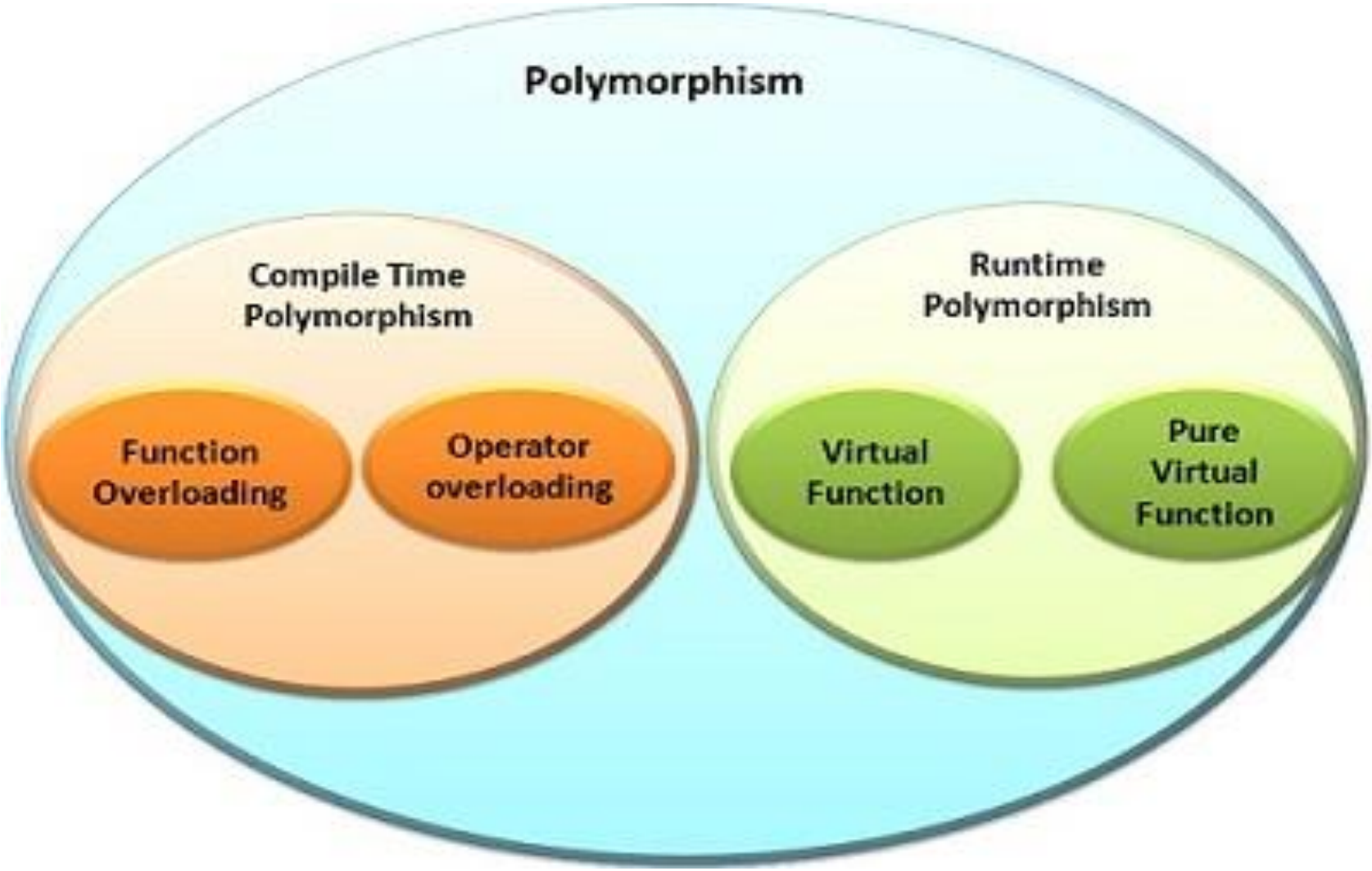
Static/Compile time Polymorphism:

- Memory allocation at compile time.
- Object is bound to the function call at the time of compilation.
- Also termed as Early binding.
  - Function Overloading
  - Function Overriding
  - Constructor Overloading
  - Operator Overloading

Types of Polymorphism

Dynamic/Runtime Polymorphism:

- Memory allocation at run time.
- Object is bound to the function call at the time of execution.
- Also termed as Late binding
  - Virtual Functions


Polymorphism
Compile Time Polymorphism
Function Overloading
Operator overloading
Runtime Polymorphism
Virtual Function
Pure Virtual Function

Operator Overloading

- Process of making an operator to exhibit different behavior for different instances.
- Mechanism to give a special meaning to an operator.
- Provides a flexible option for creating new definitions for different operators when they operate on class objects.

Rules for Operator Overloading

- Only existing operators. New operators cannot be created.
- One of the operand has to be user defined type.
- Basic meaning cannot be changed.
- Syntax rules for the original operator must be followed.
- Few operators cannot be overloaded.

General Process

- Defining a class with the data types to be used for overloading operation.
- Declaration of operator overloading function. This can be either as a member function or as a friend function.
- Defining the operator overloading function for implementing the required operations.

General Syntax

Return type class name :: **operator** op (arg list)

{

}

Eg: float operator + (float)

{ }

vector operator – (vector)

{ }

Why Operator Overloading

- Any functionality performed using operator overloading can also be done without operator overloading.
- But operator overloading makes program more readable as the operator which are used for basic data types can also be used for user-defined data types.

Overloading Unary Operators

- Can be overloaded as a member function or as a friend function.
- As a member function no explicit argument is taken, neither returned.
- As a friend function takes one reference argument which is object of relevant class.

Unary Operators (Example)

- Program To Overload Unary Minus (-)

```
class example { int a,b;
    public: void input() { cout<<"Enter a and b: ";
    cin>>a>>b; }
   void operator -();  // (example &x)
   void display() {
    cout<<"a="<<a<<endl<<"b="<<b<<endl; } };
void example:: operator -()  // (example &x)
{     a=-a;    b=-b; }        // x.a = -x.a;   x.b = -x.b;
```

Unary Operators (Example)

```
int main()
{ example e;
  e.input();
  cout<<"Before overloading unary minus  operator“ <<endl;
                e.display();
    -e;
    cout<<"After overloading unary minus operator“ <<endl;
   e.display();
   return 0; }
```

Overloading Binary Operators

- Can be overloaded as a member function or as a friend function.
- As a member function one explicit argument is taken.
- As a friend function takes two explicit arguments which may be objects of the relevant class.

Overloading Bin Op as friend function

- Overloading the operator as a member function fails in typical situations where left hand side of the overloaded operator has to be a built in data type.
- Since left hand operand responsible for invoking the function should a object.
- In such situations overloading the operator as friend function becomes essential as it allows both the approaches.

Binary Operators (Example)

```
const size = 5;
class vect
{  int v[size];
public: vect()
{ for(int i=0;i<size; i++)
   v[i] = 0;  }
vect (int *x)
{ for(int i=0;i<size; i++)
   v[i] = x[i];
   }
```

```
vect operator *(int a, vect b)
   {   vect c;
       for(int i=0; i< size; i++)
       c.v[i] = a * b.v[i];
       return c;      }
vect operator *(vect b, int a)
      {   vect c;
          for(int i=0; i< size; i++)
          c.v[i] = b.v[i]*a;
          return c;    }
```

Binary Operators (Example)

```
istream & operator >> (istream &din, vect &b)
{for(int i=0; i<size; i++)  din >> b.v[i];  return(din); }
ostream & operator << (ostream &dout, vect &b)
{for(int i=0; i<size; i++) dout << b.v[i];  return(dout); }
int x[size]={2,4,6,8,10};
int main() { vect m; vect n = x;
cin >> m;   cout << m;
vect p,q;  p = 2 * m;  q = n * 2;
cout << p << q;   return 0;  }
```

Limitations/ Restrictions

- Precedence and Associativity of an operator cannot be changed.
- Arity (numbers of Operands) cannot be changed. Unary operator remains unary, binary remains binary etc.
- No new operators can be created, only existing operators can be overloaded.
- Cannot redefine the meaning of a procedure.
- Cannot change how integers are added.

Operators ! overloaded

Member Access operator .

Scope resolution operator ::

Size operator sizeof

Ternary operator ? :

Pointer to member access operator .*

These operators cannot be overloaded since these operators take names (like int, class name) as their operand instead of values.

Pitfalls of Op Overloading and Conversion

- Code can be innovative and readable but it may also be hard to understand.
- Similar meaning and syntax should be given to the overloaded operators else it makes difficult to understand the functionality.
- Ambiguity should be avoided.
- Remember that all operators cannot be overloaded.

Type Casting

Type casting/Automatic/implicit type conversion:

Conversion that takes place automatically or implicitly.

Built-in to built-in: eg. int a; float d = 7.2361

a = d;

User defined to user defined (same type)

x1 = x2 + x3 or x3 = x1;

Works smoothly if x1,x2,x3 are objects of same class. (Compatible types)

Data Conversion

For incompatible types the conversion routines should be defined by the programmer to convert:

1. Basic to user-defined;
   - Through Constructors that take a argument of the type to be converted.
2. User-defined to Basic;
   - Overloaded casting operator which uses a general syntax as operator type-name() { }
3. User-defined to another user-defined.

   Constructor or Conversion function

When to use What?

- Using a one argument constructor or the conversion function the selection is up to the programmer.
- When conversion function is used conversion takes place in the source class and the result is passed to the destination class i.e. type-name refers to destination class in:

  operator type-name()

- When constructor is used the argument belongs to source-class and is passed to the destination class for conversion.

Explicit and Mutable

Explicit: restricts implicit conversion of argument received in a constructor to object of the class.

Mutable: allows a const data member to be modified. Useful in classes that have secret implementation details that do not contribute to the logical value of an object.

Mutable (Example)

```
class TestMutable { public:
TestMutable(int v=0)
{ value = v;  }
int getvalue() const { return ++value;  }
private:  mutable int value;   };
int main() { const TestMutable test(59);
cout << “Value on first call” << test.getvalue();
cout << “Value on second call” << test.getvalue();
return 0;
}
```

Function Overloading

- Matches the prototype having same number and type of arguments to call the appropriate function for execution.
- Steps involve for function selection:
- Tries to find an exact match with type of actual arguments same as formal.
- If exact match is not found, compiler uses **integral promotions** to the actual arguments like: char to int, float to double etc. and tries to find a match.

Function Overloading

- If this fails compiler tries to use **built-in (implicit) conversions** to the actual arguments and then tries to find a match. If the conversion has multiple matches compiler generates an error message.

  eg. double Square (double a)

  long Square (long p)

  If there is a call Square(10); will cause an error as int argument can be converted to either long or double that creates an ambiguity.

Function Overloading

- If all the above steps fail compiler tries the user defined conversion in combination with integral promotions and built-in conversions to find a match.
- If it does not find a match in spite of all the above conversions and promotions then it report as an error.

Runtime Polymorphism

- Memory allocation at run time.
- Object is bound to the function call at the time of execution.
- Also termed as Late binding in which function call is resolved at runtime. Hence, compiler determines the type of object at runtime, and then binds the function call.

Virtual Functions

- A function in base class, which is overridden in the derived class.
- Tells the compiler to perform Late Binding on this function.
- Virtual Keyword is used to make a member function of the base class Virtual.

Pointers

- A pointer can point to an object of a class.
- Student *ptr1; is a pointer that can point to an object of class Student.
- Object pointers are useful in creating objects at runtime. Access the public members of an object. Eg. ptr1->getdata(); ptr1->display();
- A base class pointer can point to any type of derived object, but the reverse is not true.
- Incrementing or decrementing a base class pointer pointing to derived class object cannot make it to point to the next object of derived class. Increment or decrement is relevant to its base type.

Rules for Virtual Function

- Should be member of some class.
- Cannot be static, can be accessed using object pointers, can be friend of another class.
- Prototype of the base class version must be identical to the derived class version.
- If defined in the base class, need not be necessarily redefined in the derived class. In such case call invokes the base function.
- Constructors cannot be virtual, Destructors can be virtual.

Virtual Function: Significance

- Virtual function makes C++ compiler to determine which function to be used at execution time based on the type of object pointed and not the type of pointer.
- Making the base pointer to point to different derived objects, different versions of the virtual function can be executed.

Example Virtual Functions

```
class BaseClass { //Abstract class
   public: virtual void Display1()=0; //Pure virtual
   virtual void Display2()=0; //Pure virtual
   void Display3() { cout<<“This is Display3() method
    of Base Class"; } };
class DerivedClass : public BaseClass
{ public: void Display1()
 { cout<<“This is Display1() method of Derived Class"; }
void Display2() { cout<<“This is Display2() method of
    Derived Class"; } };
```

Example Virtual Functions

```
int main()
{ DerivedClass D;
  D.Display1();
  D.Display2();
  D.Display3();
   return 0;
 }
```

```
int main()
{  BaseClass *bptr;
     Derived D;
 bptr = &D;
 bptr->Display1();
 bptr->Display2();
 bptr->Display3();
}
```

Example Virtual Functions

```
class Shape { protected: double width, height;
   public: void set_data (double a, double b)
{ width = a; height = b; }
virtual double area() = 0; };
class Rectangle: public Shape
{ public: double area ()
{ return (width * height); } };
class Triangle: public Shape
{ public: double area ()
{ return (width * height)/2; } };
```

Example Virtual Functions

```
int main ()
 { Shape *sPtr; Rectangle Rect;
     sPtr = &Rect; sPtr -> set_data (5,3);
 cout << "Area of Rectangle is “<< sPtr -> area();
   Triangle Tri; sPtr = &Tri;
 sPtr -> set_data (4,6);
cout << "Area of Triangle is " << sPtr -> area();
return 0; }
```

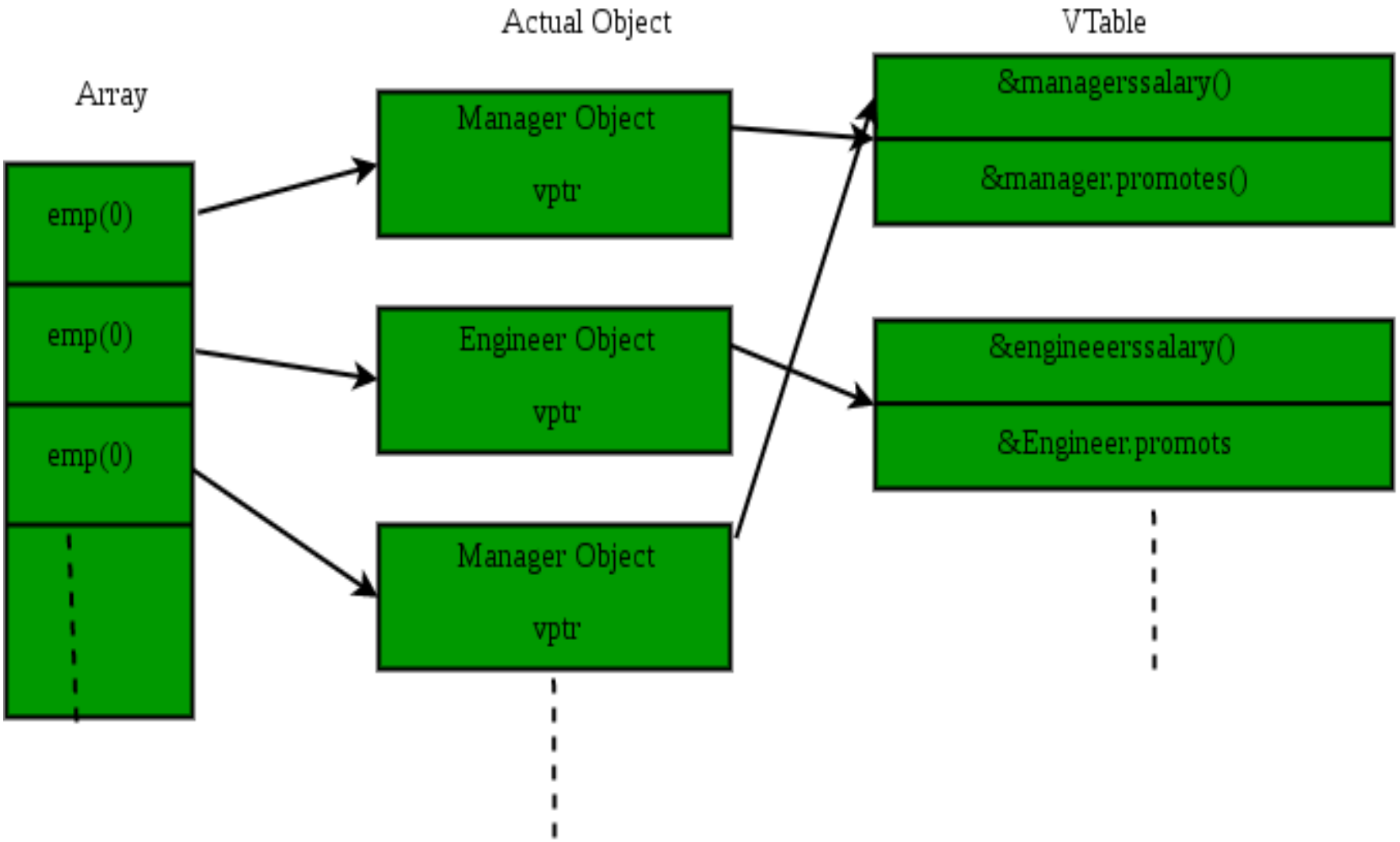
Pure Virtual Function

- Pure virtual Functions are virtual functions with no definition.
- They start with virtual keyword and ends with = 0
- Syntax virtual show() = 0;
- Pure virtual function is also known as abstract function

Virtual Table

- For every class that contains virtual functions, the compiler constructs a virtual table, vtable.
- The vtable contains an entry for each virtual function accessible by the class and stores a pointer to its definition.
- Only the most specific function definition callable by the class is stored in the vtable.

Virtual Table



Virtual Destructors

- If we create an object of a derived class dynamically using the base class pointer.
- If we destroy this derived object with non-virtual destructor explicitly using the delete operator in association with the base class pointer then only the destructor of the base class is executed and not the derived class destructor.

Virtual Destructors

- The reason behind the non execution of derived class destructor is that the compiler uses static binding when calling the destructor.
- The solution to this problem is to make the destructor to be dynamically bound which is achieved by making the base class destructor as virtual
- Now if you destroy the derived object explicitly using delete operator in association with the base class pointer, the destructor of appropriate derived class is called depending upon the object to which pointer of base class points.

Abstract Class

- A class which contains at least one Pure Virtual function in it.
- Used to provide an Interface for its sub classes.
- Its derived Class must provide definition to all the pure virtual functions, otherwise it becomes abstract class.
- A class with at least one pure virtual function is called abstract class.

Abstract Class

- Abstract class cannot be instantiated, but pointers and references of Abstract class type can be created.
- Abstract class can have normal functions and variables along with a pure virtual function.
- Abstract classes are mainly used for Up casting, so that its derived classes can use its interface.

Case Study

Study about use of C++ SDKs wrappers for Java and .Net

Thank you