ISSN:0975-9646
Devaendra Gahlot et al, / (IJCSIT) International Journal of Computer Science and Information Technologies, Vol. 2 (1) , 2011, 569-571

Run Time Polymorphism Against Virtual Function in Object Oriented Programming

Devendra Gahlot[#1], S. S. Sarangdevot#[2], Sanjay Tejasvee[#3]

[#]*Department of Computer Application, Govt. Engineering College Bikaner, Bikaner, Rajasthan, India.*

[#]*Department of IT & CS, Janardan Rai Nagar Rajasthan Vidyapeeth University, Pratap Nagar, Udaipur, Rajasthan, India)*

[#]*Department of Computer Application,Govt. Engineering College Bikaner,Bikaner ,Rajasthan,India.*

***Abstract*- The Polymorphism is the main feature of Object Oriented Programming. Run Time Polymorphism is concept of late binding; means the thread of the program will be dynamically executed, according to determination of the compiler. In this paper, we will discuss the role of Run Time Polymorphism and how it can be effectively used to increase the efficiency of the application and overcome complexity of overridden function and pointer object during inheritance in Object Oriented programming.**

1. Introduction to Polymorphism In Object-Oriented Programming

Subtype polymorphism, almost universally called just polymorphism in the context of object-oriented programming, is the ability of one type, A, to appear as and be used like another type, B. This article is an accessible introduction to the topic, which restricts attention to the object-oriented paradigm. The purpose of polymorphism is to implement a style of programming called message-passing in the literature[citation needed], in which objects of various types define a common interface of operations for users.

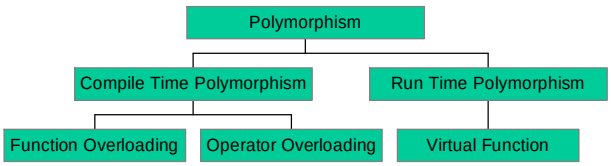
In strongly typed languages, polymorphism usually means that type A somehow derives from type B, or type C implements an interface that represents type B. In weakly typed languages types are implicitly polymorphic.

Operator overloading of the numeric operators (+, -, *, and /) allows polymorphic treatment of the various numerical types: integer, unsigned integer, float, decimal, etc; each of which have different ranges, bit patterns, and representations. Another common example is the use of the "+" operator which allows similar or polymorphic treatment of numbers (addition), strings (concatenation), and lists (attachment). This is a lesser used feature of polymorphism. The primary usage of polymorphism in industry (object-oriented programming theory) is the ability of objects belonging to different types to respond to method, field, or property calls of the same name, each one according to an appropriate type-specific behavior. The programmer (and the program) does not have to know the exact type of the object in advance, and so the exact behavior is determined at run-time (this is called late binding or dynamic binding).

The different objects involved only need to present a compatible interface to the clients' (calling routines). That is, there must be public or internal methods, fields, events, and properties with the same name and the same parameter sets in all the super classes, subclasses and interfaces. In principle, the object types may be unrelated, but since they share a common interface, they are often implemented as subclasses of the same super class. Though it is not required, it is understood that the different methods will also produce similar results (for example, returning values of the same type).

Polymorphism is not the same as method overloading or method overriding.[1] Polymorphism is only concerned with the application of specific implementations to an interface or a more generic base class. Method overloading refers to methods that have the same name but different signatures inside the same class. Method overriding is where a subclass replaces the implementation of one or more of its parent's methods. Neither method overloading nor method overriding are by themselves implementations of polymorphism.[2]

Type of Polymorphism



2.Inheritance with Polymorphism(Virtual Function)

If a Dog is commanded to speak ( ), it may emit a bark, while if a Pig is asked to speak ( ), it may respond with an oink. Both inherit speak ( ) from Animal, but their subclass methods override the methods of the super class, known as overriding polymorphism. Adding a walk method to Animal would give both Pig and Dog objects the same walk method.

Inheritance combined with polymorphism allows class B to inherit from class A without having to retain all features of class A; it can do some of the things that class A does differently. This means that the same "verb" can result in different actions as appropriate for a specific class. Calling code can issue the same command to their super class or interface and get appropriately different results from each one.

C++

```
#include <iostream.h>
#include <string.h>
class Animal
{
        public:
        Animal(const string& name) : name(name) {}
        virtual string talk() = 0;
        const string name;
};
class Cat : public Animal
{
        public:
        Cat(const string& name) : Animal(name) {}
        virtual string talk() { return "Meow!"; }
};
 class Dog : public Animal
{
        public:
        Dog(const string& name) : Animal(name) {}
        virtual string talk() { return "Arf! Arf!"; }
};
 // prints the following:
// Missy: Meow!
// Mr. Mistoffelees: Meow!
// Lassie: Arf! Arf!
int main()
{
        Animal* animals[] ={
                new Cat("Missy"),
                new Cat("Mr. Mistoffelees"),
                new Dog("Lassie")
        };
 for(int i = 0; i < 3; i++){
cout << animals[i]->name << ": " << animals[i]->talk() << endl;
delete animals[i];}
return 0;
}
```

Note: that the talk() method is explicitly declared as virtual. This is because non-polymorphic method calls are very efficient in C++ and a polymorphic method call is roughly equivalent to calling a function through a pointer stored in an array.[3] To take advantage of the efficiency of non-polymorphic calls, all method calls are treated as non-polymorphic, unless explicitly marked as virtual by the developer.

Java

```
interface Animal
{
   String getName();
   String talk();
}
abstract class AnimalBase implements Animal
{
        private final String name;
         protected AnimalBase(String name){
         this.name = name;
         }
         public String getName(){
         return name;
            }
}
 class Cat extends AnimalBase
{
   public Cat(String name) {
      super(name);
   }
    public String talk(){
      return "Meowww!";
   }
}
 class Dog extends AnimalBase
{
   public Dog(String name) {
      super(name);
   }
    public String talk(){
      return "Arf! Arf!";
   }
}

public class TestAnimals
{
   // prints the following:
   // Missy: Meowww!
   // Mr. Mistoffelees: Meowww!
   // Lassie: Arf! Arf!
   public static void main(String[] args)
   {
      Animal[] animals =  {
         new Cat("Missy"),
         new Cat("Mr. Mistoffelees"),
         new Dog("Lassie")
         };
      for (Animal a : animals){
         System.out.println(a.getName() + ": " + a.talk());
      }
   }
}
```

3. PARAMETRIC POLYMORPHISM

In computer science, the term polymorphism has several different but related meanings; one of these, known as parametric polymorphism in type system theory and functional programming languages, is known as generic programming in the Object Oriented Programming Community and is supported by many languages including C++, C# and Java.

Generics allow compile-time type-safety and other benefits and/or disadvantages depending on the language's implementation.

C++ implements parametric polymorphism through templates. The use of templates requires the compiler to generate a separate instance of the templated class or function for every permutation of type parameters used with it, which can lead to code bloat and difficulty debugging. Benefit C++ templates have over Java and C# is that they allow for template metaprogramming, which is a way of pre-evaluating some of the code at compile-time rather than run-time.

Java parametric polymorphism is called generics and implemented through type erasure.
C# parametric polymorphism is called generics and implemented by reification, making C# the only language of the three which supports parametric polymorphism as a first class member of the language. This design choice is leveraged to provide additional functionality, such as allowing reflection with preservation of generic types, as well as alleviating some of the limitations of erasure (such as being unable

to create generic arrays). This also means that there is no performance hit from runtime casts and normally expensive boxing conversions. When primitive and value types are used as generic arguments, they get specialized implementations, allowing for efficient generic collections and methods.

4. DISCUSSION

As far as research is concern, we are going to discuss run-time polymorphism in c++. At time of class inheritance, if we override the function means redefinition of function through base class to related derived class. At the time of calling overridden function of desired class means run-time polymorphism; compiler will decide that which function going to be called. Basically to do this we have to use virtual function with pointer object. Which is little bit complicated, because the use of pointer used to complex.

Exmaple.

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
class a{
        public:
        virtual  void show(){
        cout<<"show a class";
        }
};
class b:public a
{
        public:
         virtual void show(){
        cout<<"show b class";
        }
};
class c:public b
{
        public:
        void show(){
        cout<<"show c class";
        }
};
void main()
{
clrscr();
a *ap,A;
b B;
c C;
cout<<"memeory ="<<sizeof(ap)+sizeof(A)+sizeof(B)+sizeof(C);
ap=&A;
ap->show();
ap=&B;
ap->show();
ap=&C;
ap->show();
getch();
}
```

To call the show function of related class. We must use pointer object with reference of object of the class of related function. The research is concern that without using virtual and pointer object; the process can be implemented. When we adopt nesting of classes with law memory requirement rather then virtual or pointer object.

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
class a
{
        public:
        void show(){
        cout<<"show a class";
        }

        class b{
                public:
                void show(){
                cout<<"show b class";
                }
        }B;
        class c{
                public:
                void show()
                {
                cout<<"show c class";
                }
        }C;
};
void main()
{
clrscr();
cout<<sizeof(A)+sizeof(A.C)+sizeof(A.B);
A.show();
A.B..show();
A.C.show();
getch();
}
```

5. CONCLUSION

When we implement multilevel inheritance in C++ To avoid complexity due to virtual function or pointer object and their required memory management by compiler can be overcome using nesting of class. In this method, you also can define visibility of data and method like inheritance and you can call desired overridden function in different class without virtual function and object pointer with low memory requirement. This method will improve efficiency of program with less complexity.

REFERENCES

[1]Sierra, Kathy; Bert Bates (2005). Head First Java, 2nd Ed.. O'Reilly Media, Inc.. ISBN 0596009208.

[2]Stroustrup, Bjarne (2000). The C++ Programming Language Special Edition. O'Reilly Media, Inc.. ISBN 0-201-70073-5.

[3] "Technical Report on C++ Performance", ISO/IEC TR 18015:2006(E)