Analysis of Early Binding Polymorphism in Object Oriented Programming

Cite as: AIP Conference Proceedings **1324**, 51 (2010); https://doi.org/10.1063/1.3526264
Published Online: 03 December 2010

Devendra Gahlot, S. S. Sarangdevot and Sanjay Tejasvee

Analysis of Early Binding Polymorphism in Object Oriented Programming

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

Department of Computer Application,Govt. Engineering College Bikaner,
Karni Nagar Industrial Area, Bikaner 334001(Raj)
devendragahlot@gmail.com,
Depertment, Janardan Rai Nagar Rajasthan Vidyapeeth University, Pratap Nagar, Udaipur (Raj)
dr.sarangdevot@gmail.com
Department of computer application,
Govt. Engineering College Bikaner,(Raj.)
sanjaytejasvee@gmail.com[3]

Abstract
Polymorphism means one thing exists in many forms of its own with each form having different implementation of its own. Polymorphism can be compile time or runtime. It is concern of lifetime of object. The lifetime can be determined at compile time or runtime. The compiler has a such knowledge to determine static analysis of source code.

I. INTRODUCTION

Object-oriented programming (OOP) is a programming paradigm that uses "objects" – data structures consisting of data fields and methods together with their interactions to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, modularity, polymorphism, and inheritance. It was not commonly used in mainstream software application development until the early 1990s.modern programming languages now support OOP. 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. 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).

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]

Compile time polymorphism includes:-
1.) Function Overloading
2.) Operator Overloading

Function Overloading:-
Function overloading depends on:-
1.) Variable Number of parameters to function
2.) Order of parameters to function
3.) Data type of parameters

Function overloading doesn't depend on:-
1.) Return type of function
2.) Name of parameters

We will consider various scenarios for function overloading in the next Code samples.
Let's first consider cases on which function overloading does not depend on:-

Short of the ideal, we believe there are ways to improve over existing compiler technology. For instance, static program analysis provides a compiler with the information needed t o distinguish between static and dynamic portions of a program. In particular, we have been investigating compile-time analysis techniques to determine the classes of each. object in an object-oriented program. This knowledge should permit the following compiler optimizations.

- Static binding of message sends to particular method implementations.
- Compile time type checking of some method parameters.
- In-line expansion of method bodies.

Operator overloading should only be utilized when the meaning of the overloaded operator's operation is unambiguous and practical for the underlying type and

CP1324, *International Conference on Methods and Models in Science and Technology (ICM2ST-10)*
edited by R. B. Patel and B. P. Singh

where it would offer a significant notational brevity over appropriately named function calls.
For our research we have invented that two object can be compaired if the conditional operator is being overloaded. The concern of this research is that a how to conditional operator can be overloaded.

II. STATIC, DYNAMIC BEHAVIOR AND SUBSTITUTION IN OBJECT ORIENTED LANGUAGE

Constant Pointers

The object can not be modified when use this pointer to access.

```
int n = 0;
const int * cp = &n;
* cp = 30;          // Error!
n = 30;            //OK!
//ex2constp.cpp
```

The same for parameters

```
size_t strlen(const char * str)
const char * aStr = "ABCDEFG";
char name [] = "Johnson";
unsigned n;
n = strlen(aStr);
n = strlen(name);
```

Functions returning Pointers to Constants

The variable receiving the returned value should be also a pointer to a constant.

```
class Student{
public:
  const char * GetName() const; // … }
```

Functions returning Pointers to Constants

```
Student s;
char * ncName = S.GetName();
const char * cName = S.GetName();
```

Reverse Polymorphism

- Polymorphism says we can assign a value from a child class to an instance of the parent class, but can this assignment then be reversed? Under what conditions?

Two aspects of reverse polymorphism

- There are two specific problems associated with the question of reverse polymorphism.
  - The problem of identity - can I tell if a value declared as an instance of a parent class actually holds a value from a subclass.
  - The task of assignment - can I then assign the value from the parent class to a variable declared as the subclass.
- In some languages mechanisms are provided to address these two problems together, while in other languages they are separated.

Minimum Static Space Allocation

The language C++ uses the minimum static space allocation approach. This is very efficient, but leads to some subtle difficulties.

What happens in the following assignment?

```
Window x;
TextWindow y;
x = y; //???
```

Assigning a Larger Value to a Smaller.

The Slicing Problem

The problem is you are trying to take a large box and squeeze it into a small space. Clearly this won't work. Thus, the extra fields are simply sliced off.

Question - Does this matter?
Answer - Only if somebody notices.
Solution - Design the language to make it difficult to notice (See Figure 1).

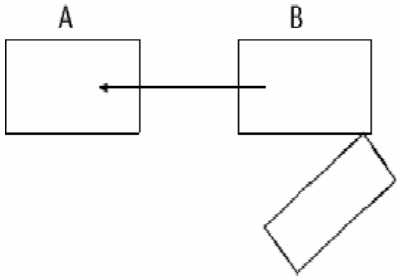

Figure 1. Rules for Member Function Binding in C++

- The rules for deciding what member function to execute are complicated because of the slicing problem.
- With variables that are declared normally, the binding of member function name to function body is based on the static type of the argument (regardless whether the function is declared virtual or not). With variables that are declared as references or pointers, the binding of the member function name to function body is based on the dynamic type if the function is declared as virtual, and the static type if not.

III. RELATED WORK AND OVERVIEW

Operator overloading (less commonly known as ad-hoc polymorphism) is a specific case of polymorphism (part of the OO nature of the language) in which some or all

operators like +, = or == are treated as polymorphic functions and as such have different behaviors depending on the types of its arguments. Operator overloading is usually only syntactic sugar. It can easily be emulated using function calls.

Consider this operation:
a + b × c
Using operator overloading permits a more concise way of writing it, like this:
operator_add (a, operator_multiply (b,c))
(Assuming the × operator has higher precedence than +.)

Operator overloading provides more than an aesthetic benefit since the language allows operators to be invoked implicitly in some circumstances. Problems and critics to the use of operator overloading arise because it allows programmers to give operators completely free functionality, without an imposition of coherency that permits to consistently satisfy user/reader expectations, usage of the << operator is an example of this problem.

// The expression
a << 1;

Will return twice the value of a if a is an integer variable, but if a is an output stream instead this will write "1" to it. Because operator overloading allows the programmer to change the usual semantics of an operator, it is usually considered good practice to use operator overloading with care.

Not all operators may be overloaded, new operators cannot be created, and the precedence, associativity or arity of operators cannot be changed (for example ! cannot be overloaded as a binary operator). Most operators may be overloaded as either a member function or non-member function, some, however, must be defined as member functions. Operators should only be overloaded where their use would be natural and unambiguous, and they should perform as expected. For example, overloading + to add two complex numbers is a good use, whereas overloading * to push an object onto a vector would not be considered good style.

Operators as member functions

Aside from the operators which must be members, operators may be overloaded as member or non-member functions. The choice of whether or not to overload as a member is up to the programmer. Operators are generally overloaded as members when they:

1. change the left-hand operand, or
2. require direct access to the non-public parts of an object.

When an operator is defined as a member, the number of explicit parameters is reduced by one, as the calling object is implicitly supplied as an operand. Thus, binary operators take one explicit parameter and unary operators none. In the case of binary operators, the left hand operand is the calling object, and no type coercion will be done upon it. This is in contrast to non-member operators, where the left hand operand may be coerced.

IV. CONCLUSION

In this paper we has analyzed that how to work static binding and dynamic binding in OOP's. Which is related to polymorphism. in analysis of object-oriented languages has produced some relatively imprecise analysis techniques. We have therefore developed better techniques. Furthermore, we provide a *means* for the compiler implementer to choose an appropriate trade-off between precision and the cost of the analysis. Separate compilation remains a challenge for any form of interprocedural static analysis. On one hand we would like to keep program units separate and minimize the - compilation effort. On the other hand we need to be able to analyze as much as possible of the source program to generate efficient code. To increase the efficiency of OOP's we should create intermediate code at the time being of conversion to machine code.

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] Haibin Zhu, Ph. D. Associate Professor of CS, Nipissing University

[4] Jan Vitek, R. Nigel Horspool and James S. Uhi Depatrment of Computer Science University of Victoria