The issue when OOP with C++

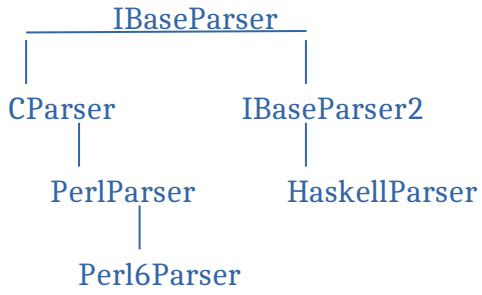
Interface design and runtime type cast

MaaJi Loeng
Sevenuc Consulting
http://sevenuc.com
July 12 2017

This document described the inherent issue and foundational thinking about OOP with C++ and explained why such type of OOP problem should be avoid in Java, Objective-C and Swift. This is the key points of programming language design, hope this issue as the kernel ability of a programming language that must be resolved, can inspire people thinking about the general issues about object construct and interface design of programming languages.

Object primer and the issue

As an example to demonstrate the issue, assume we do interface oriented programming with C++, here we declare an interface IBaseParser, and write a class CParser to implement it, and an inherited class PerlParser to extend the class. Now, we declare another interface IBaseParser2 to encapsulate extended public functions based on the interface IBaseParser, and write a class HaskellParser to implement it, and more inherited class to extend the implement. Along with the required features grows up, more and more functions added in the interface IBaseParser2. But now, we want to do one simple thing, write a class Perl6Parser that not only inherited from PerlParser, but also can utilise functions in interface IBaseParser2, how to implement it?




```
class IBaseParser                       #  abstract base class with pure virtual functions
class CParser:  public IBaseParser      #  implement interface IBaseParser
class CPerlParser:  public CParser      #  extend base class CParser
class CHaskellParser:   IBaseParser2    #  implement interface IBaseParser2
class Perl6Parser: public PerlParser,   +  IBaseParser2??
```

The C++ programming language itself doesn't have mechanism to directly resolve such problem. The same situation also exists in C++ multiple inheritance, when one class inherited from multiple base class, the object will hold multiple pointer point to multiple base class' instance, but when client call method of the object, ambiguous call occurred, this issue was generally called "*Dreaded Diamond" or "Diamond Problem*".

C doesn't support polymorphism like C++, no such issue, but type cast in C hasn't any limits (type checking), for this reason, many critical and potential bugs was caused due to "***void pointer casting***", but experienced programmer can prevent them by coding carefully.

Polymorphism required do type cast operations, C++ provided RTTI to resolve runtime type cast such as **static_cast**, **dynamic_cast**, and **reinterpret_cast**, and even **typeid** to fetch meta info of an object, but **reinterpret_cast** operation have big risk, and RTTI only works for classes that have virtual functions. Even OOP with C++ always feels very constrained, C++ still have many advantages as a senior OOP programming language, the earliest and most important one.

Java obviously know such OOP issue and designed **interface** keyword at its beginning of language design and using **instanceof** operation to detect the type of an object to resolve it. However, we only focus on the front part, i.e, interface design and ignore the rear part, type cast. And, of course, interface design for object and components is the foundational principle from the beginning of Java language design. Because of this reason, there're huge API designed from Sun and the successor Oracle, and even many of them never implemented.

As we see, the interface encapsulation (prototype definitions) developed from **struct** in C to **class** of C++, and **interface** of Java, and **protocol** in OC and Swift, the later languages have more flexibility than earlier languages. Of course, they also brings new side effect – big runtime. But they really implemented dynamic features that a programming language should have, and runtime complex objects management.

Resolve the issue with C++ itself

Why Abstract Base Class?

Because when there is a big class hierarchy from the base class, once the base class modified, all its child class must modify to correspond such changes, huge work will be required to deal with such situation. But if the base class is just an interface declare (abstract base class), only its concrete implement required to be modified in response to these changes. That's why when write big C++ software, you must write abstract base class first and then the implement class. That's why when write Java class, you write an interface first and then the implement class later.

COM (Component Object Model) extend the concept of interface to a new level abstraction. COM can be implemented by any programming language and don't allow inheritance, each component have its version to prevent ***"DLL Hell"***. Later, after OLE, COM, COM+ and DCOM, MS developed CLR to let COM components to run in a same virtual machine, but so-said great technology results to a huge and heavy runtime.

Infinity Loop!

Invented new technology to resolve existing problem, but brings new big troubles.

Resolve the issue: *Design a COM component to implement the features of CParser, and another component to implement the features of HaskellParser, separate their features, client can use the functions of the two component at runtime by query their exposed interfaces.*

Resolve the issue with Java

Java was an excellent programming language, but they made so many big bad software with this good tool. Java's language ability is better than C++, but have another inherent issue – low performance.

Back to the issue, we want to extend a class that implement an interface and at the same time aggregate another version of an extended interface of the interface. Let's see how Java can let the class Perl6Parser not only inherited from PerlParser, but also can optionally implement and/or utilise the interface IBaseParser2, please refer the ***Appendix A***.

Resolve the issue with Objective-C

Objective-C has very excellent dynamic language features than Java, and un-comparable high performance than Java and can seamlessly mix C and C++ code. Class in OC can optionally implement interface methods declared in its protocol list, and has *"Interface Inheritance"* mechanism like Java to extend base interface and override the implements of base class.

What we want is:

We want a class oriented from an interface and at the same time aggregate another version of an extended interface of the interface.

A **protocol** in OC can support inheriting from another **protocol** to extend it (*Interface Inheritance*), and a class can not only inherit a base class but also implement multiple interface **(protocol)** in the same class, such powerful feature gives people the maximum flexibility of programming. Let's see how the issue can be simply resolved in the ***Appendix B***.

Resolve the issue with Swift

As a new programming language of Apple, Swift combined many features of "Object Oriented Programming" and "Functional Programming" together, like OC, **protocol** in Swift can support inheriting from another **protocol** to extend it, but more than that, **protocol** is not simply used as an interface, but an extendable type like primary integer types, you can think it as an advanced version of C++ general template programming with dynamic types. Swift also supports class inheritance to extend and override existing functions' implements but with more strong type checking. The example code in ***Appendix C*** demonstrated how to resolve the issue gracefully.

Appendix A.

IBaseParser.java

```
package com.sevenuc.oop;
public interface IBaseParser {
   String name();
}
```

IBaseParser2.java

```
package com.sevenuc.oop;
import com.sevenuc.oop.IBaseParser.*;
public interface IBaseParser2 extends IBaseParser {
   String name2();
}
```

CParser.java

```
package com.sevenuc.oop;
import com.sevenuc.oop.IBaseParser.*;
public class CParser implements IBaseParser {
  public String name(){
    return "CParser";
  }
}
```

PerlParser.java

```
package com.sevenuc.oop;
import com.sevenuc.oop.CParser.*;
public class PerlParser extends CParser {
  public String name(){
    return "PerlParser";
  }
}
```

HaskellParser.java

```
package com.sevenuc.oop;
import com.sevenuc.oop.IBaseParser2.*;
public class HaskellParser implements IBaseParser2 {
  public String name(){
    return "HaskellParser name";
  }
  public String name2(){
    return "HaskellParser name2";
  }
}
```

Perl6Parser.java

```
package com.sevenuc.oop;
import com.sevenuc.oop.PerlParser.*;
import com.sevenuc.oop.HaskellParser.*;
public class Perl6Parser extends PerlParser {
  private HaskellParser haskell;
  Perl6Parser(){
    this.haskell = new HaskellParser();
  }
  public String name2(){
    if(haskell != null) return haskell.name2();
    return "Perl6Parser";
  }
}
```

build.sh

```
rm -rf  com/sevenuc/oop/*.class
javac com/sevenuc/oop/*.java
java com.sevenuc.oop.IssueDemo
```

Directory Structure:

```
build.sh
|___com
|___sevenuc
|    |___oop
|        |___IBaseParser.java
|        |___IBaseParser2.java
|        |___CParser.java
|        |___PerlParser.java
|        |___HaskellParser.java
|        |___Perl6Parser.java
|        |___IssueDemo.java
```

Appendix B.

```
//
//  main.m
//  OOP
//
//  Created by MaaJi Loeng on 12/7/2017.
//

#import <Foundation/Foundation.h>

@protocol IBaseParser <NSObject>
@required
- (NSString *)name;
@end

@protocol IBaseParser2 <IBaseParser>
@required
- (NSString *)name2;
@end

//===
@interface CParser: NSObject <IBaseParser>
@end
@implementation CParser
- (NSString *)name{
   return @"CParser";
}
@end

@interface PerlParser: CParser
@end
@implementation PerlParser
- (NSString *)name{
   return @"PerlParser";
}
@end
```

```
//===
@interface HaskellParser:NSObject <IBaseParser2>
@end
@implementation HaskellParser
- (NSString *)name{
   return @"HaskellParser name";
}
- (NSString *)name2{
   return @"HaskellParser name2";
}
@end

//===
@interface Perl6Parser: PerlParser <IBaseParser2>{
   HaskellParser *haskell;
}
@end

@implementation Perl6Parser
-(id)init{
   self = [super init];
   if(self){
      haskell = [[HaskellParser alloc] init];
   }
   return self;
}
- (NSString *)name{
   return [super name];
}
- (NSString *)name2{
   // fetch own's implement or HaskellParser's implement?
   if(haskell) return [haskell name2];
   else return @"Perl6Parser";
}
@end

int main(int argc, const char * argv[]) {
   @autoreleasepool {
      Perl6Parser *parser = [[Perl6Parser alloc] init];
      NSLog(@"1: %@", [parser name]);
      NSLog(@"2: %@", [parser name2]);
   }
   return 0;
}
```

Appendix C.

```
//  main.swift
//  swiftest
//  Created by MaaJi Loeng on 12/7/2017.
//
```

```
import Foundation

protocol IBaseParser {
   var name: String {get}
}

protocol IBaseParser2: IBaseParser {
   var name2: String {get}
}

//===
class CParser: IBaseParser {
   var name: String { return "CParser" }
}

class PerlParser: CParser{
   override var name: String { return "PerlParser" }
}

//===
class HaskellParser: IBaseParser2{
   var name: String { return "HaskellParser name1" }
   var name2: String { return "HaskellParser name2" }
}

//===
class Perl6Parser: PerlParser, IBaseParser2{
   var haskell: HaskellParser?
   override init() {
      self.haskell = HaskellParser()
   }
   var name2: String {
      if let id = haskell{
         return id.name2
      }else{
         return "Perl6Parser"
      }
   }
}

let parser = Perl6Parser()
print("1: \(parser.name)")
print("2: \(parser.name2)")
```

Download link:
http://www.sevenuc.com/download/the_issue.tar.gz (37KB)