Vrije
Universiteit
Brussel

Principles of Object-Oriented Programming Languages: Language Comparison

Mathijs Saey, 94451,
mathsaey@vub.ac.be,
1st Master of Science in Applied Sciences and Engineering:
Computer Science

January 13, 2013

Contents

1 Introduction

1.1 About

This paper is written for the Principles of Object Oriented Languages course at the VUB. The goal of this paper is to compare the object oriented properties of 2 languages.

1.2 Languages

Objective-C and C++ were chosen for this purpose, these languages are both strict supersets of C, but have a very different appraoch on object oriented features. C++ is focussed towards run-time efficieny and static type checking [2]; while Objective-C tries to do things dynamically whenever this is possible [1].

Both languages were compiled with llvm-gcc, with the following version info: "i686-apple-darwin11-llvm-gcc-4.2 (GCC) 4.2.1 (Based on Apple Inc. build 5658) (LLVM build 2336.11.00)". In the case of Objective-C, the *Foundation* runtime was used (received by appending -framework Foundation to the standard gcc compile instruction). Finally, all of the code files and their respective makefiles can be found at `https://github.com/mathsaey/POOL-Paper` in case of any issues.

1.3 Structure

At first, the general properties of both languages will be compared (such as access levels and the typing system). Afterwards, we will discuss the language features that are unique to each language and attempt to explain their absence in the other language.

2 General properties

2.1 Type system

Objective-C and C++ are both strict supersets of C, so it should come as no surprise to a reader familiar with C that both languages are statically typed. This is not entirely true for objective-C though. Objective-C introduces the *id* type, this type can be a pointer to any object. This effectively means that an Objective-C programmer can use dynamic and static typing next to each other.

This seemingly small difference already leads to some differences between our languages. This difference is mainly apparant when we need to create a function that can accept any type of object. An Objective-C programmer faced with such a problem simply uses a variable with the id type, the programmer can now send messages to the variable or store it internally without worrying about the type.

C++ introduces the concept of templates to deal with this. A programmer can create generic functions and classes using the template system. To use one of these functions we just provide the type of the variable to the class and the compiler will generate a version of that class that can handle the variable. This does mean that functions and classes that are built using templates can only accept the types that they were instantiated with. For instance, if we create a vector that contains strings in C++ then that vector will only be able to hold strings. An array in Objective-C on the other hand, can hold any variable with the id type, the type that can point to anything. Listings 1 and 2 show this array example in code.

2.2 Access modifiers

2.2.1 Member Variables

Objective-C and C++ both support *public*, *protected* and *private* member variables. Public variables are visible to anyone, while private variables are visible to the class (not the instance!) to which they belong. Protected variables are visible to the class to which they belong and any subclass of this class. It's important to note that an instance of class A can still access the private variables of another instance of A. This is possible in Objective-C and in C++.

It's also worth mentioning that Objective-C does allow you to access a protected variable outside of the class. The compiler generates a warning, but still allows you to do this. The following warning is shown: *"warning: instance variable b is @protected; this will be a hard error in the future"*. Furthermore, the reflective features of Objective-C still allow a programmer to access private instance variables. This is because access modifiers are checked at compile time and not at runtime. Reflection is mentioned in greater detail in section 3.1.4

2.2.2 Methods

Member variables have the same access modifier structure in C++ and Objective C, this is not the case for methods. C++ uses the same conventions for all members, methods and variables all use the same private, protected, public access modifiers. This is not the case in Objective-C.

In Objective-C any class can accept any message; the runtime will lookup the method in the class's *dispatch table*[1][Messaging], if the class has a method that matches, then the method will be executed, if it doesn't then an exception is raised. The only way to "hide" a method from the outside world in Objective-C is not including it in the class's interface. Doing this will let the compiler raise a warning when this method is called outside the class implementation. However, it's still possible for the outside world to trigger the method by sending the correct message.

2.3 Polymorphism

Objective-C and C++ both support polymorphism. Polymorphism allows different classes to react in different ways to the same message/method call. In C++ we can create a noisemaker class that has a *makeNoise* method. Another method, print, can accept any subclass of noisemaker and returns the value of makeNoise to the user (listing 5.). Thus polymorphism allows us to treat a given object (that is a subclass of noisemaker) in the same way while still receiving the class-specific behaviour.

We created the same example in Objective-C (listing 6). Note how both of our noisemaker objects don't need to share a parent to show the same behaviour? This is the id type allowing us to refer to any type, combined with the fact that an object in Objective-C can accept any message. In C++, calling an unknown message will result in a compile error. Giving 2 objects a common superclass is the only way to obtain polymorphism in C++.

In Objective-C we use a *protocol* (more on this in section 3.1.1) to indicate that car and sheep both implement the makeNoise method. It's worth noting at this point that this example would still work without the noisemaker protocol. The protocol merely allows the compiler to generate a warning if car or sheep do not implement the makenoise protocol.

2.4 Inheritance

Objective-C uses a relatively simple, single inheritance model. An object that receives a message for which it has no methods simply checks if it's superclass implements this method. Instance variables of the super class will keep their access level. An interesting property is that Objective-C doesn't allow overriding variables, adding a variable with name 'x' will raise a compiler error if a superclass already has a variable named 'x'.

This problem does not exist in C++, adding a variable that has already been defined raises no error. Accessing the variable of the superclass is simply achieved by prefixing the variable name with "*superClassName::*". The same prefix is used to access functions of a superclass. This method has to be used since C++ has no *super* keyword. This might seem strange at first, but this feature allows a C++ class to inherit from multiple classes. This feature is called "*explicit qualification*". Using explicit qualification solves any possible ambiguities when inheriting from multiple parents.

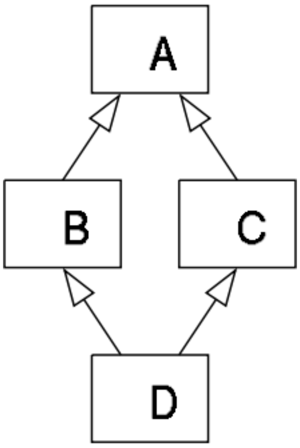

Figure 1: The diamond problem illustrated (source: Wikipedia)

A common multiple inheritance problem is the *diamond problem.* This problem occurs when a class D inherits from classes B and C that have a common parent A. If D calls a method defined in A that has different behaviour in B and C then which version of the method will be called? C++ solves this problem by keeping a sperate instance of A for both B and C, explicit qualification ensures that the correct version of the method is called. If B and C inherit A *virtually*, then the compiler ensures that only one A object is constructed.

A C++ class does not inherit access rights in the same way that Objective-C does. The access right of the inherited class has to be explicitly mentioned. This access right then transfers to the members of that superclass. If the inherited class is declared public, then the access rights remain the same. If the inherited class is declared protected, then every public member of the superclass is protected in the subclass. If the inherited class is declared private, then every member of the superclass is private in the subclass.

2.5 Constructors and Destructors

C++ constructors are normal methods with no return type and the class name as their signature. Method overloading (see section 3.2.1) allows the definition of multiple constructors. Destructors in C++ are declared in a similar way, however, a destructor cannot accept any arguments and is named "~*Classname()*". A destructor will be automatically called right before deleting an object.

The Objective-C language has no mandatory constructors or destructors. Constructing and deconstructing an object generally happens by overriding a few methods inherited from NSObject. These methods are listed here:

- **alloc** Alloc allocates the space that the object needs.
- **init** Init inialises the necessary instance variables.
- **dealloc** Dealloc releases all the instance variables.

Note that nothing prevents an Objective-C programmer from creating his own alloc, init or dealloc that don't derive from NSObject directly or indirectly. For instance, a programmer that wants to implement the singleton pattern generally overrides alloc to ensure that an object is only created when he wants it. Initialisers generally derive from NSObject's init at some point (via a call to super init). Conventions dictate that initialisers start with init. Multiple initialisers are simply constructed by creating different methods, note that - unlike in C++, where a constructor has no return type or statement - an initisialiser should return itself after instantiating the necessary instance variables.

3 Lanaguage-Specific Features

3.1 Objective-C

3.1.1 Protocols

3.1.2 Categories

3.1.3 Message Forwarding

3.1.4 Reflection

3.1.5 Metaclasses

3.2 C++

3.2.1 Method Overloading

3.2.2 Namespaces

3.2.3 Abstract Classes

3.2.4 Friend Classes

3.2.5 Inner Classes

4 Conclusion

5 References

[1] Apple. *Objective-C Runtime Programming Guide*. URL: http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008048.

[2] *C++: A brief description*. URL: http://www.cplusplus.com/info/description/.

6 Code Listings

Listing 1: Templates and static typing in C++

```
#include <vector>
#include <string>

using namespace std;

void arrayExample() {
  vector<string> vec;
  vec.push_back("Hello␣World");
}

int main(int argc, char const *argv[]) {
  arrayExample();
  return 0;
}
```

Listing 2: Objective-C using Dynamic and Static typing

```
#import <Foundation/Foundation.h>

void arrayExample() {
  NSMutableArray *arr = [NSMutableArray
      arrayWithCapacity:2];
  [arr addObject:[NSDate date]];
  [arr addObject:@"Hello␣world!"];
}

int main(int argc, char const *argv[]) {
  arrayExample();
  return 0;
}
```

Listing 3: Access rights in C++

```
#include <iostream>
using namespace std;

class A {
public:
  A() {
    a = 1;
    b = 2;
    c = 3;
  }
  void aVisibleMethod() {cout << "You␣can
      ␣see␣me!␣\n";}
  int getC(A obj) {return obj.c;}
private:
  void anInvisibleMethod() {cout << "You␣
      can't␣see␣me!␣\n";}
public:
  int a;
protected:
  int b;
private:
  int c;
};
```

```

class B : protected A {
  void getA() {printf("%d",a);}
  void getB() {printf("%d",b);}
  //void getC() {printf("%d",c);} => Access.cpp:27: error:   int A::c   is private
};

int main() {
 A t1, t2;
  t1.a = 1;
  //t1.b = 2; => Access.cpp:25: error:   int A::b   is protected
  //t1.c = 3; => Access.cpp:27: error:   int A::c   is private
  t1.getC(t2);

  t1.aVisibleMethod();
  //t1.anInvisibleMethod(); => Access.cpp:18: error:   void A::anInvisibleMethod()   is private
  return 0;
}
```

Listing 4: Access rights in Objective-C

```
#import <Foundation/Foundation.h>

@interface A : NSObject {
@public
  int a;
@protected
  int b;
@private
  int c;
}
- (id) init;
- (void) show;
- (int) getC: (A*) obj;
- (void) aVisibileMethod;
@end

@implementation A
- (id) init {
  if (self = [super init]) {
    a = 1;
    b = 2;
    c = 3;
  }
  return self;
}

- (void) show {
  printf("%d,", a);
  printf("%d,", b);
  printf("%d\n",c);
}

- (int) getC: (A*) obj {return obj->c;}
- (void) aVisibileMethod {printf("You can see me!\n");}
- (void) anInvisibleMethod {printf("You can't see me!\n");}

@end

@interface B: A {}
- (void) getA;
- (void) getB;
//- (void) getC;
@end

@implementation B
- (void) getA {printf("%d", a);}
- (void) getB {printf("%d", b);}
//- (void) getC {printf("%d", c);}  => Access.m:33: error: instance variable   c   is declared private
@end

int main() {
  A *t1 = [[A alloc] init];
  A *t2 = [[A alloc] init];

  [t1 getC:t2];

  t1->a = 9;
  t1->b = 8;
  //t->c = 10; => Access.m:45: warning: instance variable   b   is @protected; this will be a hard error in the future

  id iVar;
  object_getInstanceVariable(t1, "c", &iVar);
  printf("c is: %d\n", iVar);

  [t1 show];
  [t1 aVisibileMethod];
  [t1 anInvisibleMethod];

  return 0;
}
```

Listing 5: Polymorphism in C++

```
#include <iostream>
#include <string>
using namespace std;

class NoiseMaker {
public:
  virtual string makeNoise() =0;
};

class Sheep : public NoiseMaker {
public:
  string makeNoise() {return "Beeeeiii\n";}
};

class Car : public NoiseMaker {
```

```
public:
  string makeNoise() {return "Vrooom\n";}
};

void print(NoiseMaker* m) {
  cout << m->makeNoise();
}

int main(int argc, char const *argv[]) {
  Car* c = new Car();
  Sheep* s = new Sheep();

  print(c);
  print(s);

  return 0;
}
```

Listing 6: Polymorphism in Objective-C

```
#import <Foundation/Foundation.h>

@protocol noiseProtocol
- (NSString*) makeNoise;
@end

@interface Sheep : NSObject <noiseProtocol>
@end

@implementation Sheep
- (NSString*) makeNoise {return @"Beeeeiii\n";}
@end

@interface Car : NSObject <noiseProtocol>
@end

@implementation Car
- (NSString*) makeNoise {return @"Vrooom\n";}
@end

print(id<noiseProtocol> noiseMaker) {
  printf([[noiseMaker makeNoise] UTF8String]);
}

int main(int argc, char const *argv[]) {
  print([Sheep alloc]);
  print([Car alloc]);
  return 0;
}
```

Listing 7: Inheritance in C++

```
#include <iostream>

using namespace std;

class A {
public:
  A():a(1),b(2),c(3) {}

  int a;
protected:
  int b;
private:
  int c;
};

class B : public A {
public:
  B(): a(5) {}
  int a;
  //void testFunc(){c = 5;} => Inheritance.cpp:14: error: int A::c is private
};

class D : protected A {};
class C : private A {};

int main(int argc, char const *argv[])
{
  B b;
  cout << b.a << "\n";
  cout << b.A::a << "\n";

  D d;
  //d.a; => Inheritance.cpp:9: error: int A::a is inaccessible

  return 0;
}
```

Listing 8: Multiple inheritance in C++

```
#include <iostream>

using namespace std;

class Animal {
public: virtual void talk() =0;
};

class Horse: public Animal {
public: void talk() {cout << "Hello␣I'm␣a␣horse\n";}
};

class Donkey: public Animal {
public: void talk() {cout << "Hello␣I'm␣a␣donkey\n";}
};

class Mule: public Horse, public Donkey {};

int main(int argc, char const *argv[])
{
  Mule m;
  //m.talk();
  //m.Animal::talk();
  m.Horse::talk();
  m.Donkey::talk();
  return 0;
}
```

Listing 9: Inheritance in Objective-C

```
#import <Foundation/Foundation.h>

@interface A: NSObject {
@public
  int a;
@protected
  int b;
@private
  int c;
}
@end

@implementation A
- (id) init {
  if (self = [super init]) {
    a = 1;
    b = 2;
    c = 3;
  }
  return self;
}
@end

@interface B: A {
  //int a; => Inheritance.m:25: error: duplicate member a
}
- (void) checkLevels;
@end

@implementation B
- (void) checkLevels {
  NSLog(@”%d,%d”,a,b);
  //Inheritance.m:30: error: instance variable c is declared private
}
@end

int main(int argc, char const *argv[]) {
  B *b = [[B alloc] init];
  [b checkLevels];
  b->a = 3;
  b->b = 2;
  return 0;
}

// Variabelen houden access level van superklasse over
```

Listing 10: Constructors in C++

```
class aClass {
public:
  aClass(): _a(1) {}
  aClass(int a): _a(a) {}
  ~aClass() {_a = 0;}
private:
  int _a;
};

int main(int argc, char const *argv[]) {
  aClass a1; //calls the default constructor
  aClass a2(3);
  return 0;
}
```

Listing 11: Constructors in Objective-C

```
#import <Foundation/Foundation.h>

@interface aClass: NSObject {
  int a;
}
- (id) initWithA: (int) aValue;
@end

@implementation aClass

- (id) init {
  if (self = [super init]) {
    a = 1;
  }
  return self;
}

- (id) initWithA: (int) aValue {
  if (self = [self init]) {
    a = aValue;
  }
  return self;
}

- (void) dealloc {
  [super dealloc];
  a = 0;
}

@end

int main(int argc, char const *argv[])
{
  aClass *a1 = [[aClass alloc] init];
  aClass *a2 = [[aClass alloc] initWithA:5];
  return 0;
}
```