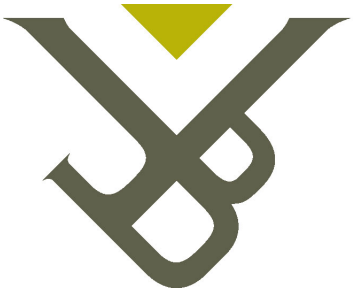

Vrije
Universiteit
Brussel

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 14, 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 approach on object oriented features. C++ is focussed towards run-time efficiency and static type checking [3]; 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 apparent 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.

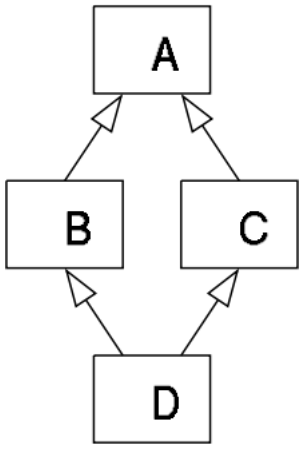

Figure 1: The diamond problem illustrated (source: Wikipedia)

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 separate 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 initialises 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 initialiser names 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 initialiser should return itself after instantiating the necessary instance variables.

3 Language-Specific - Features

3.1 Objective-C

3.1.1 Protocols

An Objective-C protocol defines a standard for objects to message each other. A protocol is simply a list of required and optional methods. A class 'A' that implements a protocol agrees to implement every required method of the protocol. A method or other class now knows that A will respond to the messages declared in the protocol. Reflection also allows other classes to see if a given class implements a certain protocol. An example of the use of protocols can be found in listing 12. Protocols allow us to work with objects without knowing the class of these objects, this allows a higher level of polymorphism, since we can ensure that the object we are working with will respond to a set of messages.

C++ has no protocol or interface keyword that allows us to declare a set of methods. It is possible to create interfaces in C++ by creating a class that only contains *pure virtual* functions. A class that derives from this interface class can only be instantiated if it implements every pure virtual function that the interface declares. Section 3.2.3 expands on virtual functions in greater detail.

3.1.2 Categories

Objective-C categories were based on smalltalk categories and were introduced to make it easier to break classes down into smaller pieces. A category adds a set of methods to a class, these methods are added to the class at runtime. A category that extends a class can access all of the instance variables of this class, it's important to remember that a category is a class extension rather then an actual subclass. A category also has the ability to override existing methods of the class it extends.

Categories are generally used to either split the interface of a class over multiple files, or to extend pre-existing classes at runtime. Indeed, a category allows us to extend any class, regardless of where it's interface is located. This allows us to expand native and user-defined class on the fly.

C++ has no category system, and does not allow this amount of runtime flexibility. A C++ class cannot be changed at runtime. It's also impossible to split a class declaration over multiple header files. It is possible

to split the class implementation.

3.1.3 Message Forwarding

Objective-C works with explicit message sending to objects. When an object receives a message, it looks up the method in the dispatch table of this object's class, If this is message is not recognised, then the message is looked up inside the superclass. An exception is thrown if the message is not recognised by the class highest in the hierarchy (NSObject).

Objective-C allows a programmer to intercept this process before the exception is thrown. The object receives the message and has the ability to do as it pleases. This allows the programmer to use the message and to pas it along to another class. This feature is known as message forwarding. An example of message forwarding is presented in listing 15. In this example an instance of A sends messages it does not understand to an instance of the B class if B can respond to this message. When we send the message doSomething to an instance of A, it will forward it to an instance of B, which can respond to this message. Message forwarding is often used when creating proxy objects.

Message forwarding can be used to mimic multiple inheritance in Objective-C (we can say that A inherits the doSomething method from B), it is important to realise that message forwarding is not a replacement for multiple inheritance though. Multiple inheritance adds features from different classes to a single class, while message forwarding divides responsibilities between different classes.

C++ does not offer message forwarding in the way the Objective-C does. It is possible to explicitly forward certain messages, but it is not possible to have a generic method that forwards every message it does not understand to another class. This is due to the fact that the C++ compiler returns errors when we call messages on an object that does not define them. In fact, sending a message to a C++ object is almost completely transformed into a method call by the compiler. This is another example of C++ favouring run-time efficiency over run-time flexibility.

3.1.4 Reflection

Another example of the run-time flexibility of Objective-C is reflection. Reflection is the ability to examine and modify the behaviour of an object at runtime.

A listing of some of the reflective features of Objective-C is mentioned here. Keep in mind that this is only a listing of some basic features, the full list can be found at [2].

- Query an object about it's instance variables, type, superclass, protocols it implements or method list.
- Get and Set instance variables
- Add methods, replace methods

An example of adding and replacing methods is added in listing 16.

C++ has no reflection, by now this should come as no surprise, reflection happens purely at run-time and does not really fit in the run-time efficiency philosophy of C++.

3.1.5 Metaclasses

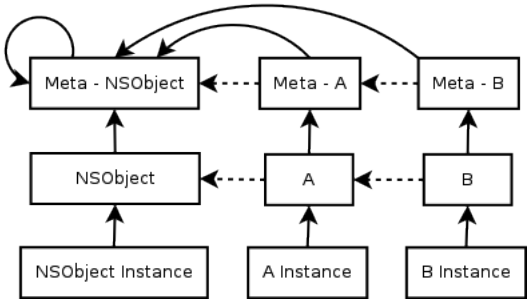

Figure 2: Inheritance and instance relations in Objective-C. A dashed line indicates an inheritance relation, while a normal line indicates an instance of relation.

A lot of the inspiration for Objective-C came from smalltalk; in smalltalk, everything is an object, even classes are objects. The class of a class object is called the meta class. Meta classes also exist in Objective-C.

Objective-C classes have 2 different types of methods, instance methods are methods that work on a single instance, and class methods, methods that work on the class instance (which is an instance of the meta-class). These class methods don't require an initialised instance of class, and can be compared to static class methods in other languages. Class methods are simply instance methods of the meta-class. An example of a class method is alloc.

Metaclasses are also instances of a class, a metaclass is an instance of the NSObject metaclass. This means that every object has a parent class that is also an object, except for the metaclass of NSObject, the metaclass of NSObject is simply an instance of itself, this is the only class that is allowed to do this.

Metaclasses also have their own inheritance hierarchy, if class A inherits from class B, then metaclass A inherits from metaclass B. An illustration of all this can be found in figure 2.

C++ has no metaclasses.

3.2 C++

3.2.1 Method Overloading

C++ allows the use of method overloading. This means that it is possible to create different methods with the same name, as long as their arity or argument types differ. Method overloading also works on constructors in C++.

Objective-C does not allow method overloading. This is generally not an issue due to the smalltalk-style messages that Objective-C uses (e.g. doSomethingWith A: (int) a andB: (int) b) which automatically differ when the arity of the method differs. The lack of method overloading is only an issue in Objective-C when we want to declare 2 functions with the same name, the same arity, but different variable types. We can use the id types as the type of the variable in combination with a double dispatch or a type-check as a substitute if having the same name is absolutely required.

3.2.2 Namespaces

Namespaces allow C++ programmers to divide their classes and other code into separate blocks. Data in a namespace is impossible to access without either prefixing it with *namespacename::* or by adding "*using namespace namespacename*" to the code. An example of namespaces is presented in listing 18.

This mechanism is especially useful when working on large code-bases, or when 3rd party libraries are used.

Objective-C has no similar mechanism for code separation. Convention dictates that classnames should have a 2 letter prefix that indicates the company or the product that this class belongs to. Examples of this are the NSString, NSArray, UIButton and various other classes. However, this convention only makes the problem less frequent rather then solving it.

3.2.3 Abstract Classes

An abstract class in C++ is a class that contains at least one pure virtual function. An abstract class cannot be instantiated on it's own. A class that derives from an abstract class without implementing every pure virtual function is also an abstract class. Abstract classes can be used for a number of things. One example that we presented earlier was the use of abstract classes to implement protocols (section 3.1.1). Abstract classes are generally used to force a subclass to implement a certain method which can then be called in a polymorphic way.

Objective-C has no virtual keyword, since every method is virtual by default. Due to this, Objective-C has no abstract classes. We can however replace the pure virtual method of our C++ example by a method that simply throws an exception when it's called. This will raise an error at runtime if the "virtual" method is called by the abstract class.

3.2.4 Friend Classes

The friend class system allows a class to gain extra access rights to another class. If A declares B as it's friend, then B will receive access to all the methods and variables in A. It's important to note that this relation is not bidirectional, A will not gain access to the private variables that are part of B.

Objective-C doesn't have a friend system. We could replicate one by creating a category on A that provides all the extra functions that B needs. We can simply import this category into every class that needs friend access to A. This makeshift solution has it's drawbacks, any class can import this category, in the C++ system the friend is declared inside the class that gives access. In our solution, any class can just import the category and gain access into A, even if this is not intended. It's also possible to use reflection to access the private and protected members of A. Once more it is shown that access modifiers are checked at compile time and not at runtime.

3.2.5 Inner Classes

Inner classes are classes that are defined inside another class. The inner class can access the private, public and protected members of the outer class in C++. The outer class uses the standard rules when accessing the inner class.

Inner classes do not exist in Objective-C, it is possible to reach a similar effect by creating the inner class inside the implementation file of the outer class. This is just another class, but it's unreachable for other classes since it's never been declared inside any header file. The inner class will remain private to the implementation of the outer class, as long as the file that contains the interface of the inner class is never imported into another file.

4 Conclusion

Objective-C and C++ are both powerful object-oriented languages with a very different approach on adding objects to C. While C++ attempts to provide a language that is both close to the hardware and high level, with a focus on run-time efficiency and type safety at compile type; Objective-C tries to build a lightweight, smalltalk style object-oriented system on top of C that offers as much run-time flexibility as possible.

It's quite clear that both languages attract a different audience, even though they share a common ancestor. Both languages have their advantages that can drastically improve a project depending on the needs of the programmer.

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] Apple. *Objective-C Runtime Reference*. URL: `http://developer.apple.com/library/ios/#documentation/Cocoa/Reference/ObjCRuntimeRef/Reference/reference.html`.

[3] *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]];
```

```
6   [arr addObject:@"Hello world!"];
7 }
8
9 int main(int argc, char const *argv[]) {
10  arrayExample();
11  return 0;
12 }
```

Listing 3: Access rights in C++

```
1 #include <iostream>
2 using namespace std;
3
4 class A {
5 public:
6   A() {
7     a = 1;
8     b = 2;
9     c = 3;
10  }
11  void aVisibleMethod() {cout << "You can see me! \n";}
12  int getC(A obj) {return obj.c;}
13 private:
14  void anInvisibleMethod() {cout << "You can't see me! \n";}
15 public:
16  int a;
17 protected:
18  int b;
19 private:
20  int c;
21 };
22
23 class B : protected A {
24  void getA() {printf("%d",a);}
25  void getB() {printf("%d",b);}
26  //void getC() {printf("%d",c);} => Access.cpp:27: error:   int A::c    is private
27 };
28
29 int main() {
30  A t1, t2;
31  t1.a = 1;
32  //t1.b = 2; => Access.cpp:25: error: int A::b    is protected
33  //t1.c = 3; => Access.cpp:27: error: int A::c    is private
34  t1.getC(t2);
35
36  t1.aVisibleMethod();
37  //t1.anInvisibleMethod(); => Access.cpp:18: error:   void A::anInvisibleMethod()    is private
38  return 0;
39 }
```

Listing 4: Access rights in Objective-C

```
1 #import <Foundation/Foundation.h>
2
3 @interface A : NSObject {
4 @public
5   int a;
6 @protected
7   int b;
8 @private
9   int c;
10 }
11 - (id) init;
12 - (void) show;
13 - (int) getC: (A*) obj;
14 - (void) aVisibileMethod;
15 @end
16
17 @implementation A
18 - (id) init {
19   if (self = [super init]) {
20     a = 1;
21     b = 2;
22     c = 3;
23   }
24   return self;
25 }
26
27 - (void) show {
28   printf("%d,", a);
29   printf("%d,", b);
30   printf("%d\n",c);
31 }
32
33 - (int) getC: (A*) obj {return obj->c;}
34 - (void) aVisibileMethod {printf("You can see me!\n");}
35 - (void) anInvisibleMethod {printf("You can't see me!\n");}
36
37 @end
38
39 @interface B: A {}
40 - (void) getA;
41 - (void) getB;
42 //- (void) getC;
43 @end
44
45 @implementation B
46 - (void) getA {printf("%d", a);}
47 - (void) getB {printf("%d", b);}
48 //- (void) getC {printf("%d", c);}  => Access.m:33: error: instance variable c    is declared private
49 @end
50
51 int main() {
52   A *t1 = [[A alloc] init];
53   A *t2 = [[A alloc] init];
54
55   [t1 getC:t2];
56
57   t1->a = 9;
58   t1->b = 8;
59   //t->c = 10; => Access.m:45: warning:
```

```
        instance variable    b    is
        @protected; this will be a hard
        error in the future
60
61  id iVar;
62  object_getInstanceVariable(t1, "c", &
        iVar);
63  printf("c is: %d\n", iVar);
64
65  [t1 show];
66  [t1 aVisibileMethod];
67  [t1 anInvisibleMethod];
68
69  return 0;
70}
```

Listing 5: Polymorphism in C++

```
1#include <iostream>
2#include <string>
3using namespace std;
4
5class NoiseMaker {
6public:
7  virtual string makeNoise() =0;
8};
9
10class Sheep : public NoiseMaker {
11public:
12  string makeNoise() {return "Beeeeiii\n"
        ;}
13};
14
15class Car : public NoiseMaker {
16public:
17  string makeNoise() {return "Vrooom\n";}
18};
19
20void print(NoiseMaker* m) {
21  cout << m->makeNoise();
22}
23
24int main(int argc, char const *argv[]) {
25  Car* c = new Car();
26  Sheep* s = new Sheep();
27
28  print(c);
29  print(s);
30
31  return 0;
32}
```

Listing 6: Polymorphism in Objective-C

```
1#import <Foundation/Foundation.h>
2
3@protocol noiseProtocol
4- (NSString*) makeNoise;
5@end
6
7@interface Sheep : NSObject <
      noiseProtocol>
8@end
9
10@implementation Sheep
11- (NSString*) makeNoise {return @"
      Beeeeiii\n";}
12@end
13
14@interface Car : NSObject <noiseProtocol>
15@end
16
17@implementation Car
18- (NSString*) makeNoise {return @"Vrooom\
      n";}
19@end
20
21print(id<noiseProtocol> noiseMaker) {
22  printf([[noiseMaker makeNoise]
        UTF8String]);
23}
24
25int main(int argc, char const *argv[]) {
26  print([Sheep alloc]);
27  print([Car alloc]);
28  return 0;
29}
```

Listing 7: Inheritance in C++

```
1#include <iostream>
2
3using namespace std;
4
5class A {
6public:
7  A():a(1),b(2),c(3) {}
8
9  int a;
10protected:
11  int b;
12private:
13  int c;
14};
15
16class B : public A {
17public:
18  B(): a(5) {}
19  int a;
20  //void testFunc(){c = 5;} =>
        Inheritance.cpp:14: error:   int A
        ::c    is private
21};
22
23class D : protected A {};
24class C : private A {};
25
26int main(int argc, char const *argv[])
27{
28  B b;
29  cout << b.a << "\n";
```

```
30   cout << b.A::a << "\n";
31
32   D d;
33   //d.a; => Inheritance.cpp:9: error:
           int A::a    is inaccessible
34
35   return 0;
36 }
```

Listing 8: Multiple inheritance in C++

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Animal {
6 public: virtual void talk() =0;
7 };
8
9 class Horse: public Animal {
10 public: void talk() {cout << "Hello I'm a
       horse\n";}
11 };
12
13 class Donkey: public Animal {
14 public: void talk() {cout << "Hello I'm a
       donkey\n";}
15 };
16
17 class Mule: public Horse, public Donkey
       {};
18
19 int main(int argc, char const *argv[])
20 {
21   Mule m;
22   //m.talk();
23   //m.Animal::talk();
24   m.Horse::talk();
25   m.Donkey::talk();
26   return 0;
27 }
```

Listing 9: Inheritance in Objective-C

```
1 #import <Foundation/Foundation.h>
2
3 @interface A: NSObject {
4 @public
5   int a;
6 @protected
7   int b;
8 @private
9   int c;
10 }
11 @end
12
13 @implementation A
14 - (id) init {
15   if (self = [super init]) {
16     a = 1;
17     b = 2;
18     c = 3;
19   }
20   return self;
21 }
22 @end
23
24 @interface B: A {
25   //int a; => Inheritance.m:25: error:
         duplicate member    a
26 }
27 - (void) checkLevels;
28 @end
29
30 @implementation B
31 - (void) checkLevels {
32   NSLog(@"%d,%d",a,b);
33   //Inheritance.m:30: error: instance
         variable    c    is declared
         private
34 }
35 @end
36
37 int main(int argc, char const *argv[]) {
38   B *b = [[B alloc] init];
39   [b checkLevels];
40   b->a = 3;
41   b->b = 2;
42   return 0;
43 }
```

Listing 10: (De)Constructors in C++

```
1 class aClass {
2 public:
3   aClass(): _a(1) {}
4   aClass(int a): _a(a) {}
5   ~aClass() {_a = 0;}
6 private:
7   int _a;
8 };
9
10 int main(int argc, char const *argv[]) {
11   aClass a1; //calls the default
         constructor
12   aClass a2(3);
13   return 0;
14 }
```

Listing 11: (De)Constructors in Objective-C

```
1 #import <Foundation/Foundation.h>
2
3 @interface aClass: NSObject {
4   int a;
5 }
6 - (id) initWithA: (int) aValue;
7 @end
8
9 @implementation aClass
```

```
10
11- (id) init {
12   if (self = [super init]) {
13     a = 1;
14   }
15   return self;
16 }
17
18- (id) initWithA: (int) aValue {
19   if (self = [self init]) {
20     a = aValue;
21   }
22   return self;
23 }
24
25- (void) dealloc {
26   [super dealloc];
27   a = 0;
28 }
29
30 @end
31
32 int main(int argc, char const *argv[])
33 {
34   aClass *a1 = [[aClass alloc] init];
35   aClass *a2 = [[aClass alloc] initWithA:
         5];
36   return 0;
37 }
```

Listing 12: A protocol in Objective-C

```
1 #import <Foundation/Foundation.h>
2
3 @protocol Protocol
4 @required
5 - (void) doSomething;
6 @optional
7 - (void) doNothing;
8 @end
9
10 @interface Delegate : NSObject <Protocol>
       {}
11 @end
12
13 @implementation Delegate
14 - (void) doSomething {printf("Hello world
      !\n");}
15 @end
16
17 int main(int argc, char const *argv[]) {
18   id<Protocol> d = [Delegate alloc];
19
20   [d doSomething];
21   //[d doNothing];
22   return 0;
23 }
```

Listing 13: A category in Objective-C

```
1 #import <Foundation/Foundation.h>
2
3 @interface A : NSObject
4 - (void) doSomething;
5 @end
6
7 @implementation A
8 - (void) doSomething {printf("Nothing \n"
      );}
9 @end
10
11 @interface A (doSomething)
12 - (void) doSomething;
13 @end
14
15 @implementation A (doSomething)
16 - (void) doSomething {printf("Something \
      n");}
17 @end
18
19 int main(int argc, char const *argv[]) {
20   A *obj = [[A alloc] init];
21   [obj doSomething];
22   return 0;
23 }
```

Listing 14: Message forwarding in C++

```
1 #include <iostream>
2
3 using namespace std;
4
5 template <typename T>
6 class A {
7 public:
8   A(T obj): _obj(obj) {}
9
10   void messageForT() {_obj.messageForT()
        ;}
11
12 private:
13   T _obj;
14 };
15
16 class B {
17 public:
18   void messageForT() { cout << "Something
        \n";}
19 };
20
21 int main(int argc, char const *argv[])
22 {
23   B b;
24   A<B> a = A<B>(b);
25   a.messageForT();
26   return 0;
27 }
```

Listing 15: Message forwarding in Objective-C

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

```
2
3@interface B: NSObject {}
4- (void) messageForB;
5@end
6
7@implementation B
8- (void) messageForB {printf("Something \
    n");}
9@end
10
11
12@interface A : NSObject {
13  B *b;
14}
15@end
16
17@implementation A
18
19- (id) init {
20  if (self = [super init]) {
21    b = [[B alloc] init];
22  }
23  return self;
24}
25
26- (void)forwardInvocation:(NSInvocation
    *) anInvocation {
27    if ([b respondsToSelector: [
        anInvocation selector]])
28        [anInvocation invokeWithTarget:b
            ];
29    else [super forwardInvocation:
        anInvocation];
30}
31
32- (NSMethodSignature *)
    methodSignatureForSelector:(SEL)
    aSelector {
33    return [B
        instanceMethodSignatureForSelector
        :aSelector];
34}
35
36@end
37
38
39int main(int argc, char const *argv[]) {
40  A *a = [[A alloc] init];
41  [a messageForB];
42  return 0;
43}
```

Listing 16: Changing and adding methods at runtime in Objective-C

```
1#import <Foundation/Foundation.h>
2#import <objc/runtime.h>
3
4@interface A : NSObject {}
5@end
6
7@implementation A
8- (void) doSomething {printf("Hello world
    !\n");}
9@end
10
11void newMethod(id _self,SEL _cmd) {
12  printf("It's really nice to add new
      methods at runtime! \n");
13}
14
15void somethingReplacer(id _self, SEL _cmd
    ) {
16  printf("I'm doing something else! \n");
17}
18
19int main(int argc, char const *argv[]) {
20  A *a = [[A alloc] init];
21  [a doSomething];
22
23  // Adding a method
24  class_addMethod([A class], @selector(
      addedMethod), (IMP) newMethod, "v@:
      ");
25  [a addedMethod];
26
27  //Replacing a method
28  Method methodToModify =
      class_getInstanceMethod([A class],
      @selector(doSomething));
29  method_setImplementation(methodToModify
      , (IMP) somethingReplacer);
30  [a doSomething];
31
32  return 0;
33}
```

Listing 17: Method overloading in C++

```
1#include <iostream>
2#include <string>
3
4using namespace std;
5
6void doSomethingWithA(int a) {
7  printf("%d\n", a);
8}
9
10void doSomethingWithA(const char* a) {
11  printf("%s\n", a);
12}
13
14int main() {
15  doSomethingWithA(1);
16  doSomethingWithA("Haha");
17  return 0;
18}
```

Listing 18: Namespaces in C++

```
1#include <iostream>
2using namespace std;
```

```
3
4 namespace A {
5   void doSomething() {printf("Nothing\n"
        );}
6 }
7 namespace B {
8   void doSomething() {printf("Something\
        n");}
9 }
10
11 int main(int argc, char const *argv[]) {
12   A::doSomething();
13   B::doSomething();
14   return 0;
15 }
```

Listing 19: Abstract class in C++

```
1 #include <iostream>
2 using namespace std;
3
4 class AbstractClass {
5 public:
6   virtual void doSomething()=0;
7 };
8
9 class DerivedClass : private
      AbstractClass {
10 public:
11   virtual void doSomething() {
12     cout << "Hello world!\n";
13   }
14 };
15
16 class StandAloneClass {
17 public:
18   virtual void doSomething() {
19     cout << "Hello world!\n";
20   }
21 };
22
23 int main(int argc, char const *argv[]) {
24   DerivedClass a;
25   a.doSomething();
26   return 0;
27 }
```

Listing 20: Replicating abstract classes in Objective-C

```
1 #import <Foundation/Foundation.h>
2
3 ////////////////////
4 // Abstract Class //
5 ////////////////////
6
7 @interface AbstractClass : NSObject {}
8 - (void) doSomething;
9 @end
10
11 @implementation AbstractClass
12 - (void) doSomething {
13   [self doesNotRecognizeSelector:_cmd];
14 }
15 @end
16
17 ////////////////////
18 // Derived Class //
19 ////////////////////
20
21 @interface DerivedClass : AbstractClass
      {}
22 - (void) doSomething;
23 @end
24
25 @implementation DerivedClass
26 - (void) doSomething {
27   printf("Hello world!\n");
28 }
29 @end
30
31 ////////////////////////
32 // StandAlone Class //
33 ////////////////////////
34
35 @interface StandAloneClass : NSObject {}
36 - (void) doSomething;
37 @end
38
39 @implementation StandAloneClass
40 - (void) doSomething {
41   printf("Hello world!\n");
42 }
43 @end
44
45 int main(int argc, char const *argv[]) {
46   AbstractClass *abstract = [
        AbstractClass alloc];
47   DerivedClass *derived = [DerivedClass
        alloc];
48
49   [derived doSomething];
50   [abstract doSomething];
51
52   return 0;
53 }
```

Listing 21: Friend classes in C++

```
1 #include <iostream>
2
3 class A {
4   friend class B;
5 public:
6   A():a(1),b(2) {};
7
8 private:
9   int a;
10  int b;
11 };
12 class B {
```

```
13 public:
14   B(A a):_a(a.a), _b(a.b) {}
15   int _a;
16   int _b;
17 };
18
19 int main(int argc, char const *argv[])
20 {
21   A a;
22   B b = B(a);
23   b._a;
24   b._b;
25   return 0;
26 }
```

Listing 22: Inner classes in C++

```
1 #include <iostream>
2
3 using namespace std;
4
5 class A {
6 public:
7   void doSomething() {
8     B b;
9     b.innerMethod();
10  }
11
12 private:
13  class B {
14  public:
15    void innerMethod() {cout << "Hello␣
          world!\n";}
16  };
17 };
18
19 int main(int argc, char const *argv[])
20 {
21   A a;
22   a.doSomething();
23   return 0;
24 }
```