Comparison of two object-oriented languages: C++ and Ruby

Jeroen Heymans

January 3, 2012

Contents

1 Introduction

1.1 Foreword

This paper is written for the master course Principles of Object Oriented Languages on the Vrije Universiteit Brussel. It contains a comparison of the object-oriented features from two languages that contain multiple object-oriented features.

1.2 The compared languages

The choice for this paper was C++, a well-known static typed language with good object-oriented features, and Ruby, a dynamic typed language that was designed to be as object-oriented as possible.

1.2.1 C++

Originally called "C with classes", C++ is a multi-paradigm, static typed language based on the popular language C. C++ does not require you to write all code in an object-oriented fashion though, it is also possible to write all your code in a procedural way via functions. As a multi-paradigm language, it supports however many object-oriented features. We have tested all the C++ code in this paper with the GCC compiler on Ubuntu (version 4.5.2).

1.2.2 Ruby

When designing Ruby, the creator of Ruby, Ykihiro Matsumoto, looked at other languages to find an ideal syntax. He wanted a dynamic typed scripting language that was more powerful than Perl and more object-oriented than Python. In Ruby, everything is an object. It was influenced by Smalltalk, every type was given method instance variables. All Ruby code was tested with Ruby version 1.9.2p0 (2010-08-18 revision 29036).

1.3 What follows next

We will compare several important features that belong to object-oriented languages like inheritance, polymorphism, access control and many more.

2 Comparisons

2.1 Dynamic versus static typing

C++ is a static typed language while Ruby is a dynamic typed language. Ruby supports duck typing which is a credited to James Whitcomb Riley's words:

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

The type of a variable or object is determined at runtime.

Because of the difference in typing, there are some differences in Ruby and C++. For example, in C++ we can have multiple functions or methods with the same number of parameters but with other types. *void foo(int)* and *void foo(float)* are both different functions. In Ruby, this is not possible, only one function or method is allowed with the same name.

Due to the dynamic typed nature of Ruby, there is no need for constructions like templates in C++ to enforce one function or method to accept multiple types.

2.2 Classes

In object-oriented programming, a class is a blueprint that is used to construct objects which we call object instances. We compare the several types of classes that can be made in Ruby and C++.

2.2.1 Abstract classes

An abstract class is a class that can not be instantiated for it is either labelled as abstract or it specifies abstract methods. The general idea of abstract classes is that the class must be extended. In C++, an abstract class is a class having at least one pure virtual function. For example, this is an abstract class in C++:

```
class A {
  public:
    virtual void f() = 0;
};
```

If we are trying to compile C++ code where instantiate this abstract class, for example like this:

```
int main() {
   A a = new A();
   return 0;
}
```

We will receive an error message from the compiler that notifies us that this operation is not allowed:

```
g++ a.cpp -o a
a.cpp: In function 'int_main()':
a.cpp:10:16: error: cannot allocate an
    object of abstract type 'A'
a.cpp:3:9: note: because the following
    virtual functions are pure within 'A':
a.cpp:5:16: note: virtual void A::f()
a.cpp:10:6: error: cannot declare variable '
    a' to be of abstract type 'A'
a.cpp:3:9: note: since type 'A' has pure
    virtual functions
make: *** [a] Error 1
```

Abstract classes in Ruby are a different story. Unlike some object-oriented languages, Ruby does not support a keyword like *abstract* or a way like C++ to define an abstract class. Actually, Ruby does not provide the necessary tools to create abstract classes. However, the programmer himself can force a class to behave like an abstract class when he makes *:new* (the constructor) private. The extending class must then make *:new* public:

```
class A
  private_class_method :new
  def initialize(txt = "Hello!")
    puts txt
  end
end

class B < A
  public_class_method :new
end
```

As long as there is no class that extends A, the methods in A will never be accesible as it is not possible to construct an instance of A.

2.2.2 Final classes

A final class is a class that can not be extended. This could be done for reasons of security and efficiency. Final classes are not supported in C++ nor Ruby however there exist mechanisms to prevent programmers from extending other classes. An example of such a mechanism in C++:

```
class Final;

class MakeFinal {
  private:
      MakeFinal() { }
    friend class Final;
};

```

```
class Final : virtual MakeFinal {
  public:
    Final() { }
};

class Derived : public Final { };
```

In this example, we made the class named Final final. We did this by inheriting class Final from the class MakeFinal. In class MakeFinal, we have a private constructor which can only be called by the class Final as the class Final is marked as *friend*. By inheriting MakeFinal *virtual* in the class Final, we make sure that the constructor *MakeFinal()* must be called if we derive the class Final. As only Final is a friend, this will generate a compile error that notifies us that *MakeFinal()* is private.

In Ruby, we can create a similar mechanism:

```
class Class
   def final(klass)
      class << klass
         def inherited(subclass)
            raise RuntimeError, "Illegal attempt to subclass #{self} with #{subclass}"
         end
      end
   end
end

class Foo
   final(self)
end

class Bar < Foo
end
```

In this example, we add a method *final* to the standardclass *Class*. In this method *final*, we define that a new method called *inherited* should be added to the class given by the parameter *klass*. *inherited* is a method that will always be called whenever we try to inherit the class that contains this method. In our example, this will cause a *RuntimeError*, thus disallowing the programmer to continue programming as long as we try to inherit our *final* class.

But is a class really final in Ruby if we force a class to be final like we did in C++? Unfortunately no. This is due to the *open classes* concept of Ruby that will always allow you to add methods to a class, whether it is marked as *final* like in our example or not.[1]

[1]See the section "Partial classes" however for an example on how to prevent extending an open class

2.2.3 Friend classes

C++ has an unique keyword: *friend*. With this, one can declare a class as a friend:

```
class A;
class B {
  friend class A;
}
```

In this example, class A is a friend of B. This means that class A has more access rights to the members and methods of class B. More concrete, this means that all private and protected members and methods of B are accessible by A while this would not be the case by default. The *friend* keyword defines the following features:

- Friendships are not corresponded: if class A is a friend of class B, class B is not automatically a friend of class A.
- Friendships are not transitive: if class A is a friend of class B and class B is a friend of class C, then is class A not a friend of class C.
- Friendships are not inherited: if there is a class Base with a friend, then the class Derived (that inherits from Base) does not have that same friend. The same goes up for all friends of Derived, they are not automatically friends of Base. The same goes in the other direction: if Base (or Derived) is a friend of a class, then is Derived (or Base) not automatically a friend of the same class.
- The restricted members and methods from Base that are inherited by Derived are accessible by a friend class of Derived if Derived itself can access these members and methods. So if Derived inherits publicly[2] from Base, Derived only has access to the protected and public members from Base, not the private members, so neither does a friend (the same goes for methods).

  Friend classes are not possible in standard Ruby, however there exist some libraries[3] that allow a programmer to define friend classes in Ruby.

[2]See the section "Derived classes" in "Access control" for more details on deriving classes.

[3]An example can be found on https://github.com/lsegal/friend

2.2.4 Inner classes

Inner classes are called nested classes in Ruby. They can simply be defined like you would expect:

```ruby
class A
  attr_accessor :a
  def initialize
    @a = "a"
  end
  def f
    puts @a
  end
  class B
    attr_accessor :b
    def initialize
      @b = "b"
    end
    def g
      puts @a
    end
  end
end
```

We can then construct an object of the type *A* and *A::B*:

```ruby
b = A::B.new
a = A.new
```

With the *::* operator in *A::B*, we simply say: "Take class B that is defined inside the scope of A". The object of class B can be treated like it was never defined in A. Both instances of the classes can exist completely on their own: if you make an instance of A, this will not indirectly create an instance of B and vice-versa. This means that an instance of B will not have access to the methods and members of A, only to the methods and members of its own instance. The same goes for an instance of A. Only if there is an explicit construction of an object B or A in respectively A or B, they will be able to access the methods and members.

With inner classes in C++, almost the same story exists. In order to access data from the inner class via the outer class or from the outer class via the inner class, an instance must be present to access this data. This can be achieved when constructing the inner or outer class:

```cpp
#include <iostream>

class A {
  private:
    int a;
  public:
    A(int newA): a(newA), b(this) { }
    class B {
      public:
        B(A* a): outer(a), test(1) { }
        int f() {
          return outer->a;
        }
        A* outer;
      public:
        int test;
    } b;
    int f() {
      return b.test;
    }
};

int main()
{
  A a = A(3);
  std::cout << a.b.f() << std::endl;
  std::cout << a.f() << std::endl;
  return 0;
}
```

Let us explain the code. What we have here are two classes: A as an outer class and B as an inner class. Upon construction of A, we automatically call the constructor of B with an instance of A. With this, B has a pointer that points to the object of A. A itself remembers the pointer to B that has been constructed. With this, it is possible to access A from within B and B from within A. Similar code can be written for Ruby where we also construct an instance of class A in class B and vice-versa.

However, there are some rules concerning the access to and from the inner class in C++. The code above will compile and work. But if you look closely, the method *f()* from the class B can access member *a* from class A eventhough *a* is defined as private. So in fact, the inner class gets the privilege to act as if it is part of the outer class.

We must notice though that there is a different story when you want to access the inner class. In the code above, A has a method *f()* that access the member *test* that is defined in B, via the pointer to B that was saved in A. This is because the member *test* is defined as public. If we however define *test* as protected or private, the compiler will notify us that this is not allowed.

We can therefore conclude that an inner class in C++ gets the privilege to access everything from its outer class, even the private and protected members or methods. This is in contrast to how the outer class can access the inner class, in this case the standard access rules are applied. In Ruby, there are no special access rights for inner and outer classes, the standard rules are applied as enforced by the access specifiers *public*, *protected* and *private*.

2.2.5 Metaclasses

For every defined class in Ruby, there exists a metaclass. For example if we define a class Person, Ruby automatically creates a metaclass MetaPerson[4]. The role of the metaclass MetaPerson is to split behaviour between Class (to create new classes) and class specific behaviour.

Ruby has two different types of methods: class methods and instance methods. Class methods are not designed to work on a specific instance of the class while instance methods are specifically designed to work on an instance of the class. Class methods are often compared to what we call static methods in other languages while instance methods can be compared to normal methods in other languages. The class methods are saved in the metaclass of the class. The class methods of Person are in fact instance methods of MetaPerson.

The same goes up for class variables and instance variables. Class variables are comparable to static variables in other languages: they are not instance-specific while instance variables are instance-specific. The class variables of Person are in fact instance variables of MetaPerson.

C++ does not have built-in support for metaclasses. However, it is possible to mimic the behaviour of meta-classes. This would mean however for example that you manage a metaclass yourself where the metaclass is a singleton class.

2.2.6 Partial classes

A partial class is a class whose definition may be splitted into multiple pieces. These pieces can be within one source file or across multiple files. Ruby supports the notion of partial classes. With this, it is possible to extend existing classes with new methods or members. The class must not be user-defined, it can be for example the built-in *String* class:

```
class String
  def doublePrint
    print self
    print self
  end
end
puts "test".doublePrint
```

This will give as a result: ”*testtest*”.

[4]Internally a class for X will not be called MetaX but we want to make it more readable.

Partial classes like in Ruby do not exist in C++. It is not possible to add new methods or members on-the-fly. The reason for this is that Ruby classes are considered ”open” while C++ classes are ”closed”. When you define a C++ class in the typical header-file, there is no possibility to split this definition in several files, e.g. multiple header files for the same class are not allowed. Only the definition of the methods themselves can be split over mutiple source-files. It is for example possible to have a header-file ”a.h” and multiple source-files like ”method1-from-a.cpp” and ”method2-from-a.cpp”. Still, this does not create pure open classes like in Ruby where you can add methods at runtime, it is merely a way to make smaller files.

If you do not want that classes are further extended in Ruby, then the possibility of *freezing* the class exists. By simple adding the keyword *freeze* to the class, you declare the class as frozen (e.g. not modifiable).

2.3 Access control

2.3.1 Access control on members

C++ offers three keywords that define access control on members and methods of classes: *public*, *protected* or *private*.

- *public*: the member/method is accessible without any restriction
- *protected*: only the class that defines the member/method, the friend classes of the class and the subclasses of the class can access the member/method
- *private*: only the class that defines the member/method and the friend classes of the class can access the member/method

Ruby also provides these three keywords but there are some subtle differences. We will now explain one difference, the rest will be explained in the following section. The difference between Ruby and C++ can be found in the objects on which a private method is called. This C++ code compiles:

```
class A {
  private:
    void foo() { return; }
  public:
    void test(A* a) {
      foo();
      this->foo();
      a->foo();
    }
};

int main()
{
  A* a = new A();
  A* b = new A();
  a->test(b);
  return 0;
}
```

As you can see, we have a class A in which the *test()*-method accepts a parameter to an A object. All three calls to the method *foo()* work: with an implicit receiver and an explicit receiver like *this* or *a*. If we try the Ruby equivalent of this code, we get into trouble:

```
1 class A
2   private
3     def foo
4     end
5   public
6     def test(obj)
7       foo
8       self.foo
9       obj.foo
10    end
11 end
12
13 a = A.new
14 b = A.new
15 a.test(b)
```

Both the lines *self.foo* and *obj.foo* generate the same error *NoMethodError*. This is because we use an explicit receiver. The implicit receiver in line 7 allows the call to *foo*. In Ruby, it depends on the type of receiver that you use while this is not important in C++.

2.3.2 Derived classes

When declaring a derived class in C++, we can provide an access specifier: *public*, *protected* or *private*. For example if we have the base class A, we can create subclass B from A and provide one of the three access specifiers like this:

```
class A { };
class B: public A { };
```

The purpose of this access specifier is to alter the access control of the members of the base class via the derived class. The access control is not altered on the original base class. An example:

```
1 #include <iostream>
2
3 class A {
4   public:
5     int foo;
6     A(): foo(0) {};
7 };
8
9 class B : public A { };
10
11 int main()
12 {
13   B* b = new B();
14   b->foo = 3;
15   std::cout << "Value foo: " << b->foo <<
        std::endl;
16   return 0;
17 }
```

When compiling and excecuting this code, the program will output correctly:

```
Value foo: 3
```

If we however change the line access keyword when defining the class B to for example *protected* or *private* (instead of the current *public*), we get a compilation error eventhough everything in A was originally defined as *public*:

```
g++ a.cpp -o a
a.cpp: In function 'int main()':
a.cpp:5:7: error: 'int A::foo' is
    inaccessible
a.cpp:14:7: error: within this context
a.cpp:5:7: error: 'int A::foo' is
    inaccessible
a.cpp:15:40: error: within this context
make: *** [a] Error 1
```

When deriving a class in C++, the following situations can occur:

- Derive as *public*: public and protected members of the base class remain public and protected members of the derived class.
- Derive as *protected*: public and protected members of the base class become protected members of the derived class.
- Derive as *private*: public and protected members of the base class become private members of the derived class.

In all cases, private members of the base class remain private, e.g.: not even the derived class can access them unless there is a specific friend declaration within the base class that explicitly grants access to them. If we do not provide an access specifier, the access rules of the baseclass are simply copied, e.g.: public remains public, protected remains protected and private remains private.

Unlike C++, Ruby does not support the changing of the access properties of members and methods of the superclass via one keyword (when we derive a class). However, Ruby has the strange property to allow the call to private methods and members from the superclass via the subclass. An example:

```
class Animal
  private
  def speak
    puts "Random noise"
  end
end

class Cat < Animal
  public
  def speak
    super
  end
end
```

When we excecute the following code:

```
cat = Cat.new
cat.speak
```

This will print *Random noise*. The explanation behind this is that Ruby actually allows to call private methods and members from a class via an implicit receiver. If you use an explicit receiver (for example *Animal.speak*) the call will fail like you would expect. So although Ruby does not have an access specifier when deriving a class, it is however possible to mimic the behaviour from C++, even with more detail as you can control the access specifiers for all methods separatly. Only thing you have to do is rewrite the methods where you want some other access control and call the original method.

2.4 Inheritance

In Ruby, a class can only inherit from a single other class. Some other languages like C++ support multiple inheritance, a feature that allows classes to inherit features from multiple classes.

```
class Animal
  def breath
    puts "Breathe"
  end
  def speak
    puts "Random noise"
  end
end

class Cat < Animal
  def speak
    puts "Meow"
  end
end
```

When using these two classes, one can see that methods will be overriden:

```
cat = Cat.new
cat.speak
cat.breath
```

This will output:

```
Random noise
Breathe
```

C++ supports multiple inheritance. With multiple inheritance it is possible to inherit from multiple classes like this:

```
class C: public A, protected B { };
```

In this example, a class C is created that inherits publicly from class A and protectedly from class B. As you can see, the access modifiers can be different.

One of the common pitfalls when using multiple inheritance in C++ is ambiguities:

```
class A { virtual void f(); };
class B { virtual void f(); };
class C : public A ,public B { void f(); };
```

This issue can be solved by using explicit qualification. We explicitly tell the compiler which *f()* we would like to call:

```
C* c = new C;
c->f();
c->A::f(); // call f() from class A
c->B::f(); // call f() from class B
```

Because of multiple inheritance, C++ does not support the keyword *super* which makes it possible to call methods and retrieve members from the superclass. If you inherit from class A and class B, to which class should *super* point? To avoid confusion and get clear code, one always has to use explicit qualification when calling the superclass.

Another more complex problem which can arise with multiple inheritance is called the *diamond problem*. The diamond problem is an ambiguity that arises when two classes B and C inherit from A, and class D inherits from both B and C. If a method in D calls a method that is defined in A (and D does not override the method) and B and C have overridden that method differently, then from which class does it inherit: B or C? This is a problem that can be solved by using virtual inheritance.

When constructing an object of class D, C++ actually creates in the memory objects of B and C internally. B and C in their turn also construct each an object of A. This means that upon the creation of D, 4 internal objects are created: B, C and 2 times A. If we however have inherited B and C from A as virtual (for example: *class B: virtual public A*), C++ will make sure that only one object of A is constructed.

Although Ruby does not support multiple inheritance, it eliminates the need for multiple inheritance by providing mixins. As a programmer, you can define modules in Ruby. In classes, you can then include these modules. For example:

```
module Foo
  def Bar
    puts "Wazaa!"
  end
end

class FooBar
  include Foo
end
```

In this example, the class FooBar contains the content of module Foo (in this case the method Bar). It is possible to include multiple modules, even if the modules contain the same methods:

```
module A
  def foo
    puts "Module A"
  end
end

module B
  def foo
    puts "Module B"
  end
end

class C
  include A
  include B
end

c = C.new
c.foo
```

This will output *Module B*. The order in which a programmer includes modules, depends on what methods will be available. If we would have included A after B, the output would be *Module A*. As modules are included, they override functionality that was defined in the lines before the include. Methods that are defined after the include statement, will override methods that were included.

2.5 Polymorphism

The literal meaning of polymorphism is "the ability to take on multiple forms or shapes". This refers, in a broader sense, to the ability of different objects to respond in different ways to the same message or method invocation. We have written an example in Ruby that shows what polymorphism can do:

```
class Animal
  def makeNoise
    throw NotImplementedError.new("makeNoise
          () not implemented")
  end
end

class Dog < Animal
  def makeNoise
    puts "Woof!"
  end
end

class Cat < Animal
  def makeNoise
    puts "Meow!"
  end
end

[Dog, Cat, Animal].each do |obj|
  a = obj.new
  a.makeNoise
end
```

In this example, we see that the classes Dog and Cat are subclasses from Animal. We dynamically create a Dog and Cat object and call the method *makeNoise*. The output is correctly:

```
Woof!
Meow!
```

Even though Cat and Dog are different classes, they implement the same messages with a different implementation but can be called in exactly the same way. Via the implementation of *makeNoise* in the class Animal, we make sure that there will be a clean error when someone has subclassed Animal but not overwritten *makeNoise*; thus ensuring that all the subclasses of Animal understand the same message *makeNoise* but with their own implementation. An equivalent example in C++ code:

```
#include <iostream>

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

class Dog: public Animal {
  public:
    void makeNoise() {
      std::cout << "Woof!" << std::endl;
    }
};

class Cat: public Animal {
  public:
    void makeNoise() {
      std::cout << "Meow!" << std::endl;
    }
};

int main()
{
   Animal* a = new Dog();
   a->makeNoise();
   Animal* b = new Cat();
   b->makeNoise();
   return 0;
}
```

This correctly outputs:

```
Woof!
Meow!
```

As you can see, we treat the objects Dog and Cat like they were Animal objects. We call the same method *makeNoise()* which results in different results, e.g.: the same message results in a different execution.

2.6 Namespaces/modules

Namespaces in C++ are way to group classes, functions and other identifiers in one block. Outside this block, one must prefix the identifiers he wants to access with the namespace specifier. To ommit the prefixing, one can use the following statement:

```
using namespace abc;
```

With this line, it is no longer necessary to prefix with *abc::*. Namespaces are hierarchical, there can be mutiple namespaces defined in another namespace. When requesting an identifier, the compiler start searching from the namespace where the identifier is requested. For example if we request C in A::B, the compiler will first search if there exists a C in A::B. If not, the compiler searches in A and after that in the global namespace.

Modules are the Ruby equivalent of namespaces. They can also be defined hierarchical. The difference with C++ namespaces is that modules can be included in classes. This allows programmers to add new methods to classes. If several classes implement exact the same method, this method can be placed in a module to ensure that every class has the same code. This concept is called mixins which is an alternative in Ruby to multiple inheritance in C++.

2.7 Reflection

In standard C++, reflection is not possible. The popular framework Qt has however implemented reflection features[5].

Reflection is possible in Ruby. Via several standard methods, we get lots of possiblities in Ruby to perform reflection-related operations:

- Instance variables: setting, getting and removing
- Class variables: setting, getting and removing
- Methods: define, undefine, alias

Define, undefine and alias of methods can be done on instance methods and class methods. To perform these operations on instance methods, we must work on an instance of a class. If we want to perform these operations on class methods, we must work on the metaclass.

2.8 Interfaces

Both C++ and Ruby do not support a keyword like *interface*. It is however possible to enforce interfaces in these languages. In C++ this is done very easily: create an abstract class that only consists of virtual functions without a body. This enforces inherited classes to implement all the methods. If this is not done, the inherited classes will be abstract classes.

Via several modules, one can enforce interfaces in Ruby[6]. Generally, it is the same story as with abstract classes but you enforce the programmer to override all the methods of a class.

2.9 Everything is an object?

One of the main features of Ruby is the fact that everything is an object, even primitive types like an integer. Ruby follows the influence of the Smalltalk language by giving methods and instance variables to all of its types. An example of this feature:

```
5.times { print "test".length }
```

This will generate the output: *44444*. What it does is take an object *5* of the type Number, call the method *times* on it so it excecutes five times the part between { and }. The code *print "test".length* will be excecuted five times which is effectively printing the length of the String object "test".

In C++, not everything acts as an object. The primitive data types: *char*, *short*, *int*, *long*, *bool*, *float*, *double*, *long double* and *whar_t* ; do not have

[5] A class that is important to have reflection in Qt is for example QMetaObject: http://doc.qt.nokia.com/5.0-snapshot/qmetaobject.html

[6] An example of an implementation of interfaces can be found on http://www.metabates.com/2011/02/07/building-interfaces-and-abstract-classes-in-ruby/

a class that provides methods to call on the data types. The data type *string* is an exception to this as it does have it's own class that can be used when including the library *string*.

3 Conclusion

Ruby was designed to be as object-oriented as possible. Everything had to be an object. Eventhough this is correctly implemented, there are quite some differences with the object-oriented features that are provided by C++. The abscence of certain keywords do not allow features like for example friend classes. Though, most of these features that are not standard available in Ruby, can be implemented by clever use of modules and method implementations.

Does the abscence of certain features make Ruby less object-oriented than C++? Not at all, there is just a difference between the audiences of both languages. C++ is more low-level than Ruby which is why concepts as "everything is an object" and reflection are most of the time obsolete in C++. This is because these concepts make the code possibly more memory-consuming or the program more dynamic than one needs.

Both languages are equipped with a good basis for object-oriented programs, they simply have other implementations basic of the difference in audiences.

4 References

http://www.ruby-lang.org/en/about/
http://www.cplusplus.com/info/history/
http://publib.boulder.ibm.com/infocenter/lnxpcomp/v8v101/index.jsp?topic=%2Fcom.ibm.xlcpp8l.doc%2Flanguage%2
http://www.klankboomklang.com/2007/10/05/the-metaclass/ http://ruby-extra.rubyforge.org/classes/Object.html
http://www.wikyblog.com/AmanKing/Metaclass$_i n_R$*ubyhttp* : *//www.klankboomklang.com/*2007/10/12/*objects − classes − and − jruby − internals/http : //shiningthrough.co.uk/A − comparison − of − Ruby − pass − by − reference − and − C + + − pass − by − referencehttp : //rubylearning.com/satishtalim/ruby*$_i$*nheritance.htmlhttp : //publib.boulder.ibm.com/infocenter/lnxpcomp/v*8*v*101*/index.jsp?topic =* %2*Fcom.ibm.xlcpp*8*l.doc*%2*Flanguage*%2*Fref*%2*Fcplr*130*.htmhttp : //www.skorks.com/*2010/04/*ruby − access − control − are − private − and − protected − methods − only − a − guideline/http : //weblog.jamisbuck.org/*2007/2/23/*method − visibility − in − rubyhttp : //stackoverflow.com/questions/*137661/*how − do − you − do − polymorphism − in − rubyhttp : //www.cplusplus.com/doc/tutorial/variables/http : //www.brpreiss.com/books/opus*8*/html/page*597*.htmlhttp : //www.java2s.com/Code/Ruby/Class/classvariablevsobjectvaria*
*//cplusplus.co.il/*2009/09/01/*final − frozen − classes − in − cpp/http : //www.geeksforgeeks.org/archives/*16222*http : //www.metabates.com/*2011/02/07/*building − interfaces − and − abstract − classes − in − ruby/*