作业 1: 程序设计语言的 OO 特性调研报告

姓名：昝妍
学号：1601214481
日期：2017/03/05

内容:

面向对象已经为很多程序设计语言所支持, 但不同语言的面向对象特征和关注重点并不尽相同。请调研至少 2 种你之前了解甚少的语言, 了解其设计思想和面向对象特征, 并和一门你比较熟悉的语言如 Java, C++, C# 做比较, 给出代码示例, 提交一份调研报告。

调研范围包含但不限于:

Objective-C, Go, Python, Ruby, Scala, Swift, Erlang, PHP, Smalltalk, Java, C++, C#

作业内容：
本报告调研三种语言，分别是 C++，Ruby 与 Python。

一、 语言的设计思想

A. Ruby

Ruby 于 1995 年面世，是一种快捷易用面相对象程式设计的直译式脚本语言。发行者松本行弘 Matz 混合 Perl,Smalltalk,Eiffel,Ada 及 Lisp 与自己的想法生产新语言，更以 Perl 的诞生石的命名方法为此新语言命名为 Ruby。其核心思想为”从编写程式的人出发，像人体一样外表简单内裡複杂，自然而不简单的语言。”，Matz 曾说过”想要一个比 Perl 更强大,比 Python 更物件导向的脚本语言。”

B. Python

Python 是于 1991 年发布的面相对象直译式程式语言。语法简单，与其它大多数程式设计语言使用大括弧不一样，而是使用缩进来定义语句块，其设计哲学为”优雅，明确，简单”，「用一种方法，最好是只有一种方法来做一件事」为开发者哲学。Python 经常被当作脚本语言用于处理系统管理任务和 web 程式编写，也常用在 GUI 与 OS 等大型系统开发。

C. C++

C++为 1985 年正式发布的通用程式设计语言，支援面向对象设计与程序化程式设计等多种编程范型。C++从 C 和 Simula 中取得灵感而来，得名于 C 语言中的「++」运算子，而且在共同的命名约定中，使用「+」以表示增强的程式。C++适用于各种用途，并且兼具快速和可移植性。

二、 面向对象特徵

A. Ruby

Ruby 是动态型别(Dynamically typed)的程式语言，执行期环境(runtime)会尽可能的处理所有工作(例如:不需要事先知道你的 Ruby 程式将会被连结(link)到哪一个模组，或是哪一个方法(method)将会被呼叫)。在 Ruby 中万物即物件: 所有的资料型态都可赋予方法与产生实体变数并适用于所有物件。Ruby 也包含类别、方法、迭代器(iterators)及封包(closures)等面相对象的设计。而在继承的部分，Ruby 只提供单继承，但同时模组(module)提供混入(mixin)功能，造成子类别可能不小心就覆写覆类别的功能，但 Ruby 表示这种方式比使用複杂多限制的多重继承来得清楚。

B. Python

Python 是完全物件导向的语言，函式、模组、数字、字串都是物件。并且完全支援继承、重载、衍生、多重继承，有益于增强原始码的複用性。

C. C++

C++ 为类别构成式物件导向程式设计语言，物件的概念和 C 的对应概念接近。在语法中明确地使用类别来做到资料抽象化、封装、模组化、继承、子型别多型、物件状态的自动初始化。C++中，一个类别即为一个型别，加上封装，一个类别即为一个抽象资料型别（Abstract Data Type，ADT），继承、多型、模板都加强了类别的可抽象性。在 C++可以使用 class 或 struct 这两个关键字宣告类别（class），而使用 new 运算子实体化类别产生的实体（instance）即为物件。C++以资料成员（data member）表达属性，以成员函式（member function）表达行为。

三、 比较与代码实例

语言间的相同处：

- Ruby 与 C++可以用程序式的方式写程式(但底层依然是物件导向的环境。)
- Ruby 与 C++都可以使用__FILE__ 与 __LINE__。
- Ruby 与 C++都可以定义常数。Ruby 没有特殊的 const 关键字，利用命名的惯例来强迫变数为常数：第一个字母为大写的变数便为常数。
- Ruby 的程式码放进模组(module)之中，类似 C++ 中的 namespace 的作法。
- Python 与 Ruby 所有东西都是物件，变数只是指向某个物件的参考指标。

语言间的不同处：

- C++能明确的定义参照(reference)，Ruby 中每个变数都会被自动解参照回原本的物件。

- Ruby 与 Python 的物件型别是强型别也是动态型别(dynamically typed)。Ruby 执行期环境将会在执行期的方法呼叫成功时，自行辨识型别。
- 物件的建构子 C++称为类别名称，Ruby 使用 initialize 命名。
- C++能直接存取成员变数，Ruby 所有对公开的成员变数(属性 attribute)的存取都透过方法呼叫。
- Ruby 可以在任何时候重新打开一个类别以加入新的方法。
- 迭代(Iteration)的运作方式不同，Ruby 中使用物件 mixin Enumerator 模组并且直接呼叫 my_obj.each 方法，C++使用一个独立的迭代器(Iterator)。
- Ruby 只有两种容器类别：Array 跟 Hash。
- C++中有型别转换，而 Ruby 没有。
- Ruby 的标准函式库中就包含了单元测试(Unit test)函式库。
- Python 分”新式”或”旧式”的类别写法，Ruby 的类别就只有一种写法。
- Ruby 不能直接存取物件的属性，只能透过方法呼叫。
- Ruby 与 C++有 public、private、以及 protected 三种方式来设定存取层级，Python 裡是用变数名称前后加底线的方式表示。
- Ruby 用“混入（mixing）”功能用来取代 C++与 Python 的多重继承。

	C++	Ruby	Python

類別與物件	class class_name {
	permission_label_1: member1;
	permission_label_2: member2; ... }
	object_name;
	
	物件
	
	(int a;)
		建立一個狗的類別
		ruby> class Dog
		| def speak
		| puts "Bow Wow"
		| end
		| end
		nil
		
		物件
		
		ruby> pochi = Dog.new
		#<Dog:0xbcb90>
			建立一個帳戶類別
			class Account:
			def __init__(self, id, name):
			self.id = id
			self.name = name
			self.balance = 0
			
			def deposit(self, amount):
			self.balance += amount
			
			def withdraw(self, amount):
			if amount <= self.balance:
			self.balance -= amount
			else:
			raise ValueError('餘額不足')
			
			def __str__(self):
			return ('Id:\t\t' + self.id +
			'\nName:\t\t' + self.name +
			'\nBalance:\t' +
			str(self.balance))

繼承	#include <iostream>
	#include <complex>
	using namespace std;
	
	class Base
	{
	public:
	virtual void a(int x) { cout << "Base::a(int)" << endl; }
	// overload the Base::a(int) function
	virtual void a(double x) { cout << "Base::a(double)" << endl; }
	virtual void b(int x) { cout << "Base::b(int)" << endl; }
	void c(int x) { cout << "Base::c(int)" << endl; }
	};
	
	class Derived : public Base
	{
	public:
	// redefine the Base::a() function
	void a(complex<double> x) { cout << "Derived::a(complex)" << endl; }
	// override the Base::b(int) function
	void b(int x) { cout << "Derived::b(int)" << endl; }
	// redefine the Base::c() function
	void c(int x) { cout << "Derived::c(int)" << endl; }
	};
		ruby> class Mammal
		| def breathe
		| puts "inhale and exhale"
		| end
		| end
		nil
		ruby> class Cat < Mammal
		| def speak
		| puts "Meow"
		| end
		| end
		nil
		
		(設定 cat 為 mammel 的子類)
		
		ruby> tama = Cat.new
		#<Cat:0xbd80e8>
		ruby> tama.breathe
		inhale and exhale
		nil
		ruby> tama.speak
		Meow
		nil
		
		(特定子類別不繼承父類別的特性)
		
		ruby> class Bird
		| def preen
		| puts "I am cleaning my feathers."
		| end
		| def fly
		| puts "I am flying."
		| end
		| end
		nil
		ruby> class Penguin < Bird
		| def fly
		| fail "Sorry.I'd rather swim."
		| end
		| end
		nil
			繼承了 Account 來定義一個 CheckingAccount 子類別:
			
			class CheckingAccount(Account):
			def __init__(self, id, name):
			super(CheckingAccount, self).__init__(id, name) # 呼叫父類別__init__()
			self.overdraftlimit = 30000
			
			def withdraw(self, amount):
			if amount <= self.balance + self.overdraftlimit:
			self.balance -= amount
			else:
			raise ValueError('超出信用')
			
			def __str__(self):
			return (super(CheckingAccount, self).__str__() +
			'\nOverdraft limit\t' + str(self.overdraftlimit));

多態	#include <iostream.h>
	class animal
	{
	public:
	animal(int height, int weight)
	{
	cout<<"animal construct"<<endl;
	}
	…
	};
	class fish:public animal
	{
	public:
	fish():animal(400,300)
	{
	cout<<"fish construct"<<endl;
	}
	…
	};
	void main()
	{
	fish fh; }
		class Parser
		def parse(type)
		puts 'The Parser
		class received the parse
		method'
		if type == :xml
		puts 'An instance
		of the XmlParser class
		received the parse
		message'
		elsif type == :json
		puts 'An instance
		of the JsonParser class
		received the parse
		message'
		end
		end
		end
			class Demo:
			def __init__(self, i):
			self.i = i
			def __str__(self):
			return str(self.i)
			def hello(self):
			print("hello " + self.__str__())
			class SubDemo1(Demo):
			def __init__(self, i, j):
			super().__init__(i)
			self.j = j
			def __str__(self):
			return super().__str__() + str(self.j)
			class SubDemo2(Demo):
			def __init__(self, i, j):
			super().__init__(i)
			self.j = j
			self.k = str(self.i) + str(self.j)
			def __str__(self):
			return self.k
			a = SubDemo1(22, 33)
			b = SubDemo2(44, "55")
			a.hello()
			b.hello()

抽象	// virtual members
	#include <iostream.h>
	class CPolygon {
	protected:
	int width, height;
	public:
	void set_values (int a, int b) {
	width=a;
	height=b;
	}
	virtual int area (void) =0;
	};
	class CRectangle: public CPolygon {
	public:
	int area (void) { return (width * height); }
	};
	class CTriangle: public CPolygon {
	public:
	int area (void) {
	return (width * height / 2);
	}
	};
	int main () {
	CRectangle rect;
	CTriangle trgl;
	CPolygon * ppoly1 = ▭
	CPolygon * ppoly2 = &trgl;
	ppoly1->set_values (4,5);
	ppoly2->set_values (4,5);
	cout << ppoly1->area() << endl;
	cout << ppoly2->area() << endl;
	return 0;
	}
		Ruby 本身沒有提供
		abstract class 和
		abstracted method
			import random
			from abc import ABCMeta, abstractmethod
			class Flyer(metaclass=ABCMeta):
			@abstractmethod
			def fly(self):
			pass
			class Bird:
			pass
			class Sparrow(Bird, Flyer):
			def fly(self):
			print('鳥飛')
			s = Sparrow()
			s.fly()