Paradigmas de la Programación – Segundo Parcial

14 de Mayo de 2024

Apellido y Nombre: _____

Ej. 1 Ej. 2

Ej. 3 Ej. 4

1. **[10 pt.]** La recolección de basura es una rutina residente, que se ejecuta en background, sin necesidad de que intervenga explícitamente el programador. Su objetivo es disponibilizar memoria que había sido ocupada por un programa para almacenar información, pero que el programa ya no va a requerir en el resto de su ejecución.

   Teniendo en cuenta que los objetos se almacenan en el *heap*, explique por qué la recolección de basura (o, en el caso de C++, la liberación manual de la memoria) tiene más impacto en eficiencia en los lenguajes con orientación a objetos que en otros lenguajes.

2. Indique en cuáles de las siguientes configuraciones se pueden dar *name clashes* (colisiones de nombre) en un lenguaje de programación:

   a) **[2 pt.]** si incorpora implementaciones de otras componentes (como los *traits* de Scala, como los *mixins* de Ruby) sin políticas de precedencia

   b) **[2 pt.]** si incorpora implementaciones de otras componentes (como los *traits* de Scala, como los *mixins* de Ruby) con políticas de precedencia

   c) **[2 pt.]** si tiene herencia múltiple con linearización

   d) **[2 pt.]** si tiene herencia múltiple sin linearización

   e) **[2 pt.]** si tiene interfaces como las de Java

3. **[10 pt.]** Lea la siguiente explicación sobre `override`:

   > *The* `override` *keyword is redundant in the sense that it reiterates information that is already present without it. You cannot use the override specifier to make a function override another function that it would not otherwise override. Nor does omitting the override specifier ever cause overriding not to occur.*
   > *Redundancy is not necessarily a bad thing. It protects against errors. This is well known in the case of data storage. The override specifier ensures that you don't accidentally hide a base class member function where you intend to override it.*

   Sabemos que el siguiente código es correcto:

```
1  class Base
2  {
3  public:
4      virtual void f()
5      {
6          std::cout << "Base";
7      }
8  };
9
10 class Derived : public Base
11 {
12 public:
13     void f() override
14     {
15         std::cout << "Derivada";
16     }
17 };
```

¿Cuál de las dos variantes que siguen no es equivalente al código anterior, y por qué? ¿Alguna de estas variantes produce un error de compilación? ¿Qué diferencias de comportamiento pueden llegar a ocasionar?

Código A:

```
1  class Base
2  {
3  public:
4      virtual void f()
5      {
6          std::cout << "Base";
7      }
8  };
9
10 class Derived : public Base
11 {
12 public:
13     void f()
14     {
15         std::cout << "Derivada";
16     }
17 };
```

Código B:

```
1  class Base
2  {
3  public:
4      virtual void f()
5      {
6          std::cout << "Base";
7      }
8  };
9
10 class Derived : public Base
11 {
12 public:
13     virtual void f()
14     {
15         std::cout << "Derivada";
16     }
17 };
```

4. **[10 pt.]** Complete el siguiente texto con las palabras "*Inheritance*", "*Subtyping*", *is a subtype of* y "*inherits from another type*":

   _____ refers to compatibility of interfaces. A type B _____ A if every function that can be invoked on an object of type A can also be invoked on an object of type B.

   _____ refers to reuse of implementations. A type B _____ A if some functions for B are written in terms of functions of A.