Lenguajes de Programación I

Inicialización y Destrucción - Asociación Dinámica de Métodos

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Copyright © 2007-2016

Inicialización de Objetos

- Mecanismo especial para inicializar los objetos.
  - Subrutinas denominadas **constructores**.
  - Ejecutan automáticamente a tiempo de elaboración del objeto.
  - No reservan espacio, sólo lo inicializan.
- Ambiente de ejecución reserva memoria y luego invoca el constructor.
- Típicamente se llaman igual que la clase a instanciar – algunos lenguajes permiten nombres arbitrarios.

¿Cómo escoger el constructor?

- Existen lenguajes que permiten más de un constructor.
  - Todos con el mismo nombre, pero *firma* diferente – C++, Java.
  - Nombres diferentes – Smalltalk, Eiffel.
- Existen lenguajes que no ofrecen constructores – Modula-3, Oberon.
- Existen lenguajes que condicionan la existencia de constructores – sólo para subclases de `Controlled` en Ada.

Modelo de Referencia vs. Modelo de Valores

Invocación explícita o implícita

- Modelo de Referencia – Java, Ruby, Python, Smalltalk
  - Los objetos son *referencias* a sus valores concretos.
  - Son creados de forma explícita.
  - La reserva de memoria ocurre en *heap*.
  - Es fácil asegurarse de la ejecución de los constructores – acompaña a la operación de reserva de memoria.
- Modelo de Valor – C++, Modula-3, Ada
  - Los objetos son *valores* concretos.
  - Pueden aparecer de forma implícita.
  - Pueden almacenarse en cualquier parte.
  - El ambiente de ejecución tiene que ejecutar los constructores al momento de la elaboración del objeto.
- Lenguajes que proveen *ambos* modelos selectivamente – `class` y `struct` en C#, `expanded` en Eiffel.

Dificultades del modelo de valores
El ejemplo de C++

- Declaraciones que implican constructores con parámetros.

```
foo b;           // foo::foo()

foo b(10,'x');   // foo:foo(int,char)
```

- Declaraciones que implican constructores para clonación

```
foo a;
bar b;

foo c(a);        // foo::foo(foo&)
foo d(b);        // foo::foo(bar&)
```

- Si son declaraciones globales, se ejecutan *antes* del principal.
- Si son declaraciones locales, se ejecutan en el prólogo.

Más complicaciones en C++

- Todo constructor de un argumento es **Constructor para Copia**.

```
foo a;       // foo::foo()
bar b;       // bar::bar()
...
foo c = a;   // foo::foo(foo&)
foo d = b;   // foo::foo(bar&)
```

- Pero la *asignación* funciona diferente.

```
foo a, c, d; // foo::foo() tres veces
bar b;       // bar::bar()
...
c = a;       // foo::operator=(foo&)
d = b;       // foo::operator=(bar&)
```

Inicialización y asignación son completamente diferentes.
No definir los métodos adecuados produce errores al compilar.

Dificultades del modelo de valores
Afectan el rendimiento

- Cuando el compilador encuentra una (inocente) declaración como

```
foo a = b + c
```

en realidad debe generar código equivalente a

```
foo t;
t = b.operator+(c);
foo a = t;
```

- Si se pasa un objeto por *valor* a una rutina.
  - La rutina tiene que trabajar con una copia local.
  - El prólogo debe ejecutar un constructor de copia – que más vale hayan definido en la clase.

Orden de Ejecución
B es subclase de A, ¿en que orden deben ejecutarse los constructores?

- En C++, el constructor de A debe ejecutar antes que el de B – declaración del constructor de B, incluye parámetros para el de A

```
B::B( B args ) : A( A args ) { ... }
```

- Se puede indicar los argumentos para atributos

```
class B : A {
  T1 m1;
  T2 m2;
}
B::B ( B args ) : A  ( A args  ),
                  m1 ( m1 args ),
                  m2 ( m2 args ) { ... }
```

Orden de Ejecución

B es subclase de A, ¿en que orden deben ejecutarse los constructores?

- Java tiene la misma necesidad, pero con el modelo de referencia – el programador debe invocar el constructor de la superclase.
  - Se espera que lo haga explícitamente aprovechando `super`.
  - Si lo omite, el compilador incluye una llamada implícita al constructor de cero argumentos (más vale que exista).
  - Atributos que sean objetos se inicializan con una referencia nula.
- Smalltalk, Eiffel y CLOS sólo invocan al constructor final – el programador debe invocar los constructores superiores.

Destrucción de Objetos

- Algunos lenguajes necesitan un sistema para finalizar los objetos.
  - Son subrutinas denominadas **destructores**.
  - Se ejecutan automáticamente a tiempo de finalización del objeto.
  - Generalmente sirven para liberar espacio.
  - Sólo hay *un* destructor sin argumentos.
- Necesarios en lenguajes con manejo de memoria manual – para liberar el espacio de memoria reservado explícitamente.
  - `new` reserva memoria e invoca al constructor.
  - `delete` ejecuta el destructor y libera la memoria.
  - Los epílogos ejecutan destructores para objetos locales.
- Casi inútiles en lenguajes con manejo automático de memoria – ¿cómo asegurar el momento preciso en que se ejecutarán?
- Los destructores son invocados en orden inverso de herencia – desde la subclase más profunda hasta la raíz de la jerarquía.

Polimorfismo de Subtipos

Principio de Sustitución de Liskov

- Si una clase B deriva a una clase A
  - B tiene todos los atributos y métodos de A.
  - B es estructuralmente equivalente a A.
  - Cualquier cosa posible sobre un objeto de clase A, también es posible sobre un objeto de clase B.
- Es posible expresar problemas en función de una clase general, y luego trabajar con colecciones de clases derivadas.

Polimorfismo de Subtipos

Excelente para programar de manera general

```
class persona { persona::imprimir_carnet() }

class estudiante :
   public persona { estudiante::imprimir_carnet() }

class profesor :
   public persona { profesor::imprimir_carnet() }

estudiante e;
profesor   p;

persona *x = &e, *y = &p;

e.imprimir_carnet();
p.imprimir_carnet();
```

¿Qué pasa al redefinir métodos?

- Si redefinimos `imprimir_carnet` en ambas subclases.
  - `e.imprimir_carnet()` invoca al método de la clase `estudiante`.
  - `p.imprimir_carnet()` invoca al método de la clase `profesor`.
  - ¿A cuál método debe invocar `x->imprimir_carnet()`?
- **Asociación Estática de Métodos**
  - La decisión depende del tipo de `x`.
  - Se establece a tiempo de *compilación*.
- **Asociación Dinámica de Métodos**
  - La decisión depende del tipo del objeto apuntado por `x`.
  - Se establece a tiempo de *ejecución*.

To virtual or not to virtual

Is that a question?

- Asociación dinámica *siempre* – Python, Ruby, Perl, Smalltalk.
- Ambos métodos, pero usan el dinámico por omisión – Java, Eiffel.
  - Palabra clave para indicar que un método asocia estáticamente – `final` en Java, `frozen` en Eiffel.
  - Esto impide que los métodos sean redefinidos en las subclases – compilador puede generar mejor código para la invocación.
- Ambos métodos, pero usan el estático por omisión – C++, C#, Ada.
  - Palabra clave para indicar que un método asocia dinámicamente – `virtual` en C++, combinación `virtual`/`override` en C#.
  - Se *redefinen* métodos estáticos y *reemplazan* métodos dinámicos.

Implantación de la Asociación Dinámica

En un lenguaje interpretado

- Todo objeto tiene un tipo asociado en su tabla de símbolos – incluye atributos y métodos disponibles.
- Cuando se invoca un método `obj.m`
  1. Se busca la clase de `obj` en la tabla de símbolos.
  2. Si el método `m` existe, se invoca, asociando `self` con `obj`.
  3. Si el método `m` no existe, se repite la búsqueda en la superclase.
  4. Si no quedan superclases, se emite un error de método indefinido.
- En algunos lenguajes hay un "método por omisión" – `method_missing` en Ruby, o `AUTOLOAD` en Perl.

Esto permite que un método de una *superclase* invoque un método de una *subclase*.

Implantación de la Asociación Dinámica

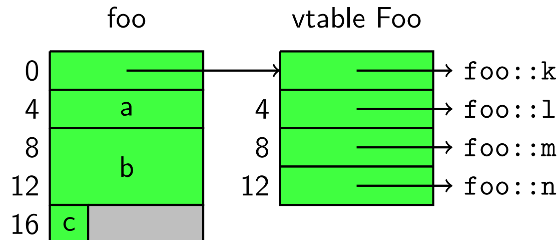
En un lenguage compilado

- Todo objeto debe contener suficiente información para decidir los atributos y métodos adecuados a tiempo de ejecución.
- Cada objeto se representa como un registro cuyo primer campo contiene una referencia a la **tabla de métodos virtuales** (*vtable*).
- La *vtable* es un arreglo cuya i-ésima entrada apunta al segmento de código del i-ésimo método virtual del objeto.
- Una *vtable* por cada clase que tenga métodos virtuales – *todas* las instancias de la clase apuntan a la *vtable*.

Construcción de *vtables*

Para la clase base

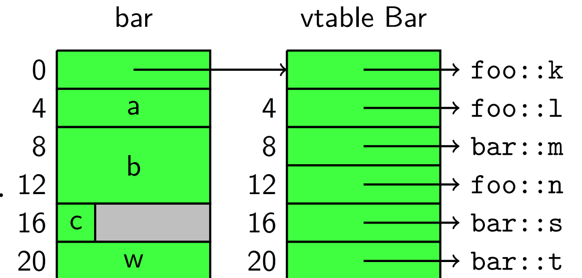
```
class Foo {
  int a;
  double b;
  char c;
public:
  virtual void k(...
  virtual int l(...
  virtual void m(...
  virtual double n(...
} foo;
```



Construcción de *vtables*

Para una clase derivada

```
class Bar :
public Foo {
  int w;
public:
  void m(...
  virtual double s(...
  virtual char *t(...
} bar;
```



Herencia Múltiple

- Si dos clases ancestrales proveen el mismo método:
  - ¿Ambas son accesibles?
  - ¿Cuál debe utilizarse?
- Si dos clases ancestrales derivan de un ancestro común:
  - ¿Hay una o dos copias de los métodos del ancestro?
- Mecanismo de Resolución de Métodos (MRO)
  - DFS – simple pero afectado por el problema del diamante.
  - C3 – ninguna clase aparece antes de sus subclases.
- ¿Cómo representar las instancias de las subclases? – *replicar* o *compartir* copias.

Duck typing

Redneck OOP

- Funciona en cualquier lenguaje con introspección simple.

```
if (obj.can('walk') &&
    obj.can('swim') &&
    obj.can('quack')) {
    print "it's a duck!"
}
```

- Opera con cualquier clase que tenga los métodos verificados.
  - Presumiblemente con la clase `Duck`.
  - Definitivamente con cualquier subclase de `Duck`.
  - Incluso con la clase `HunterWithADuckWhistle`.
- No hay manera de verificar tipos de argumentos, tipos de retorno, ni dependencias con otros métodos – muy *ad hack*.

Interfaces – Java, Go

Redneck corporate OOP

- Definición de un API que debe "cumplirse".
- Sólo firmas de métodos.
- Implantación del API corre por cuenta del programador – muchas veces repitiendo trabajo.
- *Duck-typing* formalizado
  - Se pueden verificar tipos de parámetros y valores de retorno.
  - La lista de métodos tiene un nombre – wow, such formality.
- El programador sólo verifica si la clase `does` or `implements` la interfaz antes de usarla.

Mixins – Ruby, Python

Menos trabajo y miedito a olvidar un método

- Implantación de un API – métodos con código.
- "Menos" que una clase – no se instancian.
- Cualquier clase tiene exactamente una superclase, pero puede incluir todos los *mixins* que quiera.
- Los métodos del *mixin* pueden hacer uso de métodos de cualquiera de las clases en la jerarquía.
- Puede causar invocaciones a métodos inexistentes.

Roles – Smalltalk, CLOS, Perl con Moose

Flexibility level 42000

- Abstracción no instanciable – atributos y métodos con código.
- Cualquier clase tiene exactamente una superclase, pero puede *consumir* todos los roles que quiera.
- Espacio de nombres separados para evitar colisiones.
  - Ocultar, renombrar o hacer alias de nombres.
  - Aplicables por clase y por instancia.
- Puede *exigir* y *verificar* que la clase consumidora
  - Provea atributos o métodos específicos – nombre, firma...
  - ¡Que podrían venir de superclases o roles!
- Parametrizables – sólo en Perl con Moose
  - Crear atributos adicionales según los parámetros.
  - Informar de atributos o métodos de la clase consumidora.
  - Escoger diferentes caminos de inicialización del rol.

Roles – reusabilidad al máximo

```
role Equality
requires 'compare';

bool equals(other) {
  return (self.compare(other) == 0)
}
bool different(other)
  return (!equal(other))
}
```

- Si una clase consume `Equality` *debe* proveer `compare`.
- Obtiene `equals` y `different` lista para usar.
- Puede redefinirla si hay una mejor implantación.

Despacho de Métodos

¿Cuál método invocar?

- Despacho simple – sólo un argumento es especial.
  - `s.m(arg1,arg2,...,argN)` se convierte en `m(s,arg1,arg2,...,argN)`
  - Se busca el método `m` en `s` y superclases.
  - Firma de `m` debe tener los tipos correctos.
  - La mayoría de los lenguajes OO sólo soportan despacho simple – aunque pueden emular despacho múltiple manualmente.
- Despacho múltiple – *todos* los argumentos son especiales.
  - Se hace la misma transformación.
  - Se busca el primer método `m` en el cual coincidan los tipos, estudiando *clases* y *superclases*
  - CLOS y Perl con Moose.
  - No es sobrecarga – la decisión se toma dinámicamente.

Bibliografía

- [*Scott*]
  - Secciones 9.3 a 9.5
  - Ejercicios y Exploraciones
- Página de WikiPedia sobre *v-tables*
- Página de WikiPedia sobre Herencia Múltiple
- Página de WikiPedia sobre CLOS
- Página de WikiPedia sobre Perl Moose
- Página de WikiPedia sobre Roles
- Página de C2 sobre MultiMethods