Programación Orientada a Objetos

Tema 4. Polimorfismo

José Fidel Argudo Argudo
Francisco Palomo Lozano
Inmaculada Medina Bulo
Gerardo Aburruzaga García

UCA
Universidad de Cádiz

Versión 1.0

Índice

Polimorfismo de sobrecarga

Pasos:

- Coincidencia exacta o trivial
- Promociones
- Conversiones estándar
- Conversiones definidas por el usuario
- Coincidencia con elipsis

Polimorfismo en tiempo de ejecución

```
1  #include <iostream>

3  class B {
4  public:
5    virtual void mostrar() { std::cout << i << "␣dentro␣de␣B\n"; }
6    int i;
7  };
8  class D: public B {
9  public:
10   void mostrar() { std::cout << i << "␣dentro␣de␣D\n"; }
11 };

13 int main() {
14   B b, *pb = &b;
15   D d;
16   d.i = 1 + (b.i = 1); // b.i = 1, d.i = 1 + 1;
17   pb->mostrar(); // B::mostrar()
18   pb = &d;
19   pb->mostrar(); // D::mostrar()
20 }
```

Polimorfismo en tiempo de ejecución (virtual.h)

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

4  class B {
5  public:
6    virtual void mostrar(int i)
7      { cout << i << "␣dentro␣de␣B::mostrar(int)\n"; }
8    virtual void mostrar(double d)
9      { cout << d << "␣dentro␣de␣B::mostrar(double)\n"; }
10   virtual void mostrar(char c)
11     { cout << c << "␣dentro␣de␣B::mostrar(char)\n"; }
12 };

14 class D: public B {
15 public:
16   virtual void mostrar(int i)
17     { cout << i << "␣dentro␣de␣D::mostrar(int)\n"; }
18 };
```

Polimorfismo en tiempo de ejecución

```
1  #include "virtual.h"

3  int main()
4  {
5    D d;
6    B b, *pb = &d;

8    b.mostrar(9);      // B::mostrar(int)
9    b.mostrar(9.5);    // B::mostrar(double)
10   b.mostrar('a');    // B::mostrar(char)
11   d.mostrar(9);      // D::mostrar(int)
12   d.mostrar(9.5);    // D::mostrar(int)
13   d.mostrar('a');    // D::mostrar(int)
14   pb->mostrar(9);    // D::mostrar(int)
15   pb->mostrar(9.5);  // B::mostrar(double)
16   pb->mostrar('a');  // B::mostrar(char)
17 }
```

Polimorfismo en tiempo de ejecución

```
1  class Figura {
2  public:
3    virtual ~Figura() {};
4    virtual double area() const { return 0.0; } // por omisión
5  };

7  class Rectangulo: public Figura {
8  public:
9    Rectangulo(double lado_1, double lado_2);
10   double area() const override;
11 protected:
12   double lado_1, lado_2;
13 };

15 class Circulo final: public Figura {
16 public:
17   Circulo(double radio);
18   double area() const override;
19 private:
20   double radio;
21 };
```

Polimorfismo en tiempo de ejecución. Ejemplo de uso

```
1 vector<Figura*> figura;
2 // se rellena el vector con las direcciones de
3 // diversas figuras (rectángulos y círculos).
4 // ...
5 double area = 0.0;
6 for (size_t i = 0; i < figuras.size(); ++i)
7   area += figura[i]->area(); // área total de todas las figuras
```

Clases abstractas (figura.h)

```
1  class Figura {
2  public:
3    virtual ~Figura();
4    virtual double area() const = 0;
5    // ...
6    virtual void mostrar() const = 0;
7  };

9  // Destructor virtual vacío
10 inline Figura::~Figura() {}
```

Clases abstractas (circulo.h)

```
1  #include <iostream>
2  #include <cmath>

4  class Circulo final: public Figura {
5  public:
6    Circulo(double radio): radio(radio) {}
7    double area() const override;
8    void mostrar() const override;
9  private:
10   double radio;
11 };

13 inline double Circulo::area() const
14 { return 4.0 * std::atan(1.0) * radio * radio; }

16 inline void Circulo::mostrar() const
17 { std::cout << "Círculo(" << radio << ")"; }
```

Clases abstractas (rectangulo.h)

```
1  #include <iostream>

3  class Rectangulo: public Figura {
4  public:
5    Rectangulo(double lado_1, double lado_2);
6    double area() const final;
7    void mostrar() const override;
8  protected:
9    double lado_1, lado_2;
10 };

12 inline Rectangulo::Rectangulo(double lado_1, double lado_2):
13   lado_1(lado_1), lado_2(lado_2) {}

15 inline double Rectangulo::area() const
16 { return lado_1 * lado_2; }

18 inline void Rectangulo::mostrar() const
19 { std::cout << "Rectángulo(" << lado_1 << ",␣" << lado_2 << ")"; }
```

Clases abstractas (cuadrado.h)

```
1  #include <iostream>

3  class Cuadrado final: public Rectangulo {
4  public:
5    Cuadrado(double lado);
6    void mostrar() const override;
7  };

9  inline Cuadrado::Cuadrado(double lado):
10   Rectangulo(lado, lado) {}

12 inline void Cuadrado::mostrar() const
13 { std::cout << "Cuadrado(" << lado_1 << ")"; }
```

Clases abstractas (prueba-1.cpp)

```
1  Figura* figura_aleatoria() {
2    double x = rand() % 9 + 1, y = rand() % 9 + 1;
3    switch (rand() % 3) {
4    case 0: return new Rectangulo(x, y);
5    case 1: return new Cuadrado(x);
6    default: return new Circulo(x);
7    }
8  }

10 int main() {
11   srand(time(0));
12   for (int i = 0; i < 10; ++i) {
13     Figura* f = figura_aleatoria();
14     cout << "Figura␣=␣";
15     f->mostrar();
16     cout << "␣\t" << "Área␣=␣" << f->area() << endl;
17     delete f;
18   }
19 }
```

Clases abstractas (prueba-2.cpp)

```
1  Figura& figura_aleatoria() {
2    double x = rand() % 9 + 1, y = rand() % 9 + 1;
3    switch (rand() % 3) {
4    case 0: return *new Rectangulo(x, y);
5    case 1: return *new Cuadrado(x);
6    default: return *new Circulo(x);
7    }
8  }

10 int main() {
11   srand(time(0));
12   for (int i = 0; i < 10; ++i) {
13     Figura& f = figura_aleatoria();
14     cout << "Figura␣=␣";
15     f.mostrar();
16     cout << "␣\t" << "Área␣=␣" << f.area() << endl;
17     delete &f;
18   }
19 }
```

Identificación de tipos en tiempo de ejecución

```
1  class B {
2  public:
3    virtual ~B() {}
4    // ...
5  };
6  class D: public B {
7    // ...
8  };

10 void procesar(B* pb)
11 {
12   D* pd = static_cast<D*>(pb); // Peligroso si *pb no es de tipo «D».
13   if (D* pd = dynamic_cast<D*>(pb)) {
14     // El objeto apuntado por «pb» es de tipo «D».
15     // Así, «pb» se ha convertido sin problemas en «pd»
16   }
17   else { // pd == nullptr
18     // El objeto apuntado por «pb» no es de tipo «D».
19     // La conversión ha fallado .
20   }
21 }
```

Identificación de tipos en tiempo de ejecución

```
1  void procesar2(B& b)
2  {
3    try {
4      D& d = dynamic_cast<D&>(b);
5      // El objeto «b» es de tipo «D».
6      // La referencia «b» se ha convertido sin problemas en «d».
7    }
8    catch (std::bad_cast&) {
9      // El objeto «b» no es de tipo «D».
10     // La conversión ha fallado .
11   }
12 }
```

Identificación de tipos en tiempo de ejecución

Operador typeid()

Devuelve por referencia un objeto no modificable (`const`) de un tipo de la biblioteca estándar llamado `type_info`, definido en la cabecera `<typeinfo>`.

```
1 void f(Figura& r, Figura* p)
2 {
3   typeid(r);   // tipo del objeto referido por r
4   typeid(*p);  // tipo del objeto al que apunte p
5   typeid(p);   // tipo Figura*; válido pero evidente
6 }
```

Identificación de tipos en tiempo de ejecución

```
1  class type_info { // Interfaz de la clase
2  public:
3    // Al tener como único constructor declarado explícitamente, pero
4    // suprimido, el de copia, entonces no tiene constructores.
5    // Los objetos type_info se crean con el operador typeid().
6    virtual ~type_info(); // Es una clase polimórfica.

8    bool operator==(const type_info&) const noexcept;// Los objetos son
9    bool operator!=(const type_info&) const noexcept;// comparables,
10   bool before(const type_info&) const noexcept; // se pueden clasificar
11   size_t hash_code() const noexcept; // y almacenar en c. desordenados.
12   const char* name() const noexcept; // Nombre (codificado) del tipo.

14   type_info(const type_info&) = delete; // Los objetos no se pueden
15   type_info& operator=(const type_info&) = delete; // copiar.
16 };
```

Polimorfismo paramétrico

- Soportado en C++ mediante el uso de plantillas (templates), las cuales permiten usar tipos de datos como parámetros de la definición de clases y funciones.
- Proporciona el mecanismo para aplicar técnicas de programación genércia.
- Una plantilla es una definición genércia de una clase (o función) que no depende de los tipos concretos de sus parámetros reales, sino de las propiedades de los parámetros formales que se usen en la definición.

Polimorfismo paramétrico

```
1  // pila.h
2  template <typename T> class Pila {
3  public:
4    explicit Pila(unsigned TamaMax); // requiere ctor. T()
5    void push(const T& x);
6    // ... declaraciones del resto de miembros
7  };
8  // Los métodos de una plantilla de clase se definen
9  // como plantillas de funciones.
10 template <typename T>
11 Pila<T>::Pila(unsigned TamaMax) {
12   // ...
13 }
14 template <typename T>
15 Pila<T>::push(const T& x) {
16   // ...
17 }
```

Polimorfismo paramétrico

Instanciación de plantillas

Las clases (o funciones) específicas las genera automáticamente el compilador cuando especializamos la plantilla al proporcionar los parámetros reales.

```
#include "pila.h"          // definición de la plantilla Pila<T>

Pila<char> P1(20);         // pila de caracteres, de longitud 20
Pila<double> P2(150);      // pila de double, de longitud 150
Pila<string> P3(100);      // pila de string, de longitud 100
Pila<Pila<int>> P4(5);     // Error, Pila<int> no dispone
                           // de ctor. predeterminado
P2.push(3.5);              // P2.Pila<double>::push<double>(3.5);
P3.push(string("algo"));   // P3.Pila<string>::push<string>(string("algo"));
```

Polimorfismo paramétrico (vector.h)

```
1  #ifndef VECTOR_H
2  #define VECTOR_H
3  #include <cassert>
4  #include <vector>
5  using std::vector;

7  template <typename T> class Vector {
8  public:
9    explicit Vector(size_t n = 1, T x = T());
10   T& operator [](size_t i);
11   const T& operator [](size_t i) const;
12   Vector& operator +=(const Vector& a);
13   Vector& operator -=(const Vector& a);
14   Vector& operator *=(const T& k);
15   size_t dimension() const;
16   void mostrar() const;
17 protected:
18   vector<T> v; // elementos
19 };
```

Polimorfismo paramétrico (vector.h)

```
21 // Definiciones inline

23 // Constructor
24 template <typename T> inline
25 Vector<T>::Vector(size_t n, T x): v(n, x) {}

27 // Resto de funciones miembro definidas inline
28 // ...

30 // Definiciones de plantillas no inline

32 #include "vector.cpp"

34 #endif // VECTOR_H
```

Polimorfismo paramétrico (matriz.h)

```
1  #ifndef MATRIZ_H
2  #define MATRIZ_H
3  #include "vector.h"
4  #include <cassert>
5  template <typename T> class Matriz {
6  public:
7    explicit Matriz(size_t m = 1, size_t n = 1);
8    Matriz(const Vector<T>& v);
9    Vector<T>& operator [](size_t i);
10   const Vector<T>& operator [](size_t i) const;
11   Matriz& operator +=(const Matriz& a);
12   Matriz& operator -=(const Matriz& a);
13   Matriz& operator *=(const Matriz& a);
14   Matriz& operator *=(const T& k);
15   size_t filas() const;
16   size_t columnas() const;
17   void mostrar() const;
18 protected:
19   size_t m, n; // dimensión
20   Vector<Vector<T> > a; // elementos
21 };
```

Polimorfismo paramétrico (matriz.h)

```
23 // Definiciones inline

25 // Constructor
26 template <typename T> inline
27 Matriz<T>::Matriz(size_t m, size_t n):
28   m(m), n(n), a(m, Vector<T>(n)) {}

30 // Resto de funciones miembro definidas inline
31 // ...

33 // Definiciones de plantillas no inline

35 #include "matriz.cpp"

37 #endif  // MATRIZ_H
```

Polimorfismo paramétrico (prueba.cpp)

```
1  #include "matriz.h"
2  #include <iostream>
3  #include <algortihm>
4  using namespace std;

6  Matriz<double> identidad(size_t m = 1, size_t n = 1);

8  int main()
9  {
10   Matriz<double> a = 2.0 * identidad(4, 2);
11   Matriz<double> b = 3.0 * identidad(2, 4);
12   Matriz<double> c = a * b;

14   cout << "a = " << endl;
15   a.mostrar();
16   cout << "b = " << endl;
17   b.mostrar();
18   cout << "c = a * b" << endl;
19   c.mostrar();
20 }
```

Polimorfismo paramétrico (prueba.cpp)

```
22 // Matriz identidad
23 Matriz<double> identidad(size_t m, size_t n)
24 {
25   Matriz<double> c(m, n); // matriz nula
26   for (size_t i = 0; i < min(m, n); ++i)
27     c[i][i] = 1.0;
28   return c;
29 }
```

Polimorfismo paramétrico: deducción de parámetros

```
1 template <typename T> T* crear();

3 void f()
4 {
5   vector<int> v;     // el parámetro de la plantilla vector<T> es int
6   int *p = crear(); // Error: imposible deducir T
7   int *q = crear<int>(); // OK: T es int
8 }
```

Polimorfismo paramétrico: especialización

```
1  template <typename T>
2  ostream& operator << (ostream& os, const Matriz<T>&);

4  template <>
5  ostream& operator << <bool>(ostream& os, const Matriz<bool>& M)
6  {
7    os << boolalpha;
8    for (size_t i = 0; i < filas(); ++i) {
9      for (size_t j = 0; j < columnas() ++j)
10       os << (*this)[i][j] << '␣';
11     os << endl;
12   }
13   return os;
14 }
```

Polimorfismo paramétrico: friend en plantillas

```
1  template <typename T> class Matriz {
2  public:
3    // ...
4    friend void auxiliar();
5    template <typename S>
6    friend Vector<S> operator *(const Matriz<S>&, const Vector<S>&);
7    // ...
8  };
```

```
1  template <typename T> class Matriz;
2  template <typename S>
3  Vector<S> operator *(const Matriz<S>&, const Vector<S>&);
```

```
5  template <typename T> class Matriz {
6  public:
7    // ...
8    friend void auxiliar();
9    friend
10   Vector<T> operator *<T>(const Matriz<T>&, const Vector<T>&);
11   // ...
12 };
```

Polimorfismo paramétrico: static en plantillas

```
1  template <typename T> class C {
2  public:
3    static int n; // específico
4    // ...
5  };

7  // ...

9  C<int> v1, v2; // v1.n y v2.n son el mismo atributo, C<int>::n
10 C<double> v3;  // v3.n es C<double>::n
```

Polimorfismo paramétrico: Parámetros de plantillas

```
1  template <typename T1, typename T2>
2  bool operator ==(const vector<T1>& a, const vector<T2>& b)
3  {
4    const size_t n = a.size();
5    if (n != b.size())
6      return false;
7    for (size_t i = 0; i < n; ++i)
8      if (a[i] != b[i])
9        return false;
10   return true;
11 }
```

Polimorfismo paramétrico: Parámetros de plantillas

```
1  template <typename T, size_t n> class Buffer {
2    T b[n];
3    // ...
4  };

6  // ...

8  Buffer<char, 20> a, b;
9  Buffer<char, 10> c;

11 a = b; // bien
12 c = a; // ERROR, se detecta en tiempo de compilación
```

Polimorfismo paramétrico: Parámetros de plantillas

```
1 template <typename T = char, size_t n = 256> class Buffer {
2   T b[n];
3   // ...
4 };

6 // ...

8 Buffer<double> a;
9 Buffer<> b;
```