

TEMA 5

POLIMORFISMO

Cristina Cachero, Pedro J. Ponce de León

4 Sesiones (6 horas)

Versión 1.0



Tema 4. Polimorfismo

Objetivos básicos



- Comprender el concepto de polimorfismo
- Conocer y saber utilizar los diferentes tipos de polimorfismo.
- Comprender el concepto de enlazado estático y dinámico en los lenguajes OO.
- Comprender la relación entre polimorfismo y herencia en los lenguajes fuertemente tipados.
- Apreciar la manera en que el polimorfismo hace que los sistemas sean extensibles y mantenibles.



1. Motivación y conceptos previos
 - Signatura
 - Ámbito
 - Sistema de tipos
2. Polimorfismo y reutilización
 - Definición
 - Tipos de polimorfismo
3. Sobrecarga
 - Sobrecarga basada en ámbito
 - Sobrecarga basada en signatura de tipo
 - Alternativas a la sobrecarga
4. Polimorfismo en jerarquías de herencia
 - Redefinición
 - Shadowing
 - Sobrescritura
5. Variables polimórficas
 - La variable receptora
 - Downcasting
 - Polimorfismo puro
6. Genericidad
 - Funciones genéricas en C++
 - Plantillas de clase en C++
 - Herencia en clases genéricas

1. Motivación



- Objetivo de la POO
 - Aproximarse al modo de resolver problemas en el mundo real.
- El polimorfismo es el modo en que los lenguajes OO implementan el concepto de **polisemia** del mundo real:
 - Un único nombre para muchos significados, según el contexto.

1. Conceptos previos: Signatura



- **Signatura de tipo** de un método:

- Descripción de los tipos de sus argumentos, su orden y el tipo devuelto por el método.

- Notación: **<argumentos> → <tipo devuelto>**
- Omitimos el nombre del método, el de la clase a la que pertenece (el tipo del receptor)

- Ejemplos

`double power (double base, int exp)`

- **`double*int → double`**

`bool Casilla::setPieza(Pieza& p)`

- **`Pieza → bool`**

1. Conceptos previos:

Ámbito



- **Ámbito de un nombre:**
 - Porción del programa en la cual un nombre puede ser utilizado de una determinada manera.
 - Ejemplo:

```
double power (double base, int exp)
```

 - La variable *base* sólo puede ser utilizada dentro del método *power*
 - **Ámbitos activos: puede haber varios simultáneamente**
 - Las clases, los cuerpos de metodos, cualquier bloque de código define un ámbito:

```
class A {
    int x,y;
    public:
        void f() {
            // Ámbitos activos:
            // GLOBAL
            // CLASE (atribos. de clase y de instancia)
            // METODO (argumentos, var. locales)
            if (...) {
                string s;
                // LOCAL (var. locales)
            }
        }
}
```

1. Conceptos previos:

Ámbito: **Espacio de nombres**



- Un espacio de nombres define un ámbito
 - Agrupa declaraciones (clases, métodos, objetos...) que forman una unidad lógica.
 - C++: namespace
 - Java: paquetes

Graficos.h

```
namespace Graficos {  
    class Circulo {...};  
    class Rectangulo {...};  
    class Lienzo {...};  
    ...  
}
```

Circulo.h

```
namespace Graficos {  
    class Circulo {...};  
}
```

Rectangulo.h

```
namespace Graficos {  
    class Rectangulo {...};  
}
```

1. Conceptos previos:

Ámbito: **Espacio de nombres**



- Un espacio de nombres define un ámbito
 - Agrupa declaraciones (clases, métodos, objetos...) que forman una unidad lógica.
 - C++: namespace
 - Java: paquetes

Circulo.java

```
package Graficos;
```

```
class Circulo {...}
```

Rectangulo.java

```
package Graficos;
```

```
class Rectangulo {...}
```


1. Conceptos previos:

Ámbito: **Espacio de nombres**



- C++

```
#include "Graficos.h"
int main() {
    Graficos::Circulo c;
    Graficos::operator>>(cin,c);
    Graficos::operator<<(cout,c);
}
```

```
#include "Graficos.h"
using Graficos::Circulo;
int main() {
    Circulo c;
    Graficos::operator>>(cin,c);
    Graficos::operator<<(cout,c);
}
```

1. Conceptos previos:

Ámbito: **Espacio de nombres**



- C++

```
#include "Graficos.h"
int main() {
    Graficos::Circulo c;
    Graficos::operator>>(cin,c);
    Graficos::operator<<(cout,c);
}
```

```
#include "Graficos.h"
using namespace Graficos;
int main() {
    Circulo c;
    cin << c;
    cout << c;
}
```

1. Conceptos previos:

Ámbito: **Espacio de nombres**



- Java

```
class Main {  
    public static void main(String args[]) {  
        Graficos.Circulo c;  
        c.pintar(System.out);  
    }  
}
```

```
import Graficos.*;  
class Main {  
    public static void main(String args[]) {  
        Circulo c;  
        c.pintar(System.out);  
    }  
}
```

1. Conceptos previos: Sistema de tipos



- El **sistema de tipos** de un lenguaje determina su soporte al enlace dinámico:
 - **Lenguaje fuertemente tipado**
 - **Lenguajes Procedimentales: no soportan enlace dinámico:** el *tipo* de toda expresión (identificador o fragmento de código) se conoce en tiempo de compilación.
 - **Lenguajes orientados a objetos:** sólo soportan **enlace dinámico dentro de la jerarquía de tipos** a la que pertenece toda expresión (identificador o fragmento de código) establecida en tiempo de compilación.
 - **Lenguaje débilmente tipado**
 - **Sí soportan enlace dinámico:** el enlace entre variables y tipo o llamadas y metodos a ejecutar se efectúa en tiempo de ejecución



1. Motivación y conceptos previos
 - Signatura
 - Ámbito
 - Sistema de tipos
2. Polimorfismo y reutilización
 - Definición
 - Tipos de polimorfismo
3. Sobrecarga
 - Sobrecarga basada en ámbito
 - Sobrecarga basada en signatura de tipo
 - Alternativas a la sobrecarga
4. Polimorfismo en jerarquías de herencia
 - Redefinición
 - Shadowing
 - Sobrescritura
5. Variables polimórficas
 - La variable receptora
 - Downcasting
 - Polimorfismo puro
6. Genericidad
 - Funciones genéricas en C++
 - Plantillas de clase en C++
 - Herencia en clases genéricas

2. Polimorfismo

Definición



- **Capacidad de una entidad de referenciar distintos elementos en distintos instantes de tiempo.**
 - El polimorfismo nos permite programar de manera general en lugar de programar de manera específica.
 - Hay cuatro técnicas, cada una de las cuales permite una forma distinta de **reutilización de software**, que facilita a su vez el desarrollo rápido, la confianza y la facilidad de uso y mantenimiento.
 - Sobrecarga
 - Sobreescritura
 - Variables polimórficas
 - Genericidad

Tipos de polimorfismo



- **Sobrecarga** (*Overloading*, Polimorfismo ad-hoc)
 - Un sólo nombre de método y muchas implementaciones distintas.
 - Las funciones sobrecargadas normalmente se distinguen en tiempo de compilación por tener distintos parámetros de entrada y/o salida.

- **Sobreescritura** (*Overriding*, Polimorfismo de inclusión)
 - Tipo especial de sobrecarga que ocurre dentro de relaciones de herencia en métodos con enlace dinámico.
 - Dichos métodos, definidos en clases base, son refinados o reemplazados en las clases derivadas.

Tipos de polimorfismo



- **Variables polimórficas** (Polimorfismo de asignación)
 - Variable que se declara como de un tipo pero que referencia en realidad un valor de un tipo distinto (normalmente relacionado mediante herencia).
- **Genericidad** (plantillas o *templates*)
 - Clases o métodos parametrizados (algunos tipos se dejan sin definir).
 - Forma de crear herramientas de propósito general (clases, métodos) y especializarlas para situaciones específicas.



- **Sobrecarga**

```
Factura::imprimir()
```

```
Factura::imprimir(int numCopias)
```

```
ListaCompra::imprimir()
```

- **Sobreescritura**

```
Cuenta::abonarInteres()
```

```
CuentaJoven::abonarInteres()
```

- **Variables polimórficas**

```
Cuenta *pc=new CuentaJoven();
```

- **Genericidad**

```
Lista<Cliente>
```

```
Lista<Articulo>
```

```
Lista<Alumno>
```

```
Lista<Habitacion>
```



1. Motivación y conceptos previos
 - Signatura
 - Ámbito
 - Sistema de tipos
2. Polimorfismo y reutilización
 - Definición
 - Tipos de polimorfismo
3. Sobrecarga
 - Sobrecarga basada en ámbito
 - Sobrecarga basada en signatura de tipo
 - Alternativas a la sobrecarga
4. Polimorfismo en jerarquías de herencia
 - Redefinición
 - Shadowing
 - Sobrescritura
5. Variables polimórficas
 - La variable receptora
 - Downcasting
 - Polimorfismo puro
6. Genericidad
 - Funciones genéricas en C++
 - Plantillas de clase en C++
 - Herencia en clases genéricas

3. Sobrecarga (*Overloading*, polimorfismo *ad-hoc*)

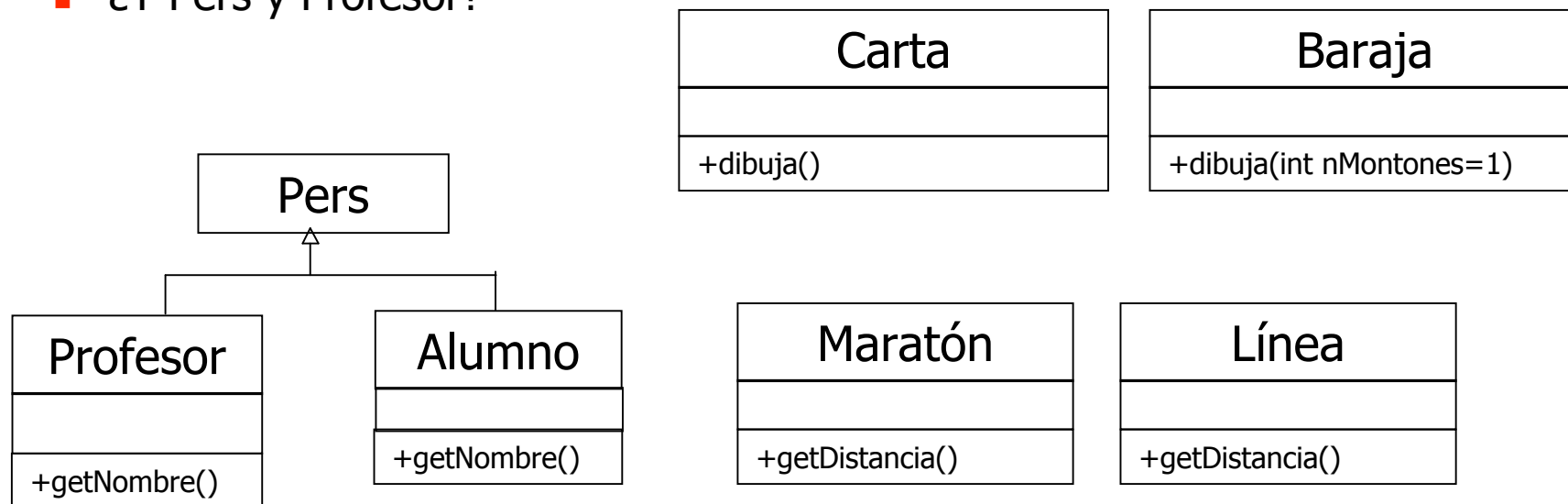


- Un mismo nombre de mensaje asociado a varias implementaciones
- La sobrecarga se realiza en **tiempo de compilación** (enlace estático) en función de la signatura completa del mensaje.
- Dos tipos de sobrecarga:
 - **Basada en ámbito:** Métodos con **diferentes ámbitos de definición**, independientemente de sus signaturas de tipo. Permitido en todos los lenguajes OO.
 - Un mismo método puede ser utilizado en dos o más clases.
 - P. ej. Sobrecarga de operadores como funciones miembro.
 - **Basada en signatura:** Métodos con **diferentes signaturas de tipo en el mismo ámbito de definición**. No permitido en todos los lenguajes OO.
 - Dos o más métodos en la misma clase pueden tener el mismo nombre siempre que tengan distinta signatura de tipos.

Sobrecarga basada en ámbito



- Distintos ámbitos implican que el mismo nombre de método puede aparecer en ellos sin ambigüedad.
- La sobrecarga por ámbito no requiere que las funciones asociadas con un nombre sobrecargado tengan ninguna similitud semántica, ni la misma signatura de tipo.
- Ejemplos
 - ¿Son Profesor y Alumno ámbitos distintos?
 - ¿Y Pers y Profesor?



Sobrecarga basada en firmas de tipo



- Métodos en el mismo ámbito pueden compartir el mismo nombre siempre que difieran en número, orden y, en lenguajes con tipado estático, el tipo de los argumentos que requieren.
 - Este estilo ocurre en muchos lenguajes funcionales, en algunos imperativos (e.g. ADA) y en lenguajes OO como C++, C#, Java, Delphi Pascal o CLOS
 - C++ permite esta sobrecarga de manera implícita siempre que la selección del método requerido por el usuario pueda establecerse de manera no ambigua en tiempo de compilación.
 - Esto implica que la firma no puede distinguirse sólo por el tipo de retorno

Suma
+add(int a)
+add(int a, int b)
+add(int a, int b, int c)

Sobrecarga basada en firmas de tipo



- Ejemplo

```
class Padre{...};
class Hijo: public Padre{...};
void Test (Padre *p) {cout << "padre";};
void Test (Hijo *h) {cout << "hijo";}; //ppio sust.
int main(){
    Padre *obj;
    int tipo;cin>>tipo;
    if (tipo==1) obj=new Padre();
    else obj=new Hijo(); //ppio de sustitución
    Test (obj); //¿a quién invoco?
}
```



- **No todos los LPO permiten la sobrecarga:**
 - Permiten sobrecarga de métodos y operadores: C++
 - Permiten sobrecarga de métodos pero no de operadores: Java, Python, Perl
 - Permiten sobrecarga de operadores pero no de métodos: Eiffel

Sobrecarga de operadores en C++



- Dentro de la sobrecarga basada en firmas de tipo, tiene especial relevancia la **sobrecarga de operadores**
- **Uso:** Utilizar operadores tradicionales con tipos definidos por el usuario.
- Forma de sobrecargar un operador @:
`<tipo devuelto> operator@(<args>)`
- Para utilizar un operador con un objeto de tipo definido por usuario, éste debe ser sobrecargado.
 - Excepciones: operador de asignación (=) y el operador de dirección (&)
 - El operador dirección (&) por defecto devuelve la dirección del objeto.
 - El operador asignación por defecto asigna campo a campo.

Sobrecarga de operadores en C++



- En la sobrecarga de operadores no se puede cambiar
 - Precedencia (qué operador se evalúa antes)
 - Asociatividad $a=b=c \rightarrow a=(b=c)$
 - Aridad (operadores binarios para que actúen como unarios o viceversa)
- No se pueden crear nuevos operadores
- No se pueden sobrecargar operadores para tipos predefinidos.
- Se pueden sobrecargar todos los operadores definidos en C++ como unarios y binarios excepto ".", ".*", "::", **sizeof**.
- "?:" es el único operador ternario definido en C++, y no puede ser sobrecargado

Sobrecarga de operadores en C++



- La sobrecarga de operadores se puede realizar mediante funciones miembro o no miembro de la clase.
 - Como función miembro: el operando de la izquierda (en un operador binario) debe ser un objeto (o referencia a un objeto) de la clase:

```
Complejo Complejo::operator+(const Complejo&)
```

```
...
```

```
Complejo c1(1,2), c2(2,-1);
```

```
c1+c2; // c1.operator+(c2);
```

```
c1+c2+c3; // c1.operator+(c2).operator+(c3)
```

Sobrecarga de operadores en C++



- Sobrecarga de operador como función no miembro:
 - Cuando el operando de la izquierda no es miembro de la clase

```
ostream& operator<<(ostream&, const Complejo&);  
istream& operator>>(istream&, Complejo&);
```

```
...
```

```
Complejo c;  
cout << c; // operator<<(cout,c)  
cin >> c; // operator>>(cin,c)
```

Sobrecarga de operadores en C++

Ejemplo: operador [] en clase Vector



- Signatura operador corchete ([])
 - **<tipo>& operator[] (int índice)**
- Ejemplo:

```
class Vector
{
    public:
        Vector( int ne=1) {
            num = ne;
            pV = new int[ num];
        };
        ~Vector() { delete [] pV; pV=NULL;};
        int& operator[] (int i) {return pV[ i];};

    private:
        int *pV; // puntero al 1 elemento
        int num; // num de elementos
};
```

Sobrecarga de operadores en C++

Ejemplo: operador [] en clase Vector



```
int main() {
    int num, i;
    cout << "Número:"; cin >> num;
    Vector v1(num);
    for (i= 0; i< num; i++)
        v1[i] = rand(); // v1[i] como valor izquierdo
    for (i= 0; i< num; i++) {
        cout << "v1[" << i << "]=" << v1[i] << endl;
        // v1[i] como valor derecho
    }
    return (0);
}
```

Valor izquierdo (*L-value*) : se puede poner a la izquierda de una asignación

Valor derecho (*R-value*): no se puede asignar (expresiones, constantes, ...)



■ **Funciones poliádicas**

- Funciones con número variable de argumentos
- Se encuentran en distintos lenguajes
 - P. ej. printf de C y C++
- Si el número máximo de argumentos es conocido, en C++ podemos acudir a la definición de valores por defecto:

```
int sum (int e1, int e2, int e3=0, int e4=0);
```

Alternativas a sobrecarga: Coerción y Conversión



- En algunas ocasiones la sobrecarga puede sustituirse por una operación semánticamente diferente: **La COERCION**
 - Un valor de un tipo se convierte DE MANERA IMPLÍCITA en un valor de otro tipo distinto
 - P.ej. Coerción implícita entre reales y enteros en C++.
 - El principio de sustitución en los LOO introduce además una forma de coerción que no se encuentra en los lenguajes convencionales

Alternativas a sobrecarga: Coerción y Conversión



- Cuando el cambio en tipo es solicitado de manera explícita por el programador hablamos de **CONVERSION**
 - El operador utilizado para solicitar esta conversión se denomina CAST
 - Ejemplo:
 - `double x; int i;`
 - `x= i + x; // COERCION`
 - `x= (double)i + x; // CONVERSION`
 - Dentro de una clase C podemos definir el *cast*:
 - De un tipo externo al tipo definido por la clase:
 - Constructor con un solo parámetro del tipo desde el cual queremos convertir.
 - Del tipo definido por la clase a otro tipo distinto en C++:
 - Implementación de un operador de conversión.

Alternativas a sobrecarga: Coerción y Conversión



```
class Fraccion{
    public:
        Fraccion(int n,int d):num(n),den(d){};
        int numerador(){return num;};
        int denominador(){return den;};
        Fraccion operator*(Fraccion &dcha){...};
    private: int num, den;
};
```

- Para poder realizar la conversión:

```
Fracción f=(Fraccion) 3;
//pasar de entero a fracción
```

Sería necesario añadir a la clase fracción el constructor:

```
Fracción (int i) {num=i;den=1;};
```

Esto a su vez permitiría operaciones como

```
Fracción f(2,3),f2; //dos tercios
f2=f*3; // daría 6/3
```

Alternativas a sobrecarga: Coerción y Conversión



- Si además deseamos realizar la conversión:

```
double d=f*3.14;
```

Sería necesario añadir a la clase fracción un operador de conversión a double:

```
operator double() { //SIN TIPO RETORNO  
    return (numerador() / (double)denominador());  
};
```

De forma que

```
double d=f*3.14;  
double d= (double)f * 3.14;  
double d= double(f) * 3.14;
```

Invoca a `Fraccion::operator double()`



1. Motivación y conceptos previos
 - Signatura
 - Ámbito
 - Sistema de tipos
2. Polimorfismo y reutilización
 - Definición
 - Tipos de polimorfismo
3. Sobrecarga
 - Sobrecarga basada en ámbito
 - Sobrecarga basada en signatura de tipo
 - Alternativas a la sobrecarga
4. Polimorfismo en jerarquías de herencia
 - Redefinición
 - Shadowing
 - Sobrescritura
5. Variables polimórficas
 - La variable receptora
 - Downcasting
 - Polimorfismo puro
6. Genericidad
 - Funciones genéricas en C++
 - Plantillas de clase en C++
 - Herencia en clases genéricas



Sobrecarga en jerarquías de herencia

- Métodos con el mismo nombre, la misma signatura de tipo y enlace estático:
 - **Shadowing (refinamiento/reemplazo)**: las signaturas de tipo son las misma en clases base y derivadas, pero el método a invocar se decide en tiempo de compilación.
 - En C++ implica que el método no se declaró como virtual en clase padre.
- Métodos con el mismo nombre y distinta signatura de tipo y enlace estático:
 - **Redefinición**: clase derivada define un método con el mismo nombre que en la base pero con **distinta signatura de tipos**.
 - Modelo **merge**: SOBRECARGA
 - Modelo **jerárquico**: NO HAY SOBRECARGA

Polimorfismo en jerarquías de herencia

Sobrecarga en jerarquías de herencia



- Dos formas de resolver la **redefinición** en LOO:
 - Modelo **merge** (Java):
 - Los diferentes significados que se encuentran en todos los ámbitos actualmente activos se unen para formar una sola colección de métodos.
 - Modelo **jerárquico** (C++):
 - Cuando se encuentra un ámbito en el que el nombre está definido, la coincidencia más cercana en ese ámbito será la seleccionada para resolver la llamada.

```
class Padre{
    public: void ejemplo(int a){cout<<"Padre";};
};
class Hija: public Padre{
    public: void ejemplo (int a, int b){cout<<"Hija";};
};
int main(){
    Hija h;
    h.ejemplo(3); // OK en Java
                //pero ERROR DE COMPILACIÓN EN C++

    return (0);
}
```



- Decimos que un método en una clase derivada sobreescribe un método en la clase base si los dos métodos tienen el mismo nombre, la misma signatura de tipos y enlace dinámico.
 - El método en la clase base tiene enlace dinámico (método virtual o abstracto).
 - Los métodos sobreescritos pueden suponer un reemplazo del comportamiento o un refinamiento.
 - La resolución del método a invocar se produce en **tiempo de ejecución** (enlace dinámico) en función del tipo dinámico del receptor del mensaje.

Polimorfismo en jerarquías de herencia

Sobreescritura



- En C++ es la clase base la que debe indicar explícitamente que un método tiene enlace dinámico, y por tanto puede ser sobreescrito (aunque no obliga a que lo sea).

- Palabra reservada **virtual**.

```
class Padre{
    public: virtual int ejemplo(int a){cout<<"padre";};
};
class Hija : public Padre{
    public: int ejemplo (int a){cout<<"hija";};
};
```

```
Padre* p = new Hija();
p->ejemplo(10); // Hija::ejemplo(10)
```

Polimorfismo en jerarquías de herencia

Sobreescritura



- En otros lenguajes:
 - Java, Smalltalk:
 - la simple existencia de un método con el mismo nombre y signatura de tipos en clase base y derivada indica sobreescritura.
 - Object Pascal:
 - la clase derivada debe indicar que sobreescribe un método.
 - C#, Delphi Pascal:
 - exigen que tanto la clase base como la derivada lo indiquen.



- **Es importante distinguir entre Sobreescritura, Shadowing y Redefinición (vistos en Sobrecarga)**
 - **Sobreescritura:** la signatura de tipo para el mensaje es la misma en clase base y derivada, pero el método se enlaza con la llamada en tiempo de ejecución (en C++ el método se declara como **virtual** en la clase padre).
 - **Shadowing:** la signatura de tipo para el mensaje es la misma en clase base y derivada, pero el método se enlaza en tiempo de compilación (en función del tipo de la variable receptora).
 - **Redefinición:** La clase derivada define un método con el mismo nombre que en la clase base y con **distinta signatura de tipos**.



1. Motivación y conceptos previos
 - Signatura
 - Ámbito
 - Sistema de tipos
2. Polimorfismo y reutilización
 - Definición
 - Tipos de polimorfismo
3. Sobrecarga
 - Sobrecarga basada en ámbito
 - Sobrecarga basada en signatura de tipo
 - Alternativas a la sobrecarga
4. Polimorfismo en jerarquías de herencia
 - Redefinición
 - Shadowing
 - Sobrescritura
5. Variables polimórficas
 - La variable receptora
 - Downcasting
 - Polimorfismo puro
6. Genericidad
 - Funciones genéricas en C++
 - Plantillas de clase en C++
 - Herencia en clases genéricas



- Una variable polimórfica es aquella que puede referenciar más de un tipo de objeto
 - Puede mantener valores de distintos tipos en distintos momentos de ejecución del programa.
- En un lenguaje débilmente tipado todas las variables son potencialmente polimórficas
- En un lenguaje fuertemente tipado la variable polimórfica es la materialización del principio de sustitución.
 - En C++: punteros o referencias

Variables Polimórficas. Tipos



- **Variables polimórficas simples**

- `Barco *b[10];` //array de punteros a Barco que en realidad //contiene punteros a las distintas clases derivadas

- **Variable polimórfica como receptor de mensaje**

- **this**: en cada clase representa un objeto de un tipo distinto.
- Especialmente potente cuando se combina con sobreescritura.

```
class Figura {
    public:
        virtual void pintar(ostream&)=0;
        void imprimirEnCout() {
            this->pintar(cout);
            // this será un circulo, un rectangulo, etc...
        }
    ... };
```



- **Downcasting** (polimorfismo inverso):
 - Proceso de 'deshacer' el ppio. de sustitución. Puede dar problemas.
- **Downcasting estático** (en tiempo de compilación)

```
Base* obj = new Derivada();  
// El objeto se manipula mediante la interfaz de Base
```

```
Derivada* obj2 = (Derivada*) obj; // Downcasting estático  
// Ahora puedo manipularlo usando la interfaz de Derivada
```

- Pero... ¿y si obj no apunta a un objeto de tipo Derivada?
 - El resultado es impredecible si uso la interfaz de la clase Derivada...



■ Downcasting dinámico

- Downcasting que comprueba el tipo de un objeto en tiempo de ejecución.

```
Base* obj = new ...;
Derivada *obj2;
obj2 = dynamic_cast<Derivada *>(obj);
if (obj2 != NULL) {
    //OK el objeto apuntado es de tipo compatible
    con Derivada. Puedo usar la interfaz de Derivada
    aquí para manipularlo.
} else {
    // El puntero no se puede convertir a Derivada*
}
```



- Variable polimórfica utilizada como argumento de un método: **polimorfismo puro** o método polimórfico.
 - Un solo método puede ser utilizado con un número potencialmente ilimitado de tipos distintos de argumento.

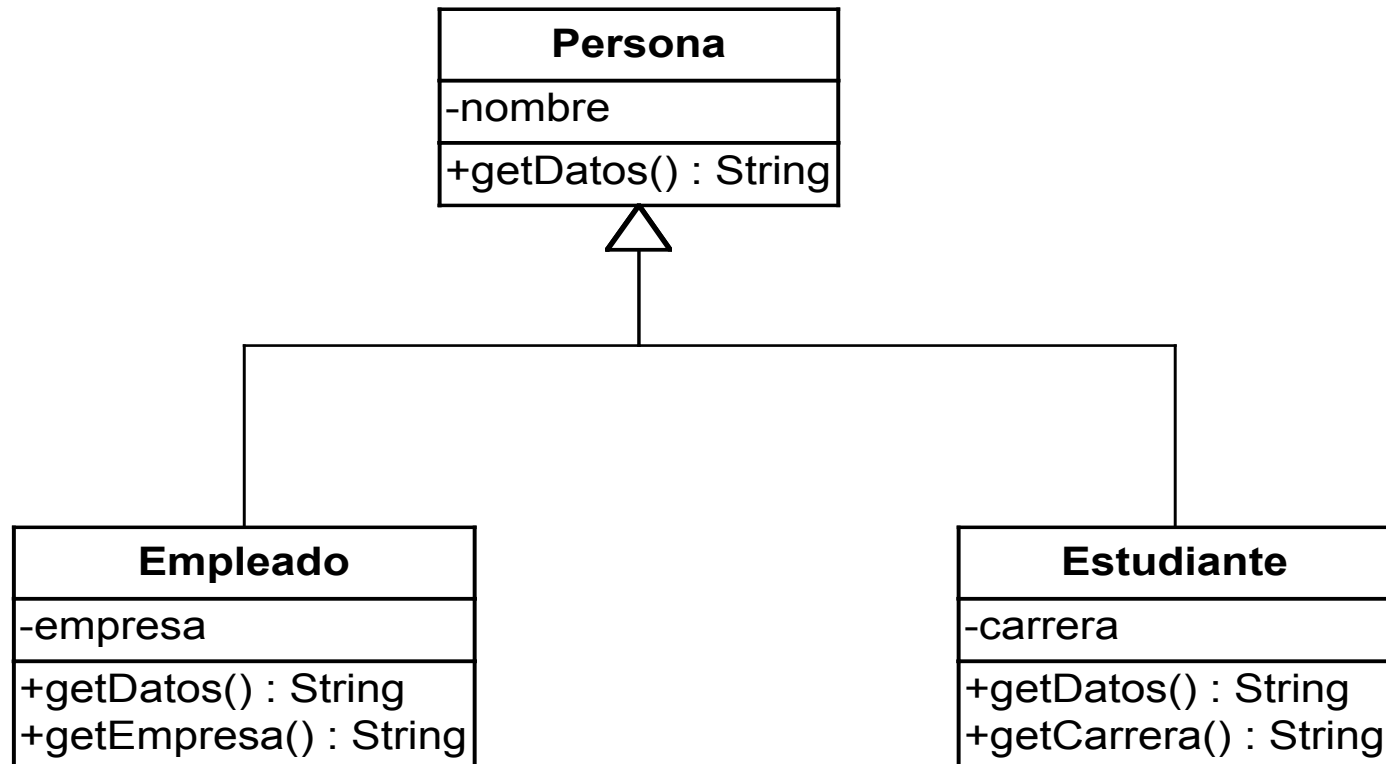
- Ejemplo de polimorfismo puro

```
class Base { ... };
class Derivada1 : Base { ... };
class Derivada2 : Base { ... };

void f(Base* obj) {
    // Aquí puedo usar sólo la interfaz de Base para manipular obj
    // Pero obj puede ser de tipo Base, Derivada1, Derivada2,...
}

int main() {
    Derivada1* objeto = new Derivada1();
    f(objeto); ...
}
```

Ejemplo: Uso de polimorfismo y jerarquía de tipos



Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
class Persona {  
    public:  
        Persona(string n)    {nombre=n;};  
        ~Persona();  
        string getDatos() {return (nombre);};  
    ...  
    private:  
        string nombre;  
};
```

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
class Empleado: public Persona {
public:
    Empleado(string n,string e): Persona(n){empresa=e;};
    ~Empleado(){};
    string getDatos(){
        return Persona::getDatos()+"trabaja en " + empresa);
    };
    ...
private:
    string empresa;
};

class Estudiante: public Persona {
public:
    Estudiante(string n,string c): Persona (n){carrera=c;};
    ~Estudiante(){};
    string getDatos(){
        return(Persona::getDatos() + " estudia " + carrera);
    };
    ...
private:
    string carrera;
};
```



Refinamiento



Refinamiento

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
int main() {  
  
    Empleado empleadoDesc("Carlos", "Lavandería");  
    Persona pers("Juan");  
    empleadoDesc=pers;  
    cout<<empleadoDesc.getDatos()<<endl;  
  
}
```



¿Qué salida dará este programa?

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
int main() {  
    int i;  
    Empleado *empleadoDesc=NULL;  
    Estudiante *estudianteDesc= NULL;  
    Persona *pers=new Persona("José");  
    empleadoDesc=pers;  
    estudianteDesc=pers;  
    cout<<empleadoDesc->getDatos()<<endl;  
    cout<<estudianteDesc->getDatos()<<endl;  
    estudianteDesc=NULL;  
    empleadoDesc=NULL;  
    delete (pers);  
    pers=NULL;  
}
```

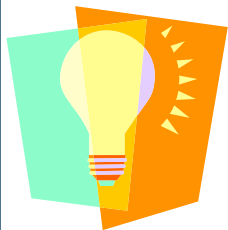


¿Qué salida dará este programa?
¿Se produce un enlazado estático o dinámico?

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
int main() {
    Empleado uno("Carlos", "lavanderia");
    Estudiante dos("Juan", "empresariales");
    Persona desc("desconocida");
    desc=uno; //le asigno empleado
    cout << desc.getDatos() << endl;
    return (0);
}
```



¿Qué salida dará este programa?
¿Se produce un enlazado estático o dinámico?

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
int main() {
    Empleado *emp= new Empleado("Carlos",
        "lavanderia");
    Estudiante *est= new Estudiante("Juan",
        "empresariales");
    Persona *pers;
    pers = emp;
    cout<<emp->getDatos()<<endl;
    cout<<est->getDatos()<<endl;
    cout << pers->getDatos() << endl;
    //Aquí habría que liberar memoria: cómo?
}
```



¿Qué salida dará este programa?
¿Se produce un enlazado estático o dinámico?

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
class Persona {  
    public:  
        Persona(string n)    {nombre=n;};  
        ~Persona();  
        virtual string getDatos() {  
            return (nombre);  
        };  
        ...  
    private:  
        string nombre;  
};
```

Puede ser resuelto
dinámicamente

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
int main() {
    Empleado uno("Carlos", "lavanderia");
    Estudiante dos("Juan", "empresariales");
    Persona desc("desconocida");
    desc=uno; //un empleado es una persona
    cout << desc.getDatos() << endl;
}
```



¿Qué salida dará este programa?
¿Se produce un enlazado estático o dinámico?

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
int main() {
    Empleado *uno= new Empleado("Carlos", "lavanderia");
    Estudiante *dos= new Estudiante("Juan",
    "empresariales");
    Persona *desc;
    desc = uno;
    cout << uno->getDatos () << endl;
    cout << dos->getDatos () << endl;
    cout << desc->getDatos () << endl;
    desc=NULL;
    delete (uno); uno=NULL;
    delete (dos); dos=NULL;
}
```



¿Qué salida dará este programa?
¿Se produce un enlazado estático o dinámico?

Ejemplo: Uso de polimorfismo y jerarquía de tipos



```
int main1() {
    Empleado *uno= new Empleado("Carlos", "lavanderia");
    Persona *desc = uno;
    cout << desc->getEmpresa() << endl;
    desc=NULL; delete(uno); uno=NULL;
    return (0);
}
```

```
int main2() {
    Persona *desc = new Persona("Carlos");
    Empleado *empPtr=(Empleado*) desc;
    cout<<empPtr->getEmpresa()<<endl;
    empPtr=NULL; delete(desc); desc=NULL;
    return (0);
}
```



**¿Qué salida dará main1? ¿Y main2?
¿Se produce un enlazado estático o dinámico?**

Polimorfismo



Consecuencias para la definición de destructores en jerarquías

```
int main() {  
    Empleado *uno= new Empleado("Carlos", "lavanderia");  
    Estudiante *dos= new Estudiante("Juan",  
    "empresariales");  
    Persona *desc;  
    desc = uno;  
    cout<<uno->getDatos()<<endl;  
    cout<<dos->getDatos()<<endl;  
    cout << desc->getDatos() << endl;  
    delete desc; desc=NULL;  
    delete dos; dos=NULL;  
    uno=NULL;  
}
```



¿Se destruirán correctamente los objetos?

Polimorfismo

Implementación interna en C++



- Las funciones virtuales son algo menos eficientes que las funciones normales.
 - Cada clase con funciones virtuales dispone de un vector de punteros llamado *v_table*. Cada puntero corresponde a una función virtual, y apunta a su implementación más conveniente (la de la propia clase o, en caso de no existir, la del ancestro más cercano que la tenga definida)
 - Cada objeto de la clase tiene un puntero oculto a esa *v_table*.

Polimorfismo



Ventajas

- El polimorfismo hace posible que un usuario pueda añadir nuevas clases a una jerarquía sin modificar o recompilar el código escrito en términos de la clase base.
- Permite programar a nivel de clase base utilizando objetos de clases derivadas (posiblemente no definidas aún): Técnica base de las librerías/frameworks



1. Motivación y conceptos previos
 - Signatura
 - Ámbito
 - Sistema de tipos
2. Polimorfismo y reutilización
 - Definición
 - Tipos de polimorfismo
3. Sobrecarga
 - Sobrecarga basada en ámbito
 - Sobrecarga basada en signatura de tipo
 - Alternativas a la sobrecarga
4. Polimorfismo en jerarquías de herencia
 - Redefinición
 - Shadowing
 - Sobrescritura
5. Variables polimórficas
 - La variable receptora
 - Downcasting
 - Polimorfismo puro
6. Genericidad
 - Funciones genéricas en C++
 - Plantillas de clase en C++
 - Herencia en clases genéricas

Genericidad

Motivación



- La genericidad es otro tipo de polimorfismo
- Para ilustrar la idea de la genericidad se propone un ejemplo:
 - Suponed que queremos implementar una función *máximo*, donde los parámetros pueden ser de distinto tipo (int, double, float)

Genericidad

Motivación



- Solución:

```
double maximo(double a, double b){
    if (a > b)
        return a;
    else
        return b;
};
int main (void){
    int y,z;
    float b,c;
    double t,u;
    double s= maximo(t,u);
    double a= maximo((double)b,(double)c);
    double x= maximo((double)y,(double)z);
}
```


Genericidad

Motivación



- Ahora suponed que creamos una clase Cuenta, y también queremos poder comparar objetos de esa clase con la función `maximo()`.

Genericidad

Motivación



- 1º Sobrecargar el operador > para TCuenta
- 2º Sobrecargar la función máximo para TCuenta

```
TCuenta maximo(TCuenta a, TCuenta b){
    if (a>b)
        return a;
    else
        return b;
}
void main (void){
    double s,t,u;
    TCuenta T1,T2,T3;
    s= maximo(t,u);
    T1= maximo(T2,T3);
}
```

Conclusión: Tenemos dos funciones máximo definidas, una para double y otra para TCuenta, pero el código es el mismo. La única diferencia son los parámetros de la función y el valor devuelto por ésta.



- ¿Y si no sabemos a priori los tipos que otros van a crear para ser comparados?
 - Necesitaríamos una función genérica que nos sirviera para cualquier tipo sin necesidad de tener la función duplicada.

Genericidad

DEFINICION



- *Propiedad que permite definir una clase o una función sin tener que especificar el tipo de todos o algunos de sus miembros o argumentos.*
 - Su utilidad principal es la de agrupar variables cuyo tipo base no está predeterminado (p. ej., listas, colas, pilas etc. de objetos genéricos: STL).
 - Es el usuario el que indica el tipo de la variable cuando crea un objeto de esa clase.
 - En C++ esta característica apareció a finales de los 80. En Java, existe desde la versión 1.5.

Genericidad en C++:

Templates



- Utilizan la palabra clave ***template*** (plantilla)
- Dos tipos de plantillas:
 - **Plantillas de funciones (o funciones genéricas)**: son útiles para implementar funciones que aceptan argumentos de tipo arbitrario.
 - **Plantillas de clase (o clases genéricas)**: su utilidad principal consiste en agrupar variables cuyo tipo no está predeterminado (*clases contenedoras*)

Genericidad

Ejemplo de **función genérica**

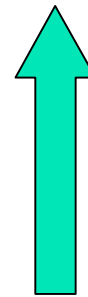


Un argumento genérico

```
template <class T>
T maximo(T a, T b){
    if (a > b)
        return a;
    else
        return b;
};
```

```
int main (void){

TCuenta a,b,c;
int x,y,z;
z= maximo(x,y); //OK
c= maximo(a,y); //INCORRECTO
c= maximo(a,b); //?
}
```



¿Qué tendría que hacer para, aplicando la función *máximo*, poder obtener la cuenta con más saldo?



Más de un argumento genérico

```
template <class T1, class T2>
T1 sumar (T1 a, T2 b){
    return (a+b);
};
```

```
int main (void){

    int euno=1;
    int edos=2;
    char cuno='a';
    char cdos='d';

    cout<<sumar(euno,cuno);
    cout<<sumar(cuno,euno);

    TCuenta c;
    cout<<sumar(euno,c);

}
```

Genericidad

Ejemplo **Clase Genérica en C++**



- A continuación se plantea un ejemplo de una clase genérica *vector*, este vector va a contener elementos de tipo genérico, no se conocen a priori.

Genericidad

Ejemplo **Clase Genérica en C++**



```
template <class T>
class vector {
    private:
        T *v;
        int tam;
    public:
        vector(int);
        T& operator[](int);
};

void main (void){
    vector<int> vi(10);
    vector<float> vf(20);
    vector<double> vd(50);
}
```

Genericidad



Ejemplo **definición de métodos** de una clase genérica

```
template <class T> // obligatorio!  
vector<T>::vector(int size) {  
    tam=size;  
    v=new T[size];  
};
```

```
template <class T>  
T& vector<T>::operator[](int i) {  
    return(v[i]);  
};
```

Genericidad

Ejemplo de uso de una Clase Genérica



- Creación de objeto:

```
vector<double> d(30);
```

```
vector<char> *p=new vector<char>(50);
```

Cada vez que se instancia una clase o método genérico, el compilador crea una nueva clase o método, reemplazando los argumentos genéricos por los que indica el programador.

Genericidad

Ejemplo: Pila de Objetos Genérica



- El siguiente ejemplo es una pila que contendrá objetos de cualquier tipo, para ello la vamos a definir como una clase genérica o plantilla. En esta pila podremos introducir nuevos elementos e imprimir el contenido de la misma.

Genericidad

Ejemplo: Pila de Objetos Genérica



```
template <class T>
class Pila{
    public:
        Pila(int nelem=10);
        void apilar (T);
        void imprimir();
        ~Pila();

    private:
        int nelementos;
        T* info;
        int cima;
        static const int limite=30;
};
```

Genericidad

Ejemplo: Pila de Objetos Genérica



```
template <class T>
Pila<T>::Pila(int nelem){
    if (nelem<=limite){
        nelementos=nelem;
    }
    else {
        nelementos=limite;
    }
    info=new T[nelementos];
    cima=0;
};
```

Genericidad

Ejemplo: Pila de Objetos Genérica



```
template <class T>
void Pila<T>::apilar(T elem){
    if (cima<nelementos)
        info[cima++]=elem;
}
```

```
template <class T>
void Pila<T>::imprimir(){
    for (int i=0;i<cima;i++)
        cout<<info[i]<<endl;
}
```

```
template <class T>
Pila<T>::~~Pila(){
    delete [] info; info=NULL;
}
```

Genericidad

Ejemplo: Pila de Objetos Genérica



```
#include "TPila.h"  
#include "TCuenta.h"
```

```
int main(){
```

```
    Pila <TCuenta> pCuentas(6);  
    TCuenta c1("Cristina",20000,5);  
    TCuenta c2("Antonio",10000,3);  
    pCuentas.apilar(c1);  
    pCuentas.apilar(c2);  
    pCuentas.imprimir();
```

```
    Pila <char> pchar(8);  
        pchar.apilar('a');  
        pchar.apilar('b');  
        pchar.imprimir();  
} //end main
```



De manera análoga, plantead una lista de objetos genérica.



- Se pueden **derivar clases genéricas** de otras clases genéricas:

Clase derivada genérica:

```
template <class T>
class doblePila: public Pila<T>
{
    public:
        void apilar2(T a, T b);
};
```

- La clase `doblePila` es a su vez genérica:

```
doblePila<float> dp(10);
```



- Se pueden **derivar clases no genéricas** de una genérica:

Clase derivada no genérica:

```
class monton: public Pila<int>
{
    public:
        void amontonar(int a);
};
```

- 'monton' es una clase normal. No tiene ningún parámetro genérico.
- En C++, no existe relación alguna entre dos clases generadas desde la misma clase genérica.

Genericidad

Constantes en plantillas



- Los argumentos de una plantilla no están restringidos a ser clases definidas por el usuario, también pueden ser tipos de datos existentes.
- Los valores de estos argumentos se convierten en constantes en tiempo de compilación para una instanciación particular de la plantilla.
- También se pueden usar valores por defecto para estos argumentos.

Genericidad

Ejemplo Constantes en plantillas



```
template <class T, int size=100>
class Array{
    public:
    T& operator[](int index) ;
    int length() const { return size; }
    private:
    T array[size];
};

int main(){
    Array <Cuenta,30> t;
    int i=10;
    Array <Cuenta,i> t2; // ERROR, no constante.
}
```

Tema 4. Polimorfismo

Bibliografía



- Cachero et. al.
 - ***Introducción a la programación orientada a Objetos***
 - Capítulo 4
- T. Budd.
 - ***An Introduction to Object-oriented Programming, 3rd ed.***
 - Cap. 11,12,14-18; cap. 9: caso de estudio en C#
- Scott Meyers
 - ***Effective C++. Third Edition***
 - Cap. 6 y 7: Algunas secciones sobre polimorfismo en herencia y genericidad