Parcial2POO-wuolah.pdf

ersosio

Programación Orientada a Objetos

2º Grado en Ingeniería Informática

Escuela Superior de Ingeniería
Universidad de Cádiz

Parcial 2 - POO

ersosio

Junio 2021

Contents

yo elijo cerveza SIN

Sea cual sea el vehículo que conduces, elige cerveza SIN.

WWW.CONDUCCIONRESPONSABLECERVEZASIN.COM

Cervezas de España 100 años más

UNA GRAN CERVEZA.
UNA GRAN RESPONSABILIDAD.

DGT Dirección General de Tráfico

1 Asociaciones

1.1 Asociación: 1 a 1

Para crear una realción 1 a 1 entre dos relaciones declaramos un atributo privado en cada clase siendo éste un puntero a la otra clase relacionada, por ejemplo, tenemos dos clases, Persona y Asignatura, si necesitamos crear una asociación 1 a 1 entre ambas, entonces, dentro de los atributos privados de la clase Persona, declaramos un atributo del tipo `Asignatura* asignatura`, de la misma manera, en la clase Asignatura, el nuevo atributo privado sería `Persona* persona`. De esta manera realizamos una asociación 1 a 1 entre las dos clases.

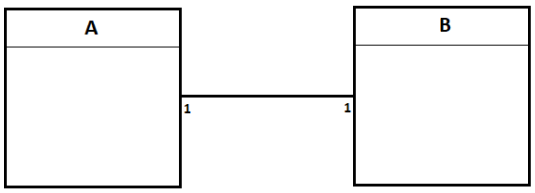


Figure 1: Ejemplo de asociación 1 a 1

Un esquema general de esta asociación sería:

```
class A {
public:
    // ...
    void asocia(B&); // enlaza con objeto B
    B& asocia() const; // objeto B enlazado
    private:
    // ...
    B* b; // enlace con objeto B
};

class B {
public:
    // ...
    void asocia(A&); // enlaza con objeto A
    A& asocia() const; // objeto A enlazado
private:
    // ...
    A* a; // enlace con objeto A
};
```

1.2 Asociación: varios a varios

1.2.1 Contenedores Asociativos

Para este tipo de asociación necesitamos la ayuda de la STL con la inclusión de contenedores asociativos, por lo que un repaso no vendría mal. Los contenedores que hemos visto en teoría son 4, set, multiset, map y multimap, éstos los podemos dividir en dos grupos, siendo set y multiset un grupo y map y multimap otro grupo.

En primer lugar veremos un repaso del set y multiset. El contenedor set que almacena un único elemento siguiendo un orden específico, este elemento es único, es decir, no habrá dos elementos con el mismo valor, además este valor identifica el elemento en el conjunto. A diferencia de set, multiset sí permite la inclusión de elementos con el mismo valor, la eliminación en ese conjunto es la eliminación del conjunto de elementos con el mismo valor, si por ejemplo, tenemos en el contenedor enteros de la siguiente manera: `1 3 4 5 5 6 9` y deseamos eliminar el elemnto de valor 5, pues el contenedor resultante sería: `1 3 4 6 9`. Las funciones que nos permite el contenedor set son las siguientes:

fx **Member functions**

(constructor)	Construct set (public member function)
(destructor)	Set destructor (public member function)
operator=	Copy container content (public member function)
Iterators:	
begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin C++11	Return const_iterator to beginning (public member function)
cend C++11	Return const_iterator to end (public member function)
crbegin C++11	Return const_reverse_iterator to reverse beginning (public member function)
crend C++11	Return const_reverse_iterator to reverse end (public member function)
Capacity:	
empty	Test whether container is empty (public member function)
size	Return container size (public member function)
max_size	Return maximum size (public member function)
Modifiers:	
insert	Insert element (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace C++11	Construct and insert element (public member function)
emplace_hint C++11	Construct and insert element with hint (public member function)
Observers:	
key_comp	Return comparison object (public member function)
value_comp	Return comparison object (public member function)
Operations:	
find	Get iterator to element (public member function)
count	Count elements with a specific value (public member function)
lower_bound	Return iterator to lower bound (public member function)
upper_bound	Return iterator to upper bound (public member function)
equal_range	Get range of equal elements (public member function)
Allocator:	
get_allocator	Get allocator (public member function)

Figure 2: Funciones miembro del contenedor *Set*

Por otro lado tenemos map y multimap, a diferencia de los anteriores contenedores, éstos almacenan pares de valores, clave y valor, usamos el valor clave para la identificación de elementos de manera que no hay dos claves repetidas, internamente, map ordena los elementos de menor a mayor. Al igual que multiset, multimap soporta varias claves duplicadas.Las funciones miembros del contenedor map son:

Member functions

(constructor)	Construct map (public member function)
(destructor)	Map destructor (public member function)
operator=	Copy container content (public member function)
Iterators:	
begin	Return iterator to beginning (public member function)
end	Return iterator to end (public member function)
rbegin	Return reverse iterator to reverse beginning (public member function)
rend	Return reverse iterator to reverse end (public member function)
cbegin (C++11)	Return const_iterator to beginning (public member function)
cend (C++11)	Return const_iterator to end (public member function)
crbegin (C++11)	Return const_reverse_iterator to reverse beginning (public member function)
crend (C++11)	Return const_reverse_iterator to reverse end (public member function)
Capacity:	
empty	Test whether container is empty (public member function)
size	Return container size (public member function)
max_size	Return maximum size (public member function)
Element access:	
operator[]	Access element (public member function)
at (C++11)	Access element (public member function)
Modifiers:	
insert	Insert elements (public member function)
erase	Erase elements (public member function)
swap	Swap content (public member function)
clear	Clear content (public member function)
emplace (C++11)	Construct and insert element (public member function)
emplace_hint (C++11)	Construct and insert element with hint (public member function)
Observers:	
key_comp	Return key comparison object (public member function)
value_comp	Return value comparison object (public member function)
Operations:	
find	Get iterator to element (public member function)
count	Count elements with a specific key (public member function)
lower_bound	Return iterator to lower bound (public member function)
upper_bound	Return iterator to upper bound (public member function)
equal_range	Get range of equal elements (public member function)
Allocator:	
get_allocator	Get allocator (public member function)

Figure 3: Funciones miembro del contenedor *Map*

Un ejemplo de **Set** y **Multiset**:

```
#include <iostream>
#include <fstream>
#include <iterator>
#include <set>
using namespace std; // para set y multiset
int main()
{
    set<int> valores;
    ifstream archi("datos.txt");
    istream_iterator<int> i(archi);
    istream_iterator<int> f;
    while( i != f) valores.insert( i++ );
    archi.close();
    copy( valores.begin(), valores.end(), // mostrar los
        elementos por pantalla
    ostream_iterator<int>( cout, " " ) );
    set<int>::iterator p = valores.find( 0 ); // buscar un
        elemento en la estructura
    if( p != valores.end() ) cout << "\nEl valor 0 esta en el
        archivo\n";
    else cout << "\nEl valor 0 no esta en el archivo\n";
    multiset<int> multivalores; // comparar con multiset
    archi.open("datos.txt");
    istream_iterator<int> j(archi);
    while( j != f) multivalores.insert( j++ );
    archi.close();
    copy( multivalores.begin(), multivalores.end(),
        ostream_iterator<int>( cout, " " ) );
    cout << "\nEl valor 0 esta "<< multivalores.count( 0 ) << "
        veces en el archivo\n";
    multivalores.erase( 2 ); // eliminar un elemento
    copy( multivalores.begin(), multivalores.end(),
    ostream_iterator<int>( cout, " " ) );
    return 0;
}
```

Un ejemplo de **Map**:

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main()
{
    map<string, long> directory;
    directory["a"] = 1234678;
    directory["b"] = 9876543;
    directory["c"] = 3459876;
    string name = "a";
    if (directory.find(name) != directory.end())
        cout << "The phone number for" << name << " is " <<
            directory[name] << "\n";
    else
        cout << "Sorry, no listing for " << name << " \n";
    return 0;
}
```

1.2.2 Asociación

Esta vez, para obtener una asociación varios a varios entre dos clases, debemos usar el contenedor set, para obtener un conjunto de punteros a objetos de la otra clase, para ello declaramos en la parte pública de las clases `typedef set<Nombre_clase*> Nombre_conjunto` y en la parte privada declaramos un objeto de tipo *Nombre_conjunto*.

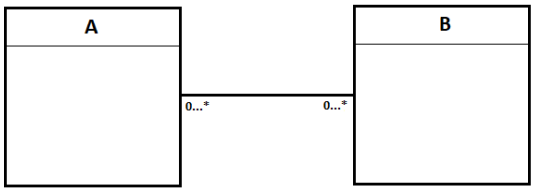


Figure 4: Ejemplo de asociación varios a varios

Un esquema general de esta asociación sería:

```
class A {
public:
    typedef set<B*> Bs;
    // ...
    void asocia(B&); // enlaza con objeto B
    const Bs& asocia() const; // objetos B enlazados
    private:
    // ...
    Bs bs; // enlaces con objetos B
};

class B {
public:
    typedef set<A*> As;
    // ...
    void asocia(A&); // enlaza con objeto A
    const As& asocia() const; // objetos A enlazados
private:
    // ...
    As as; // enlace con objetos A
};
```

1.3 Agregación y Composición

Para estos dos tipos de asociación realizamos lo siguiente:

- Agregación: Esta asociación la implementamos de la misma manera que venimos haciendo, con una referencia a la otra clase o con un contenedor.
- Composición: Puesto que la existencia de un objeto componente no tiene sentido independiente de la existencia del agregado, los enlaces en un agregado/compuesto se pueden implementar mediante atributos que sean directamente del tipo de los componentes, en lugar de que sean punteros o referencias a dichos componentes. Es decir, declaramos un objeto o un conjunto de la clase componente en la clase compuesta.

1.4 Asociación calificada

En la parte de diseño de clases, podemos utilizar las asosciaciones calificadas con el objetivo de reducir la multiplicidad de esa relación, para ello usamos cualificadores de la siguiente forma:

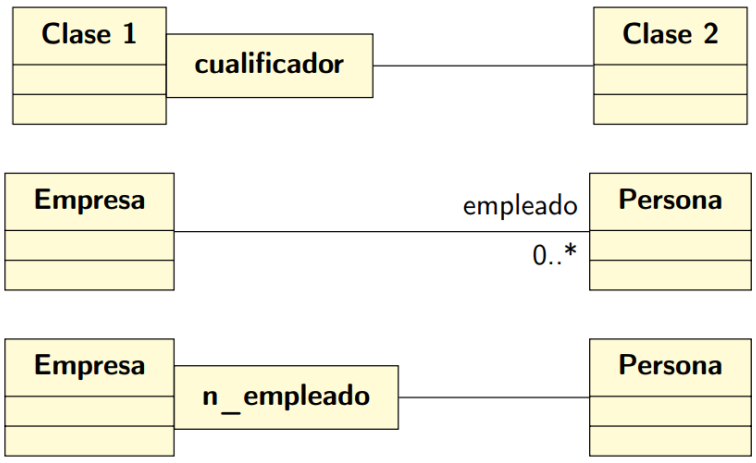
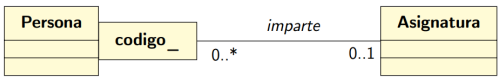


Figure 5: Ejemplo de asociación calificada

Para esto usamos los contenedores map y multimap, el primero se usaría en la figura anterior ya que la relación entre ambas clases sería una asociación 1 a 1, mientras que, por ejemplo, en vez de almacenar como clave `n_empleado`

usamos un codigo del área donde trabaja la persona, habrá 0 o varias personas trabajando en ese sector, por lo que la multiplicidad de la relación seria de 1 a varios y ya en la clase empresa usaríamos multimap, ya que varias personas comparten la clave, en este ejemplo el código del área de la empresa.

Ejemplo con map:




```
class Persona {
public:
    typedef std::map<string, Asignatura*> AsignaturasCalificadas
        ;
    // ...
    void imparte(Asignatura& asignatura); // enlaza con
        asignatura
    const AsignaturasCalificadas& imparte() const; //
        asignaturas enlazadas ordenadas por codigo
private:
    AsignaturasCalificadas asignaturas; // enlaces calificados
        con codigo de asignatura
};
```

```
class Asignatura {
public:
    typedef std::set<Persona*> Personas;
    // ...
    string codigo() const;
    void impartida(Persona& persona); // enlaza con persona
    const Personas& impartida() const; // personas enlazadas
private:
    string codigo_;
    // ...
    Personas personas; // enlaces con personas
};
```

1.5 Atributos de enlace

1.5.1 Atributos de enlace en asociaciones 1 a varios

Ene ste tipo de asociaciones con atributos de enlace por medio podemos permitirnos el lujo de usar un truco que consiste en declarar ese atributo de enlace en la clase con la multiplicidad *0...**, partiendo de la Figura 6 obteniendo la Figura 8 usando ese truco.

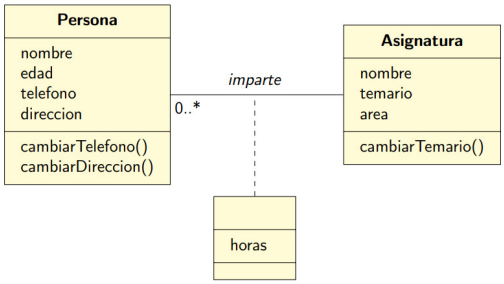


Figure 6: Asociación antes de usar el truco

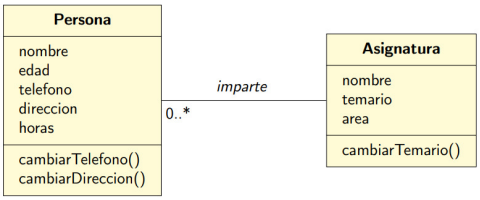


Figure 7: Asociación después de usar el truco

Esto también lo podemos usar en asociaciones 1 a 1 dándonos igual en qué clase colocar ela tributo de enlace.

1.5.2 Atributos de enlace en asociaciones varios a varios

En este tipo de asociaciones no podemos usar ese truco, por lo que tendremos que realizar un diccionario en cada clase, donde la clave será el enlace a asignatura y el elemento será el tipo del atributo de enlace. Veamos un ejemplo:

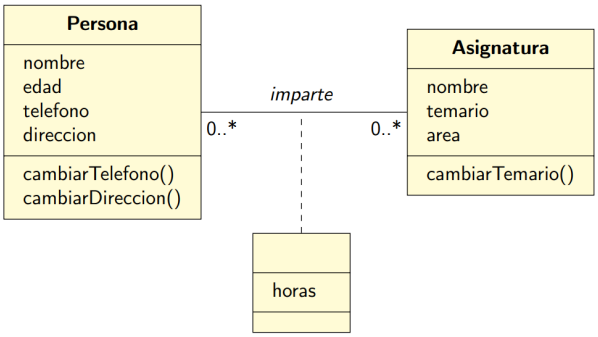


Figure 8: Asociación después de usar el truco

Para ello, crearemos un *map* dentro de Persona del tipo `map<Asignatura*, int>` y otro dentro de Asignatura del tipo `map<Persona*, int>` de la siguiente forma:

```
class Persona {
public:
    typedef map<Asignatura*, int> Asignaturas;
    // ...
    void imparte(Asignatura& asignatura, int horas);
    const Asignaturas& imparte() const;
private:
    // ...
    Asignaturas asignaturas; // enlaces y sus atributos
};
```

```
class Asignatura {
public:
    typedef map<Persona*, int> Personas;
    // ...
    void impartida(Persona& persona, int horas);
    const Personas& impartida() const;
private:
    // ...
    Personas personas; // enlaces y sus atributos
};
```

Por lo que un esquema general sería:

```
class A {
public:
    typedef map<B*, C> Bs;
    // ...
    void asocia(B&, C); // enlaza con objeto B y atributo C
    const Bs& asocia() const; // objetos B enlazados
private:
    // ...
    Bs bs; // enlaces y sus atributos
};
```

```
class B {
public:
    typedef map<A*, C> As;
    // ...
    void asocia( A&, C); // enlaza con objeto A y atributo C
    const As& asocia() const; // objetos A enlazados
private:
    // ...
    As as; // enlaces y sus atributos
};
```

1.6 Clases de asociación

Para esto debemos crear una nueva clase que encapsulará los atributos de enlace y otra clase que representará la clase de asociación que tendrá todos los miembros necesarios para almacenar los enlaces de esa asociación.

Pongamos un ejemplo, tenemos la asociación en la Figura 9 que tiene una asociación bidireccional con multiplicidad 1 a varios, donde hay una clase de asociación llamada AEE.

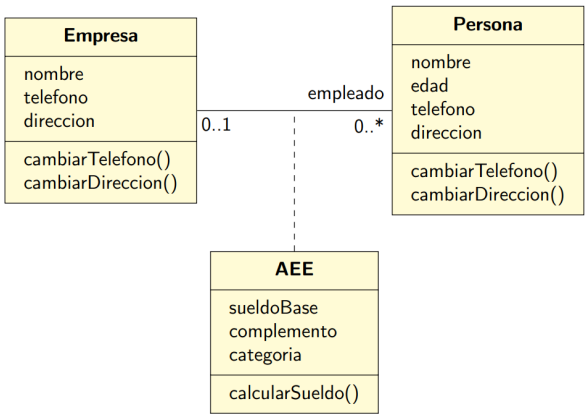


Figure 9: Ejemplo de clase de asociación

Clase Empresa:

```
class Empresa{
public:
    void cambiarTelefono(int telefono);
    void cambiarDireccion(string direccion);
private:
    string nombre;
    int telefono;
    string direccion;
}
```

Clase Persona:

```
class Persona{
public:
    void cambiarTelefono(int telefono);
    void cambiarDireccion(string direccion);
private:
    string nombre;
    int edad;
    int telefono;
    string direccion;
}
```

Clase de asociación:

```
//Clase donde se almacenan todos los atributos de enlace
class Salario{
public:
    void calcularSueldo() const;
private:
    double sueldoBase, complemento;
    int categoria;
}

//Clase con los m todos para enlazar las clases
class AEE{
public:
    void asocia(Empresa& e, Persona& p);
    void asocia(Persona& p, Empresa& e);
    const std::map<Persona*, Salario*>* asocia(Empresa& y) const
        ;   // A partir de una Empresa obtenemos el conjunto
        ordenado de objetos tipo persona junto a su salario
    const std::pair<Empresa*, Salario*>* asocia(Persona& x)
        const;  // A partir de una Persona obtenemos el conjunto
         ordenado de objetos tipo Empresa junto a su salario
private:
    typedef std::map<Empresa*, std::map<Persona*, Salario*> > AD
        ;
    typedef std::map<Persona*, std::pair<Empresa*, Salario*> >
        AI;
    AD empresa-empleado;
    AI empleado-empresa;
};
```

2 Generalización y herencia

2.1 Introducción

La generalización de clases consiste en que, por ejemplo tenemos 10 clases que comparten unos atributos, pues en vez de repetir varias veces ese atributo entre las diferentes clases pues generalizamos ese atributo que comparten esas clases en una propia. La especialización sería el proceso inverso, es decir, de una clase, por ejemplo, empleado, salen más variantes con distintos tipos que dentro de éstos hay diferentes atributos.

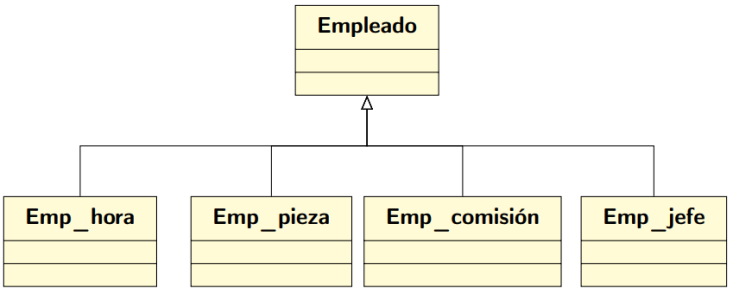


Figure 10: Ejemplo de Generalización

2.2 Herencia

Bien, una vez aprendidos estos conceptos, ya podemos desarrollar el concepto de herencia y ver su implementación. En el punto anterior se dijo que atributos iguales se generalizaban en una clase, pues estos atributos se heredan a las subclases, por ejemplo, en la Figura 11, si en la clase `Empleado` hubiera un atributo llamado *nombre*, todas las subclases, es decir, `Emp_hora`, `Emp_pieza`, `Emp_comisión` y `Emp_jefe`, heredarían ese atributo, teniendo un miembro nuevo en cada clase.

Para ello usamos la siguiente plantilla:

```
class clase-derivada: [accesibilidad] clase-base
{
// declaraciones de miembros
};
```

Se heredan todos los miembros de la clase excepto los constructores, el destructor y las operaciones de asignación.

Hay tres tipos de accesibilidad: *public*, *protected* y *private*. La utilización de

estos tipos de accesibilidad se comportan de manera distinta, como vemos en la siguiente tabla:

Accesibilidad	un miembro ... de la clase base	pasa a ser ... en la derivada
`public` (por defecto en `struct`)	público protegido privado	público protegido inaccesible
`protected`	público protegido privado	protegido protegido inaccesible
`private` (por defecto en `class`)	públic protegido privado	privado privado inaccesible

Vamos fila por fila explicando cada situación:

- Accesibilidad `public`: Los miembros declarados como públicos en la clase base, pasan a ser miembros públicos de la clase derivada, pasa exactamente lo mismo para los miembros protegidos, pero los miembros privados de la clase base se vuelven inaccesibles para la clase derivada, por lo que debemos acceder a ellos mediante los métodos públicos que devuelven el valor de esos miembros privados.
- Accesibilidad `protected`: Tanto los miembros públicos como los protegidos de la clase base pasan a ser miembros protegidos en la clase derivada, con los atributos privados de la clase base pasa igual que en el caso anterior.
- Accesibilidad `private`: Pasa igual que en el caso anterior, pero en vez de estar protegidos, ahora son privados.

2.2.1 Herencia múltiple

Explicamos este concepto mejor con un ejemplo.

Tenemos el siguiente diagrama de clases:

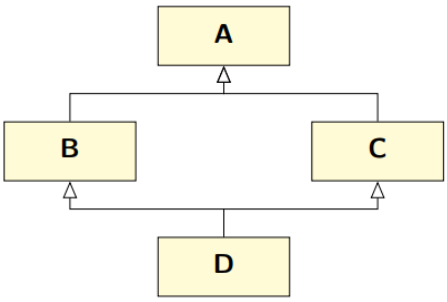


Figure 11: Ejemplo de Herencia Múltiple

La implementación de este diagrama se haría de la siguiente manera:

```
struct A { int a; };
struct B: A { int b; };
struct C: A { int c; };
struct D: B, C { int d; };
```

Tras la ejecución de estas cuatro líneas de código, el `struct D` tendrá 5 atributos, un entero d de la clase, dos enteros a y b provenientes de la clase B y dos enteros a y c provenientes de la clase C. Ahora bien, si queremos acceder al atributo a de la clase D tenemos las siguientes opciones:

```
D d;
d.a = 0;
d.B::a = 0;
d.C::a = 0;
```

La primera asignación daría error de ambigüedad ya que no sabría a qué a asignarle el 0, recordemos que tenemos dos a, uno proveniente de B y otro de C. La segunda y asignación estaría bien ya que indicamos a qué a queremos asignarle el valor 0, la a de la clase B y de la clase C, respectivamente.

2.2.2 Herencia Virtual

La herencia virtual nos resuelve la duplicidad de miembros que puede aparecer utilizando herencia múltiple, por ejemplo, en el anterior apartado teniamos en la clase D 5 atributos, dos a, un b, un c y un d, pues usando virtual en la accesibilidad de las herencias de A en las clases B y C, tenemos un único atributo a.

```
struct A { int a; };
struct B: virtual A { int b; }; //Da igual poner public virtual
    que virtual public
struct C: virtual A { int c; };
struct D: B, C { int d; };  //La a que se hereda pues no se sabe
     de quien se hereda, depende del compilador

D d;
d.a = 0; // bien , d solo tiene un atributo a
```

3 Polimorfismo

En C++ existen tres tipos de polimorfismos:

- Polimorfismo de sobrecarga
- Polimorfismo en tiempo de ejecución
- Polimorfismo paramétrico

3.1 Polimorfismo de sobrecarga

Se resuelve en tiempo de compilación y consiste en utilizar el mismo nombre para un conjunto de funciones, las cuales se diferencian en el número, tipo y orden de parámetros que reciben, si devuelve una variable, ...

3.2 Polimorfismo en tiempo de ejecución

Para este tipo de polimorfismo debemos usar la palabra reservada `virtual` con métodos, el comportamiento de éste se hereda a las clases derivadas de la clase que lo usa.

En el caso de que la clase base tenga el método virtual y la clase derivada no lo tenga, esta última hereda el método y su comportamiento virtual, es decir, el implementado en la clase base.

```
class B {
public:
    virtual void mostrar() { std::cout<< i <<"dentro de B\n"; }
    int i;
};
class D: public B {};
```

En otro caso, si la subclase tiene un/varios método/s se puede acceder a ambos métodos, tanto los heredados como los de la subclase.

```
class B {
public:
    virtual void mostrar() { std::cout<< i <<" dentro de B\n"; }
    int i;
};
class D: public B {
public:
    void mostrar() { std::cout<< i <<" dentro de D\n"; }
};
int main() {
    B b, *pb = &b;
    D d;
    d.i = 1 + (b.i = 1); // b.i = 1, d.i = 1 + 1;
    pb->mostrar(); // B::mostrar()
    pb = &d;
    pb->mostrar(); // D::mostrar()
}
```

3.3 Polimorfismo paramétrico

Soportado en C++ mediante el uso de plantillas (templates), las cuales permiten usar tipos de datos como parámetros de la defínción de clases y funciones.
Proporciona el mecanismo para aplicar técnicas de programación genérica.
Una plantilla es una definición gráfica 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.
Las clases (o funciones) específicas las genera automáticamente el compilador cuando especializamos la plantilla al proporcionar los parámetros reales.

La declaración de una plantilla es `template <typename T>`
Ejemplos de plantillas:

- Plantilla con un único parámetro:

```
template <typename T> class C {
public:
    static int n; // especifico
    // ...
};
// ...
C<int> v1, v2; // v1.n y v2.n son el mismo atributo, C<int>::n
C<double> v3; // v3.n es C<double>::n
```

- Plantilla con dos parámetros:

```
template <typename T1, typename T2>
bool operator ==(const vector<T1>& a, const vector<T2>& b)
{
    const size_t n = a.size();
    if (n != b.size())
        return false;
    for (size_t i = 0; i < n; ++i)
        if (a[i] != b[i])
            return false;
    return true;
}
```

- Plantilla con dos parámetros, el tipo de dato y un tamaño n

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

// ...

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

a = b; // bien
c = a; // ERROR, se detecta en tiempo de compilaci n
```

Incluso si tiene uno o los dos parámetros omitidos

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

// ...

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