INTRODUCCIÓN A LA PROGRAMACIÓN

ANEXO 2

El paradigma orientado a objetos

INTRODUCCIÓN

1) Programación orientada a objetos (P.O.O.)

Un proyecto de software es complejo. Las GUI[1], acceso transparente a datos y capacidad de trabajo. En red, lo hacen más complejo aún. Para enfrentarse a esta complejidad, nace la POO.

2) ¿Qué es la POO?

Es una técnica o estilo de programación que utiliza objetos como bloque fundamental de construcción.

3) Elementos básicos de la POO

- ***Bloques***

[11] Interfaz gráfica del usuario (graphical user interface)

Son un conjunto complejo de datos (atributos) y funciones (métodos) que poseen una determinada estructura y forman parte de una organización. Los atributos definen el estado del objeto; los métodos, su comportamiento.

- ***Métodos***

Es un programa procedimental que está asociado a un objeto determinado y cuya ejecución solo puede desencadenarse a través del mensaje correspondiente.

- ***Mensajes***

Es una petición de un objeto a otro para que éste se comporte de una manera determinada, ejecutando uno de sus métodos. Los mensajes comunican a los objetos con otros y con el mundo exterior. A esta técnica de enviar mensajes se le conoce como *paso de mensajes.*

- ***Clases***

Es un modelo o contenedor definido por el usuario que determina la estructura de datos y las operaciones asociadas con ese tipo.

4) Características

- ***Abstracción***

Significa extraer las propiedades esenciales de un objeto que lo distinguen de los demás tipos de objetos y proporciona fronteras conceptuales definidas, respecto al punto de vista del observador. Es la capacidad para encapsular y aislar la información de diseño y ejecución.

- ***Encapsulamiento***

Es el proceso de almacenar en un mismo compartimiento (una caja negra) los elementos de una abstracción (toda la información relacionada con un objeto) que constituyen su estructura y su comportamiento. Esta información permanece oculta tanto para los usuarios como para otros objetos y puede ser accedida solo mediante la ejecución de los métodos adecuados.

♦ *Herencia*

Es la propiedad que permite a los objetos construirse a partir de otros objetos. La clase base contiene todas las características comunes. Las sub-clases contienen las características de la clase base más las características particulares de la sub-clase. Si la sub-clase hereda características de una clase base, se trata de herencia simple. Si hereda de dos o más clases base, herencia múltiple.

♦ *Polimorfismo*

Literalmente significa "cualidad de tener más de una forma". En POO, se refiere al hecho que una misma operación puede tener diferente comportamiento en diferentes objetos. En otras palabras, diferentes objetos reaccionan al mismo mensaje de modo diferente.

5) Ventajas

♦ *Modelos*

La POO permite realizar un modelo de sistema casi independientemente de los requisitos del proyecto. La razón es que en la POO la jerarquía la establecen los datos, en cambio en la programación estructurada, la jerarquía viene definida por

los programas. Este cambio hace que los modelos se establezcan de forma similar al razonamiento humano y, por lo tanto, resulte más natural.

- ***Modularidad***

Un programa es modular si se compone de módulos independientes y robustos. Esto permite la reutilización y facilita la verificación y depuración de los mismos. En POO, los módulos están directamente relacionados con los objetos. Los objetos son módulos naturales ya que corresponden a una imagen lógica de la realidad.

- ***Extensibilidad***

Durante el desarrollo de sistemas, ocurre la aparición de nuevos requisitos, por eso es deseable que las herramientas de desarrollo permitan añadirlos sin modificar la estructura básica del diseño. En POO es posible lograr esto siempre y cuando se hayan definido de forma adecuada la jerarquía de clases, los atributos y métodos.

- ***Eliminación de redundancia***

En el desarrollo de sistemas se desea evitar la definición múltiple de datos y funciones comunes. En POO esto se logra mediante la herencia (evita la definición múltiple de propiedades comunes a muchos objetos) y el polimorfismo (permite la modificación de métodos heredados). Solo hay que definir los atributos y los métodos en el antepasado más lejano que los comparte.

- ***Reutilización***

La POO proporciona un marco perfecto para la reutilización de las clases. El encapsulamiento y la modularidad nos permiten utilizar una y otra vez las mismas clases en aplicaciones distintas. En efecto, el aislamiento entre distintas clases

significa que es posible añadir una nueva clase o un módulo nuevo (extensibilidad) sin afectar al resto de la aplicación.

6) Lenguajes en POO

• Puros

Son los que solo permiten realizar programación orientada a objetos. Ejemeplo: smalltalk, java.

• Híbridos

Son los que permiten la POO con la programación estructurada. Ejemplo: C++, pascal.

7) POO en C++ y java

a) Tipos de clases

Una de las principales decisiones al trabajar con POO es la de selección de clases. Existen 4 tipos:

1. Manejadoras de datos o de estados: su responsabilidad principal es mantener información de datos o estado. Se reconocen como los sustantivos en la descripción de un problema y generalmente son los bloques de construcción más importantes de un diseño.

2. Pozos o fuentes de datos: estas clases generan datos o los aceptan para procesarlos mas adelante. A diferencia de los anteriores, estas clases no retienen

los datos por un periodo de tiempo, sino que los genera sobre demanda o los procesa cuando se le llama.

3. Vistas: se encargan de la presentación de la información.

4. Auxiliares o de ayuda: guardan poca o ninguna información de estado, pero que asisten en la ejecución de tareas complejas.

b) Sintaxis de una clase

C++:

```
Class nombre_clase [: [public/protected/private] clase_madre]
     {
          [lista de atributos];
          [lista de metodos];
     };
```

Java:

```
[public]   [final/abstract]   class   nombre_clase   [extends
clase_madre]
                         [implements              interface1,
[interface2,...]...]
     {
          [lista de atributos];
          [lista de metodos];
     };
```

c) Modificadores de acceso a miembros de clases

Existen 3 tipos de usuarios de una clase:

– La propia clase.

– Usuarios genéricos (otras clases, métodos, etc.)

– Clases derivadas.

Cada usuario tiene distintos privilegios o niveles de acceso.

C++:

- *Private*

Por defecto todo lo declarado dentro de la clase es privado y solo puede ser accedido por Funciones miembro o por funciones amigas.

- *Public*

Pueden ser accedidos por funciones miembro y no miembro de una clase.

- *Protected*

Pueden ser accedidos por función miembro, por funciones amigas o por funciones miembro de sus Clases derivadas.

Java:

- *Private*

Solo puede ser accedida por métodos propios de la clase.

- *Private-protected*

Pueden ser accedidos por las sub-clases, sin importar el paquete al que pertenezcan. Sin embargo, las sub-clases solo pueden modificar estos atributos para objetos de la sub-clase, no de la clase Madre.

- *Protected*

Permite el acceso a las sub-clases y a las clases del mismo paquete.

- *Friendly*

Es el valor por defecto. Permite el acceso solo a clases del mismo paquete.

- *Public*

Pueden ser accedidas por todos.

d) Atributos

Las variables declaradas dentro de un método son locales a el; las declaradas en el cuerpo de la clase son miembros de ella y son accesibles por todos los métodos de la clase. Los atributos miembros de la clase pueden ser: atributos de clase (declarado como static) o atributos de instancia.

C++:

[static]tipo_dato nombre_dato; (no se puede inicializar un dato miembro de una clase)

Java:

[modificador_de_acceso] [static] [final] [transient] [volatile] tipo_dato nombre_dato [= valor];

- *Final*

Implica que un atributo no puede ser sobre-escrito o redefinido, es decir que no se trata de una Variable sino de una constante.

- *Transient*

Son atributos que no se graban cuando se archiva un objeto, o sea no forman parte permanente del estado de éste.

- *Volatile*

Se utiliza cuando la variable puede ser modificada por distintos *threads*. Básicamente implica que varios *threads* pueden modificar la variable en forma simultánea, y *volatile* asegura que se vuelva A leer la variable, por si fue modificada, cada vez que se la vaya a usar.

e) Métodos

Dentro de los métodos pueden incluirse:

- ✓ Declaraciones de variables locales.
- ✓ Asignaciones a variables.
- ✓ Operaciones matemáticas.
- ✓ Llamados a otros métodos.
- ✓ Estructuras de control.
- ✓ Excepciones.

C++:

Se puede declarar y definir dentro de la clase:

```
Class nombre_clase
{
     tipo_dato dato;
     [modificador_de_acceso]:
                    tipodato_devuelto
nombre_metodo(parametros)
                              {
                                   cuerpo_metodo;
                              };
```

```
}
```

También, se puede declarar dentro de la clase y definirlo fuera:

```
Class nombre_clase
{
      tipo_dato dato;
      [modificador_de_acceso]:
                        tipodato_devuelto nombre_metodo(parametros);
};

Tipodato_devuelto nombre_clase :: nombre_metodo(parametros)
{
      cuerpo_metodo;
}
```

Java:

En java se debe declarar y definir la función dentro de la clase:

```
Class nombre_clase
{
     tipo_dato dato;
     [modificador_de_acceso]   [static]   [abstract]   [final]
[native] [syncronized] tipodato_devuelto
                        nombre_metodo     (parametros)[throws
exepcion1 [,exepcion2]]
                              {
                                   cuerpo_metodo;
                              }
```

}

- Static

Al igual que con los atributos, un método declarado como static es compartido por todas las Instancias de la clase.

- Abstract

Son aquellos de los que se da la declaración pero no la implementación, para generar una clase Abstracta.

- Final

Es un método que no puede ser redefinido por las sub-clases herederas.

- Native

Es un método que esta escrito en otro lenguaje que no es java.

- Syncronized

Permite sincronizar varios threads para el caso en que dos o más quieran acceder en forma Concurrente.

- Throws

Sirve para indicar que la clase genera determinadas excepciones.

- Llamada a métodos

Se llama a un método de la misma clase simplemente con el nombre del método y los parámetros entre paréntesis.

Cuando se necesita llamar a un método de un objeto de otra clase, se utiliza:

Nombre_objeto.nombre_metodo(parametros)

f) Métodos o funciones miembros

Funciones simples (c++ y java)

Son los vistos anteriormente, se pueden definir dentro o fuera de la clase.

Funciones en línea (inline) (c++)

Se crean definiendo la función dentro de la clase (declaración implícita) o definiendo la función fuera de la clase pero anteponiendo la palabra reservada inline (declaración explícita).

Funciones constructores (c++ y java)

Es una función especial que sirve para inicializar objetos de una clase. En general, tienen el mismo nombre de la clase que inicializa, no devuelven valores, pueden admitir parámetros, pueden existir mas de un constructor (e incluso no existir), si no se define ningún constructor el compilador genera uno por defecto, se llaman al momento de crear un objeto.

Constructores por defecto

Es un constructor que no acepta argumentos. Si no se define, el compilador genera uno que asigna espacio de memoria y lo pone en cero.

Constructores con argumentos

Inicializan un objeto con los valores del argumento. Pueden existir varios que se diferencian por la cantidad y tipo de argumentos.

Constructores copiadores

Crea un objeto a partir de uno existente. Toma como único parámetro otro objeto del mismo tipo. Si no se declara el compilador genera uno por defecto asignándole al nuevo objeto los valores del objeto a la izquierda del operador =.

En java se puede inicializar el objeto tras la creación física del mismo, asignándole los valores que se le dará.

Función destructor (c++)

Cumplen la función inversa a la del constructor, eliminando el espacio de almacenamiento que ocupo el objeto al ser creado. Características: tienen el mismo nombre de la clase, van precedidos por el carácter ~, solo existe un destructor por clase, no admiten argumentos, no retornan ningún valor, el compilador llama a un destructor cuando el objeto sale fuera de ámbito.

Funciones amigas (c++)

Es una función no miembro que puede acceder a las parte private de una clase. Se declaran anteponiendo la palabra reservada friend. También se pueden declarar como amigas a las clases. En este caso todas las funciones de la clase amiga pueden acceder a las partes privadas de la otra clase. Funciones sobrecargadas (c++ y java) es una función que tiene más de una definición. Las funciones sobrecargadas tienen el mismo nombre, pero deben tener un número distinto de argumentos o diferentes tipos de argumentos. Las definiciones operan sobre funciones distintas.

Las únicas funciones miembro que no se pueden sobrecarga son los destructores.

Funciones operador (c++)

Permiten sobrecargar un operador existente (ej: suma) para utilizarlos con objetos. Se declara poniendo la palabra reservada operator seguida por el operador específico y la lista de argumentos y el cuerpo de la función.

Restricciones:

- El operador debe ser uno valido para c++ y no se puede cambiar el símbolo.
- Funciona solo al utilizar objetos de clase.
- No se puede cambiar la asociación de los operadores, es decir para que sirven.
- No se puede cambiar un operador binario para funcionar con un único objeto.
- No se puede cambiar un operador unario para que funcione con dos objetos.

g) Objetos

C++:

Se pueden crear en forma estática (nombre_clase objeto1) o dinámica (nombre_clase *objeto2; Objeto2= new nombre_clase;)

Java:

Solo se crean los objetos en forma dinámica: nombre_clase objeto1=new nombre_clase();

h) Arrays de objetos

Se pueden crear arrays de objetos de la misma forma que se crea un array normal.

C++:
Estatico: nombre_clase objeto[10];
Dinamico: nombre_clase *objeto;
Objeto=new nombre_clase[10];

Java:
Nombre_clase objeto=new nombre_clase[10];

i) Puntero this

Es un puntero al objeto asociado con la invocación de una función miembro. Normalmente no se explicita ya que el lenguaje realiza esta operación transparente y en forma automática. Igualmente hay casos en los que se debe usar el puntero this explícitamente:

- Como argumento en una llamada a una función para pasar un puntero al objeto asociado con La invocación de la función miembro.

Ej: f(this);

Hacer una copia del objeto asociado con la invocación o asignar un nuevo valor al objeto.

```
Ej: void t::g (t &a, t &b)
    {     ...
          a= *this;
          ...
          *this= b;
```

```
}
```

Devolver una referencia al objeto asociado con la invocación de la función miembro o

Constructor

Ej: t& t::f (int a)

```
{       ...
        return *this;
}
```

C++:

This->nombre_objetomiembro

Java:

This.nombre_objetomiembro

2. HERENCIA Y POLIMORFISMO

1) Herencia

Herencia es la propiedad de que los ejemplares de una clase hija extiendan el comportamiento y los datos asociados a las clases paternas. La herencia es siempre transitiva, es decir que una sub-clase hereda características de superclases alejadas muchos niveles.

2) Beneficios de la herencia

Reusabilidad del software cuando el comportamiento se hereda de otra clase, no es necesario reescribir el código que lo define.

a) Compartición de código

Muchos usuarios o proyectos distintos pueden usar las mismas clases. Por otro lado, la herencia reduce el tiempo de escritura y el tamaño final del programa.

b) Consistencia de la interfaz

El comportamiento de una clase madre que heredan todas sus clases hijas será el mismo, de esta manera se asegura que las interfaces para objetos serán similares y no solo un conjunto de objetos que son parecidos pero que actúan e interactúan de forma diferente.

c) Componentes de software

La herencia nos permite escribir componentes de software reutilizables.

d) Modelado rápido de prototipos

Cuando un sistema se construye casi totalmente de componentes reutilizables, se puede dedicar más tiempo a la realización de aquellas partes nuevas. A este estilo de programación se lo conoce como 'modelado rápido de prototipos o programación exploratoria'.

e) Ocultación de información

Un programador puede reutilizar un componente conociendo solamente la naturaleza e interfaz del mismo y sin la necesidad de conocer los detalles técnicos empleados para su realización.

f) Polimorfismo

Permite al programador generar componentes reutilizables de alto nivel que puedan adaptarse a diferentes aplicaciones mediante el cambio de sus partes de bajo nivel.

3) Heurísticas para crear sub-clases

Para saber si una clase debe convertirse en una subclase de otra mediante la herencia, hay que aplicar la regla del es-un y es-parte-de.

a) Es-un

Se dan entre dos conceptos cuando el primero (la sub-clase) es un ejemplar especificado del segundo (la clase base). Ej: un avión es un transporte.

b) Es-parte-de

Se da entre dos conceptos cuando el primero (la sub-clase) es una parte del segundo (la clase base), sin ser ninguno, en esencia, la misma cosa. Ej: un motor es parte de un auto.

c) Distintos tipos de herencia

→ Especialización

Es la forma de herencia mas común y cumple en forma directa la regla es-un.

→ Especificación

Se trata de un caso especial de sub-clasificación por especialización, excepto que las sub-clases no son refinamientos de un tipo existente, sino más bien realizaciones de

una especificación incompleta. Es decir, que la superclase describe un comportamiento que será implantado solo por las sub-clases.

→ Construcción

Se da cuando la sub-clase ha heredado casi completamente su comportamiento de la superclase y solo tiene que modificar algunos métodos o los argumentos de cierta manera.

→ Generalización

Esta es la opuesta a la creación de sub-clases por especialización. Se debe evitar; solo deben aplicarse cuando no se pueden modificar las clases existentes o se deben anular métodos de las mismas.

→ Extensión

Agrega una capacidad totalmente nueva a un objeto existente. Se distingue de la generalización, ya que esta debe anular al menos un método de la clase base, mientras que la extensión solo agrega métodos nuevos.

→ Limitación

Es una variante de la especificación en donde el comportamiento de la sub-clase es más reducido o está más restringido que el comportamiento de la superclase. También se da en situaciones en las que las clases existentes no pueden modificarse.

→ Variación

Se da cuando do o más clases tienen implantaciones similares, pero no parece haber ninguna relación jerárquica entre los conceptos representados por las clases. Ej: código del mouse y de la placa de video.

→ Combinación

Se da cuando una sub-clase resulta de la combinación de características de dos o más clases.

4) Herencia y jerarquía de clases

C++ y java utilizan un sistema de herencia jerárquica. Una clase hereda de otra, creando así muevas clases a partir de clases existentes. Solo se pueden heredar clases, no funciones ordinarias ni variables. Una sub-clase deriva de una clase base. La clase derivada puede a su vez ser utilizada como clase base para derivar más clases, conformándose así la jerarquía de clases.

Características de las clases derivadas

Una clase derivada o sub-clase:

- Puede a su vez ser una clase base, dando lugar a la jerarquía de clase.
- Los miembros heredados por una clase derivada, pueden a su vez ser heredados por más clases derivadas a ella.
- Hereda todos los miembros de la clase base, pero solo podrá acceder a aquellos que los especificadores de acceso de la clase base lo permitan. Las clases derivadas solo pueden acceder a los miembros public, protected y private-protected (en java) de la clase base, como si fueran miembros propios.
- No tienen acceso a los miembros private de la clase base.
- Pueden añadir sus propios datos y funciones miembros.

Los siguientes elementos de la clase base no se heredan:

- Constructores y destructores.
- Funciones y datos estáticos de la clase.

- Funciones amigas y operadores sobrecargados (solo c++).

Sintaxis de una clase derivada

C++:

Class clase_derivada : [public / protected / private] clase_base [, [public / protected / private]clase_base2]

{

cuerpo clase derivada;

};

Java:

Class nombre_derivada extends nombre_base

{

cuerpo clase derivada;

}

Especificadores de acceso

C++:

Si no se especifica el tipo de derivación, c++ supone que el tipo de herencia es private.

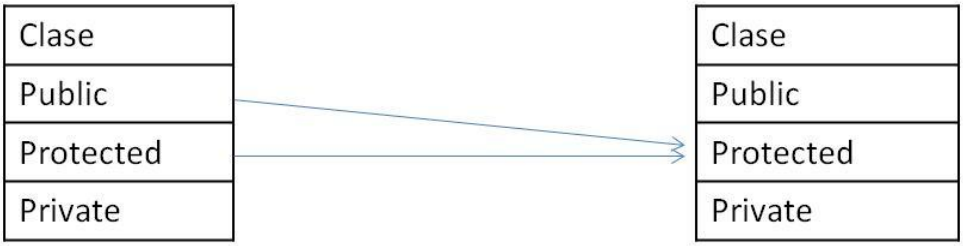
Si en la derivación especificamos:

- **Public**

Los miembros public pasan a ser public en la clase derivada. Los miembros protected pasan a ser protected. Y los privados permanecen privados en la clase base.

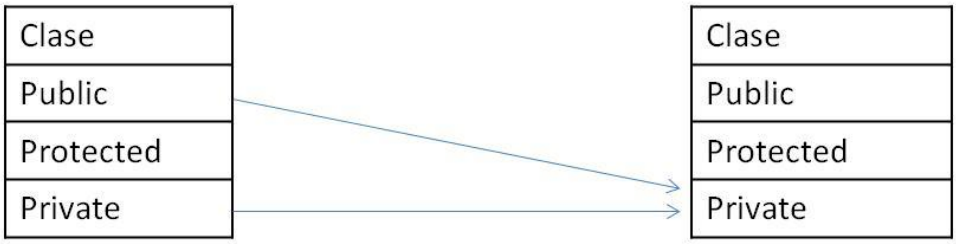
- **Protected**

Todos los miembros public y protected de la clase base son miembros protected en la derivada.



- **Private**

Todos los miembros public y private son miembros private en la clase derivada.



Java:

Para analizar los tipos de herencia en java hay que tener en cuenta que existen 5 distintos tipos de clases:

- Clase base.
- Clases derivadas del mismo paquete.
- Clases derivadas en distinto paquete.
- Clases del mismo paquete.
- Clases en distinto paquete.

	Miembros declarados como…				
	Private	Private-protected	Protected	Friendly	Public
Clase base.	S	S	S	S	S
Clases derivadas del mismo paquete.	N	R	S	S	S
Clases del mismo paquete.	N	N	S	S	S
Clases derivadas en distinto paquete.	N	R	R	N	S
Clases en distinto paquete.	N	N	N	N	S

Tipo de acceso:

S: acceso posible.
N: acceso denegado.
R: reservado a las subclases. Acceso posible si el miembro se refiere a un objeto de la subclase y no a uno de la clase base.

Redefinición de métodos

Cuando se hace heredar una clase de otra, se pueden redefinir ciertos métodos a fin de refinarlos, o bien de modificarlos. El método lleva el mismo nombre y la misma signatura, pero solo se aplica a objetos de la sub-clase o sus descendientes.

Constructores

Cuando se construye una clase hija, es necesario también llamar al constructor de la clase madre. Esta invocación puede ser implícita o explícita.

Es implícita si el constructor de la madre no toma parámetros. En este caso, ya sea que el constructor de la hija sea implícito o no, la llamada al constructor de la madre es automática.

Si el constructor de la madre necesita parámetros, hay que pasárselos. Es necesario definir un constructor en la hija y este debe llamar al constructor de la madre enviando los parámetros.

Para realizarlo se sigue el siguiente orden:

- → Llamar al constructor de la clase madre.
- → Construir los miembros de la clase hija.
- → Ejecutar las instrucciones contenidas en el cuerpo del constructor de la clase hija.

C++:

```
Derivada :: derivada (tipo1 x, tipo2 y) : base (x,y) [, base2 (x,y) ]
{
```

```
        cuerpo del constructor;
}
```

En herencia múltiple, el orden real de invocación de constructores se da de acuerdo al orden en que fueron declaradas las clases. C++ utiliza el siguiente orden de inicialización:

- → Primero, se inicializan todas las clases bases virtuales.
- → Las clases bases no virtuales se inicializan en el orden en que aparecen en la declaración de clases.
- → Por último, se ejecuta el constructor de la clase derivada.

Java:

En la definición del constructor de la clase derivada se utiliza la palabra super para simbolizar la llamada al constructor de la clase base.

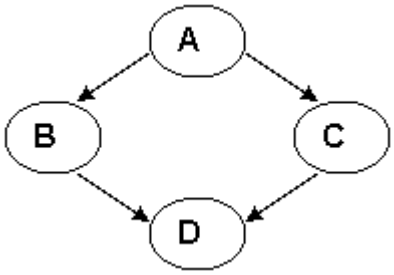
```
Class hija extends madre
{
        public hija (tipo1 x)
        {
                super (y);
                cuerpo del constructor;
        ]
}
```

Destructores

Al contrario que con los constructores, una función destructor de una clase derivada se ejecuta antes que el destructor de la clase base (los destructores no se heredan).

Clases bases virtuales

Es una clase que es compartida por otras clases base con independencia del número de veces que esta se produce en la jerarquía de derivación.



Supongamos que la clase base A, tiene sus propios miembros. Definimos dos clases derivadas de ésta, B y C, también con sus propios métodos y los heredados. Definimos también una clase D, derivada de B y C (derivación múltiple). Este tipo de jerarquía puede dar lugar a problemas ya que la clase D heredará las funciones miembro de B y C, pero también derivará las de A dos veces.

Para evitar este problema, debemos definir a las clases base como clases virtuales, lo cual significa que solo una copia de los miembros ambiguos pasara a la clase derivada.

```
Class a
{…}
Class b : virtual public a
{…}
Class c : virtual public a
{…}
Class d : public b , public c
{…}
```

5) Polimorfismo

Consiste en llamar a un método según el tipo estático de una variable, basándose en el núcleo para llamar a la versión correcta del método. El polimorfismo se realiza por estos métodos:

→ Sobrecarga de funciones (c++ y java)

En este caso el compilador determina qué función debe utilizarse en tiempo de compilación ya que conocerá el objeto implicado. Este tipo de ligadura se la conoce como ligadura temprana, estática o previa.

Cuando existen punteros implicados, el compilador no sabe a que objeto se está referenciando. En este caso se produce una ligadura dinámica.

→ Con funciones virtuales (c++)

Cuando varias sub-clases derivan de una clase base, éstas pueden emplear las funciones que heredan de la misma forma, pero otras pueden requerir elementos adicionales o incluso formatos totalmente nuevos. Para enfrentar esto, existen 3 soluciones posibles:

a) Definir las funciones con distintos nombres (es fácilmente realizable pero no es ideal porque genera complejidad).
b) Sobrecargar la función. Esto es útil cuando la jerarquía de clases es pequeña.
c) Identificar la función miembro de la clase base, que se puedan llegar a utilizar en clases derivadas y declararlas como virtuales.

En este último caso, la ligadura se lleva a cabo en tiempo de ejecución.

La función debe ser declarada como virtual en la primera clase en la que está presente. La palabra clave virtual permite definir la función bajo el mismo nombre tanto en la clase base como en la derivada. Al utilizar funciones virtuales se pueden utilizar punteros a la clase base para referenciar objetos de una clase derivada.

Función virtual pura

Es una función virtual declarada en una clase base que no está definida y no tiene cuerpo. Deben definirse luego en la clase derivada.

C++:

Virtual tipo nombre_funcion (parametros) = 0;

Java:

Se llaman métodos abstractos y se declaran pero no se escribe su implementación.
Abstract metodoabstracto ();

Clases abstractas

Son clases que se diseñan para ser heredadas y solo se pueden utilizar como clases base. No pueden ser instanciadas.

C++:

Una clase es abstracta si tiene al menos una función virtual pura.

Java:

Deben declararse como abstract.

```
Abstract nombre_clase
{
        abstract void metodo_abstracto ();
```

```
}
```

Pasos para obtener polimorfismo en C++

1. Crear una jerarquía de clases con las funciones miembro importantes definidas como virtuales. Si las clases base son tales que no se pueden implementar estas funciones en ellas, declarar funciones virtuales puras.
2. Proporcionar implementaciones concretas de clases virtuales en las clases derivadas. Cada clase derivada tiene su propia versión de las funciones.
3. Manipular instancias de las clases derivadas a través de punteros.