Metodología de la Programación:

Programación Orientada a Objetos

I. Desarrollo de Software Orientado a Objetos

1. El desarrollo del software
2. Modularidad
3. Conceptos fundamentales de POO
4. Lenguajes de POO
5. Modelado de objetos: Relaciones

II. POO con C++

6. Clases y objetos
7. Clases abstractas y herencia
8. Polimorfismo
9. Genericidad
10. Excepciones

III. Diseño y Reutilización Orientado a Objetos

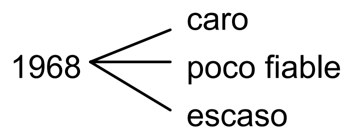
11. Reutilización de software con C++
12. Diseño orientado a objetos

IV. El lenguaje C++

13. Sintaxis del lenguaje
14. De C a C++
15. Construcción de programas en C++
16. Puesta a punto de programas. Errores típicos de programación

Tema 1: El Desarrollo del Software

La Programación Orientada a Objetos (en adelante POO), surge a partir de la llamada crisis del software.



La causa, parece ser la complejidad inherente al propio software:

- Complejidad del dominio del problema
  - Interacción usuario-desarrollador
  - Cambios requisitos durante desarrollo
- Dificultad en el proceso de desarrollo
  - Los programadores hacen partes distintas que han de ser compatibles
- Flexibilidad que se requiere al software

unida al ciclo de vida del software en cascada:

Análisis
Diseño
Implementación
Pruebas
Mantenimiento

Existían barreras para pasar de un escalón a otro, además hay que dar más importancia a la fase de mantenimiento:

REUTILIZACIÓN DEL CÓDIGO EXISTENTE

Calidad del software: Factores

- Eficiencia: buen uso de los recursos.
- Transportabilidad: entre plataformas distintas.
- Verificabilidad: facilidad de comprobar el software.
- Integridad: protección de sus componentes.
- Facilidad de uso.
- Corrección: hacer lo que se pide sin fallos.
- Robustez: salvar situaciones anormales.
- Extensibilidad: capacidad de cambio o evolución.
- Reutilización: ahorro de trabajo.
- Compatibilidad: facilidad de combinar subprogramas.

En resumidas cuentas:

Programación mediante abstracción

- Abstracción del mundo real.
- Abstracción de la máquina.

Abstracción es la capacidad de aislar y encapsular la información del diseño y la ejecución.

Evolución de la abstracción:

Procedimientos	***aMódulos***	***aTADs***	***aObjetos***
• datos globales • datos locales	• parte visible • parte oculta	• exportar tipos • proporcionar operaciones tipo • proteger datos • ocultar implementación • ejemplares múltiples	• compartición de código • reutilización

Características del modelo objeto

1. Modularización: descomponer la aplicación en partes más pequeñas.
2. Abstracción.
3. Encapsulación: ocultación de información.
4. Jerarquización: relaciones de herencia o generalización (es_un) y relaciones de contenido o agregación (tiene_un).
5. Polimorfismo: capacidad de referirse a objetos de clases distintas en una jerarquía utilizando el mismo elemento de programa para realizar la misma operación, pero de formas distintas.

Características deseables en el modelo objeto

6. Concurrencia.
7. Persistencia: un objeto puede seguir existiendo tras desaparecer su antecesor.
8. Genericidad (unidades genéricas de programación).
9. Manejo de excepciones.

Tema 2: Modularidad

Modularización

Descomponer un programa en un número pequeño de abstracciones coherentes que pertenecen al dominio del problema y enmascaran la complejidad interna.

Según **Liskov**, modularizar es dividir un programa en módulos que pueden ser compilados separadamente pero existiendo conexiones entre los módulos.

Parnas dice además que deben (las conexiones) seguir el criterio de ocultación.

Booch piensa que la modularización deber ser propiedad de un sistema que ha sido descompuesto en un conjunto de módulos cohesivos y débilmente acoplados.

Acoplamiento

Es la interacción entre módulos, sus propiedades deberían ser:

1. Facilitar la sustitución de un módulo realizando pocos cambios en los otros.
2. Facilitar el seguimiento y aislamiento de un error (módulo defectuoso).

Cohesión

Es la interacción interna de un módulo: todos los elementos de un módulo han de tener relación.

Tema 3: Conceptos fundamentales de POO

Definición de Booch

La POO es el método de implementación en el que los programas se organizan como colecciones cooperativas de objetos, cada uno de los cuales representa un ejemplar de una clase y cuyas clases son miembros de una jerarquía de clases unidas mediante relaciones de herencia.

Conclusiones

- La unidad lógica de programación es el objeto.
- Los objetos tienen relaciones.
- Existen clases que agrupan conceptualmente los objetos.
- Las clases pertenecen a una jerarquía (son las clases las que heredan y no los objetos).

Objeto

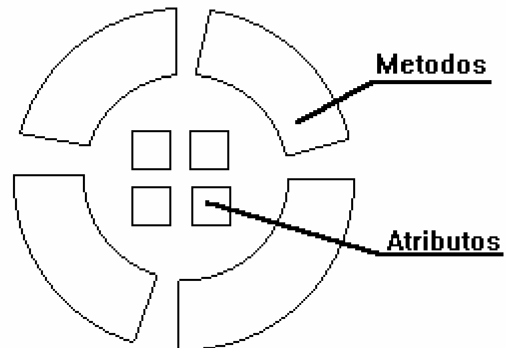
Entidad que contiene los atributos que describen el estado de un objeto del mundo real y las acciones que se asocian con el objeto del mundo real.
Un objeto es designado con un nombre o un identificador.

OBJETO = DATOS + OPERACIONES

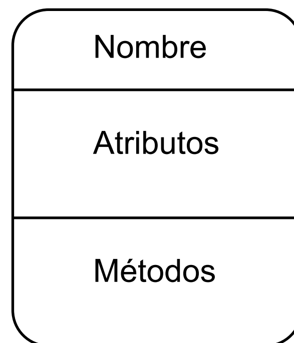
Los datos deberían estar ocultos en el objeto, y las operaciones serían el interface del objeto con el exterior, pero estas operaciones están encapsuladas en "cajas negras".

Notación

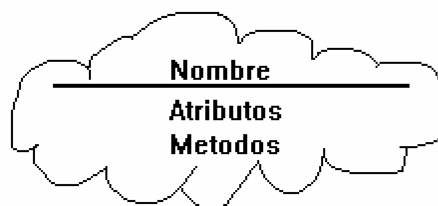
Taylor



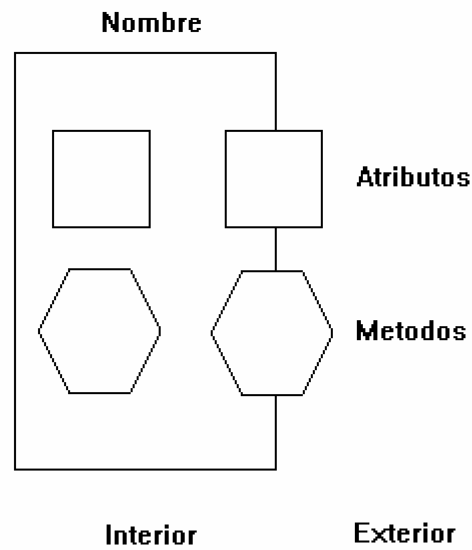
Yourdon/Coad



Booch



Booch



Métodos y mensajes

Los objetos se comunican mediante llamadas a funciones_miembro o métodos: se dice entonces que "un objeto envía un mensaje a otro objeto".
El mensaje es la acción realizada por un objeto mediante la cual se invoca un método.

Componentes del mensaje:

- Nombre del receptor del mensaje
- Método invocado
- Información necesaria para la ejecución del método

Resumiendo, mensaje es la activación de un objeto.
El conjunto de mensajes que puede recibir un objeto se conoce como protocolo de un objeto.

Clases

Una clase es la descripción de un conjunto de objetos. Consta de métodos y datos que resumen las características comunes de un conjunto de objetos.
Muestra el comportamiento general de un grupo de objetos.

DISTINTOS OBJETOS ¬ DISTINTAS CLASES

Un objeto se crea mediante el envío de un mensaje de construcción a la clase.
Análogamente para la destrucción del objeto, la clase recibe un mensaje de destrucción.

Un programa orientado a objetos se resume en 3 sucesos:

1. Creación de objetos cuando se necesitan, mediante un mensaje de construcción a la clase.
2. Intercambio de mensajes entre objetos o entre usuario de objeto y objeto (diferencia fundamental entre lenguajes POO puros e híbridos).
3. Eliminar objetos cuando no se necesitan, mediante un mensaje de destrucción a la clase.

Identificación de clases

Partiendo del enunciado de un problema:

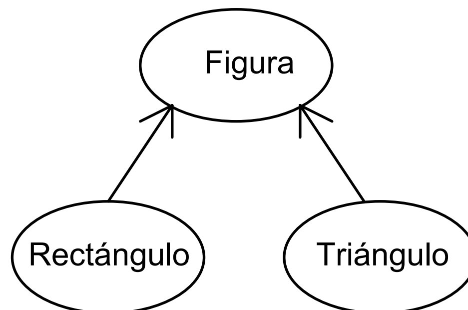
1. Todos los nombres del enunciado son objetos a tener en cuenta.
   i Cosas tangibles ("coche")
   i Roles o papeles ("empleado")
   i Organizaciones ("empresa")
   i Incidentes o sucesos ("liquidación")
   i Interacciones o relaciones ("pedido")
2. Los atributos son las características individuales de cada objeto, que serán extraídos de los adjetivos y complementos del verbo que haya en el enunciado.
3. Los métodos serán los verbos del enunciado. Tipos de método:
   i Constructor
   i Destructor
   i Modificadores del estado
   i Selectores (obtienen el estado del objeto: "visualizar")
   i Mezcladores (ej. "sumar 2 números complejos")
   i Cálculos o procesamientos

Herencia

Herencia es la capacidad de un objeto (clase) para utilizar las estructuras y los métodos existentes en antepasados o ascendientes.
Es la reutilización de código desarrollado anteriormente.
Cuando usamos herencia decimos que hacemos programación por herencia: Definición de nuevos tipos a partir de otros con los que comparten algún tipo de característica.

Tipos de herencia

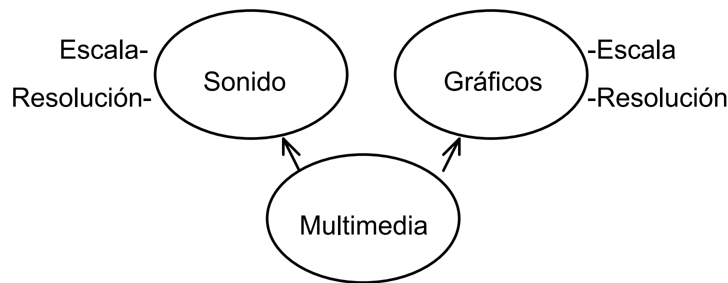
Herencia simple: un tipo derivado se crea a partir de una única clase base.



Herencia múltiple: una clase tienen más de una ascendiente inmediato.

La herencia múltiple puede plantear 2 tipos de problema:

- La *herencia repetida*: ej.: "profesor universitario hereda 2 veces los atributos de persona"
- Produce *ambigüedad* respecto a los atributos o los métodos. En la clase base pueden haber atributos que se llamen igual.



Sólo 2 lenguajes incorporan herencia múltiple: Eiffel y C++

Características de la herencia

- Anulación o sustitución: cuando redefino un método heredado en la subclase, se dice que estoy anulando o sustituyendo dicho método. Sería deseable una "herencia selectiva": seleccionar lo que se requiere heredar es la mejor forma de anulación.
- Sobrecarga: Propiedad que puede darse también sin herencia. Es designar varios elementos (identificadores) con el mismo nombre. No es anulación.
- Polimorfismo (sobrecarga con anulación) es la forma de invocar a distintos métodos utilizando el mismo elemento de programa.

Polimorfismo

Polimorfismo es invocar métodos distintos con el mismo mensaje (ligadura en tiempo de ejecución).
Para ello es necesaria una jerarquía de herencia: una clase base que contenga un método polimórfico, que es redefinido en las clases derivadas (no anulado).
Se permite que los métodos de los hijos puedan ser invocados mediante un mensaje que se envía al padre.
Este tipo de clase que se usa para implementar el polimorfismo se conoce como clase abstracta.

Objetos Compuestos

Los objetos pueden contener a su vez objetos de otras clases dentro de sí (esto es lo que da la potencia a la POO).
El contenido la mayoría de las veces no es el objeto en sí, sino una referencia (puntero) al objeto; con la ventaja de que el objeto contenido puede cambiar de contenido o posición sin afectar al objeto que contiene.
El objeto contenido puede, además, estar en varios objetos a la vez.
Al mismo tiempo que se crea un objeto, se crean los objetos que contiene.

Tema 4: Lenguajes de POO

Introducción

SIMULA

Se considera que **SIMULA** es el primer Lenguaje de Programación Orientado a Objetos (en adelante LPOO), fue diseñado por **Kristen Nygaard** y **Ole Johan Dhal**, del **Norwegan Computer Center** (NCC). Se concibió para el desarrollo de simulaciones de procesos industriales y científicos, pero llegó a considerarse apto para desarrollar grandes aplicaciones.

La versión **SIMULA_67** tenía características de **ALGOL_60**, y ya era casi un LPOO, con estas características:

- Concepto de objeto (datos + operaciones)
- Comunicación entre objetos (proceso de simulación)
- Concepto de clase (estructura que agrupa el comportamiento y características de varios objetos)
- Concepto de herencia (heredar datos y métodos)
- Fuertemente tipificado (comprobación de tipos en tiempo de compilación)
- Concepto de ligadura dinámica "polimorfismo" (tener interfaces idénticos para activar diferentes propiedades.

Smalltalk

Más tarde surge el primer LPOO en sentido estricto: **Smalltalk_80**, diseñado por **Alan Kay** y **Adele Goldberg**, de la **XEROX PARC**. Además de ser considerado como primer LPOO es el más puro (sólo tiene objetos, NO ES programación estructurada)

Extensiones de Lenguajes Convencionales

C++ [1986]

Diseñado por **Stroustrup**, es un LPOO híbrido basado en **C** y **Smalltalk**. Ha tenido mucho éxito.

Objetive C [1986]

Es una extensión de **C**. Hoy está en desuso. Surge para las plataformas **NeXT** y fracasó junto a las máquinas **NeXT**.

Object Pascal [1987]

Extensión de **Pascal**, lo popularizó **Borland** con **Turbo Pascal 5.5**

Object COBOL [1992-3]

Extensión de **COBOL**, nuevo LPOO que parece haber tenido aceptación.

Delphi [1995]

Proviene de **Object Pascal**, creado por **Borland**.

Extensiones de Lenguajes Funcionales

PROLOG++

CLOS

Common Lisp orientado a objetos

Lenguajes fuertemente tipificados

Ada_95

Ada_83 ya era casi un LPOO. **Ada_95** sólo añade herencia y ligadura dinámica.

Eiffel [1988]

Diseñado por **Beltran Meyer**, está aún en desarrollo.

Clasificación de Wegner de 1987

I. Lenguajes Basados en Objetos (LBO)

Son aquellos en los que la sintaxis y semántica del lenguaje soporta la creación de objetos.
Ejemplos: **Ada_83**, **Clipper 5.x**, **VisualBasic**.

II. Lenguajes Basados en Clases (LBC)

Son **LBO** con capacidad de crear clases.
Ejemplo: **CLU**

III. Lenguajes Orientados a Objetos (LOO)

Son **LBC** con capacidad de herencia.
Ejemplos: **C++**, **Object Pascal**, etc...

Características de un LPOO

1. Tipificación fuerte: comprobación de tipos en tiempo de compilación. Hay una excepción: **Smalltalk**.
2. Ocultación: de las características de los objetos.
3. Compilación Incremental: compilación separada.
4. Genericidad: reutilización de código.
5. Paso de mensajes.
6. Polimorfismo: basado en la ligadura dinámica.
7. Excepciones: sistema seguro.
8. Concurrencia: "el mundo real es concurrente".
9. Datos compartidos: entre varios objetos.

LPOO Puros vs. Híbridos

	Puros	*Híbridos*
Ventajas	• Más potencia. • Flexibilidad para modificar el lenguaje. • Más fácil de asimilar (enseñar).	• Más rápido. • Más fácil el paso de programación estruc-turada a POO. • Implementación de operaciones-algoritmos fundamentales más sencilla.
Inconvenientes	• Más lento. • Implementación de operaciones-algoritmos fundamentales muy complicada.	• Trabaja con 2 paradigmas a la vez. • Modificar el lenguaje es difícil.

Ada_95

Incorpora 2 nuevas características a **Ada_83**:

- Herencia
- Polimorfismo

Smalltalk_80

Características

- Todas las entidades que maneja **Smalltalk** son objetos.
- Todas las clases derivan de una clase base llamada **Object**.
- Existe herencia simple.
- Usa métodos y paso de mensajes.
- Tiene una tipificación débil: la comprobación de los tipos se hace en tiempo de ejecución.
- Soporta concurrencia, pero pobre.
- Se comercializa con un conjunto de bibliotecas de clases predefinidas, agrupadas en categorías o jerarquías (E/S, etc...)
- Existe una versión, **Smalltalk V** para computadoras **IBM**.
- **C++** y **CLOS** se han apoyado en características de **Smalltalk**.

Eiffel

Es un lenguaje inspirado en **SIMULA** con características añadidas de **Smalltalk** y **Ada**.
Se destina a aplicaciones de bases de datos e inteligencia artificial.
Por su diseño, es bueno para la ingeniería de software.

Características

- Fuertemente tipificado.
- Clases.
- Genericidad.
- Herencia múltiple.
- Ligadura dinámica y polimorfismo.
- Se ha empezado a implementar concurrencia y persistencia de objetos.
- Proporciona una biblioteca de clases predefinidas: Estructuras de datos, E/S, E/S XWindow.
- La memoria dinámica es gestionada por el entorno y no por el sistema operativo: incorpora recogida automática de basura.

El entorno tiene editor, y Browser, que muestra las clases y jerarquías.
El entorno proporciona recompilación automática.

Contrato entre proveedor y usuario de clases

Para que un método pueda ser ejecutado debe haber:

- Una precondición
- Una postcondición
- Una invariante

Hay un contrato entre el proveedor (creador) de la clase y el cliente (usuario): el proveedor asegura que la invariante se cumple y que al finalizar la ejecución del método la postcondición se cumplirá; pero sólo siempre y cuando la otra parte (el usuario) cumpla la precondición.

Tema 5: Modelado de Objetos; Relaciones.

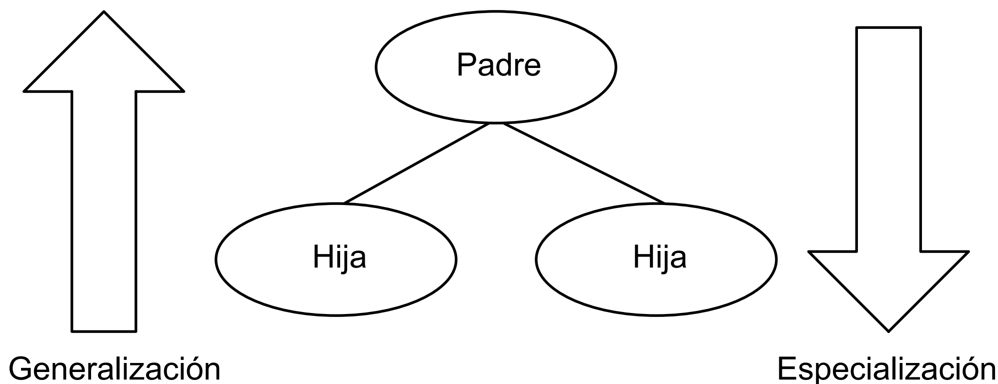
Existen 3 tipos de relaciones entre clases/objetos:

- Generalización
- Agregación
- Asociación

Relación de Generalización

Permite una estructuración jerárquica de las clases que comparten estructuras o comportamientos. Son relaciones del tipo "es_un" o "un_tipo_de". Son relaciones de herencia.

Desde el punto de vista de la clase ascendiente (padre), se trata de una generalización de las hijas en el padre.
Desde el punto de vista de las clases descendientes (hijas), se trata de una especialización de la clase base.



Una relación de herencia es siempre transitiva (Padre_Hijo_Nieto).

Características

- Incrementa la flexibilidad y la eficiencia de los programas: nos hacen escribir menos código a la vez que se mejora la abstracción.
- La velocidad del programa puede relentizarse si hay que buscar las funciones asociadas en tiempo de ejecución.

Tipos de herencia

- Herencia simple: sólo hay 1 clase base.
- Herencia múltiple: hay 2 o más clases base.

La herencia múltiple se tiende a sustituir por clases genéricas para eliminar problemas que origina.

Ventajas de la herencia múltiple

- Permite modelar más objetos y más variados.
- Es sencilla, es elegante, es flexible.
- La reutilización de código es mayor.
- Permite modificar las clases sin que por ello se tenga que modificar el interface de la clase derivada.

Inconvenientes de la herencia múltiple

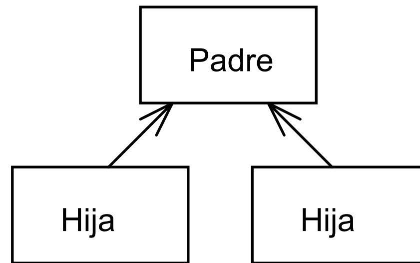
- Confusión y comportamiento impredecible: se puede heredar 2 veces la misma característica y tener métodos y atributos con el mismo nombre.
- Aumento del tiempo de ejecución; debido a que se tienen que resolver colisiones.

Métodos de resolución de colisiones (herencia múltiple)

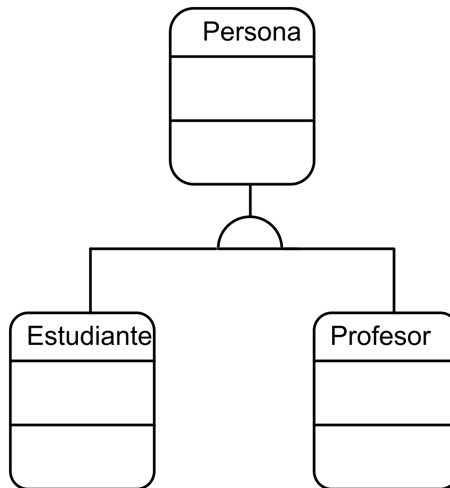
1. **Smalltalk** y **Eiffel** no permiten semánticamente la ambigüedad de datos o métodos. Además **Eiffel** permite renombrar los datos y métodos que se heredan en la clase hija.
2. **CLOS** permite ambigüedad, y es resuelta por el compilador.
3. Con datos repetidos, **C++** obliga a cualificarlos con el nombre de la clase a la que pertenecen.

Notación

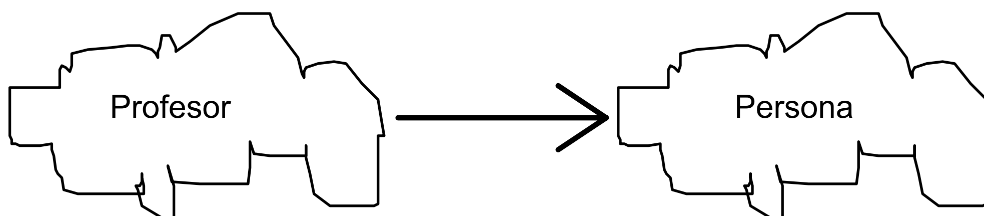
Notación general.



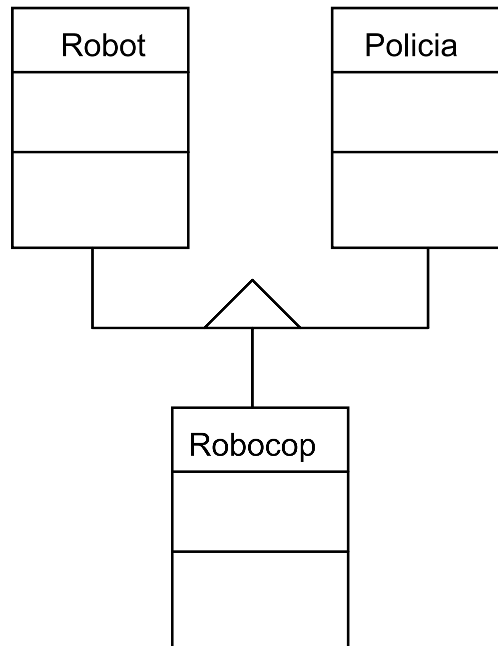
Yourdon, utiliza un semicírculo, ejemplo de herencia simple:



Booch



Rumbaugh, utiliza un triángulo, ejemplo de herencia múltiple:



Relación de Agregación

Representa los Objetos Compuestos. Son relaciones del tipo "tiene_un" o "es_parte_de".

Objeto Contenedor

Es aquel que contiene otros objetos. En la agregación, las clases contienen objetos, y no otras clases.

Rumbaugh, en su método de análisis y diseño, da las siguientes reglas para identificar las relaciones de agregación:

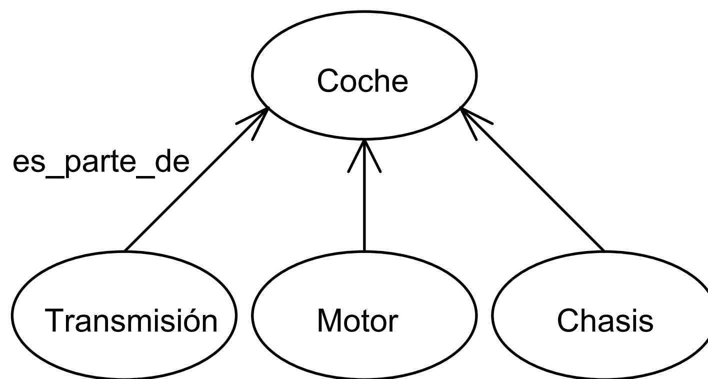
1. Existe una relación de agregación si en el enunciado del problema hay expresiones de la forma "tiene_un", "es_parte_de"...
2. Cuando existe un concepto "todo" y varios conceptos "partes" que dependen del todo, y las operaciones hechas sobre el todo afectan a las partes.

   Ejemplo: Todo: Casa
   Partes: Habitaciones
   Operación: Pintar casa
3. También habrá una relación de agragción si observamos que hay cierta asimetría entre las clases y objetos, si hay cierta subordinación de unos objetos.

Tipos de agregación

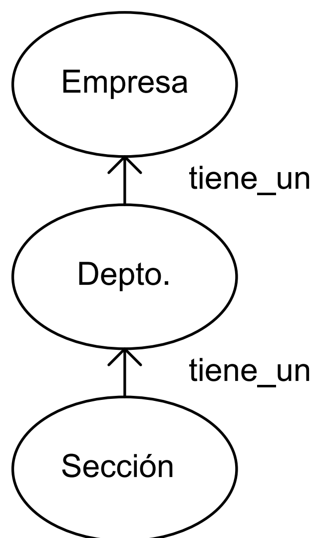
• Por contenido físico o por valor

El contenedor contiene el objeto en sí. Cuando creamos un objeto contenedor, se crean también automáticamente los contenidos.



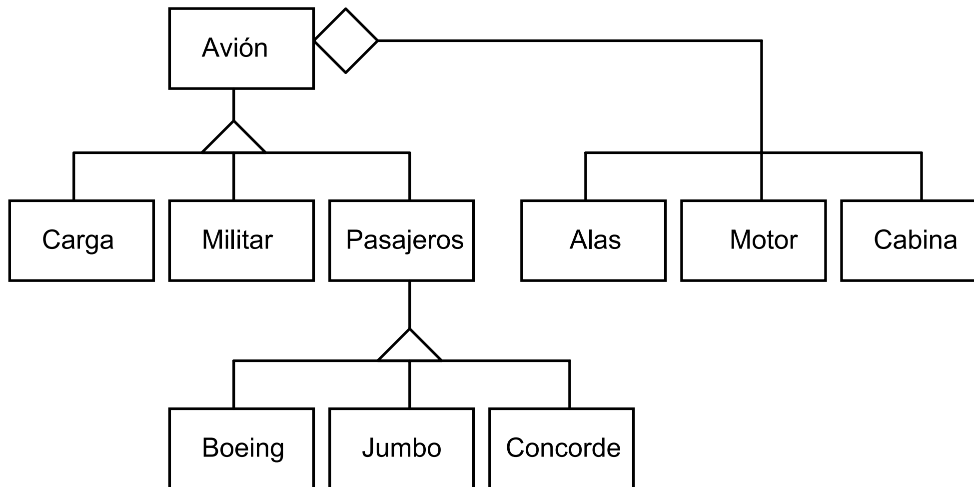
• Agregación conceptual o por referencia

Se tienen punteros a objetos. No hay un acoplamiento fuerte. Los objetos se crean y se destruyen dinámicamente.



Notación Rumbaugh

Ejemplo: Sea "Avión" una clase contenedora, un rombo representa sus relaciones de agregación, y un triángulo las de generalización



Relación de Asociación

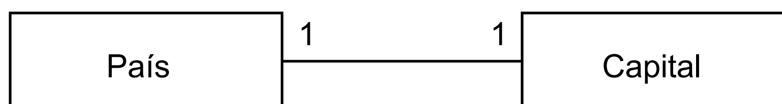
Es una relación entre clases. Implica una dependencia semántica. Son relaciones del tipo "pertenece_a" o "está_asociado_con". Se da cuando una clase usa a otra clase para realizar algo.

Multiplicidad de la Relación

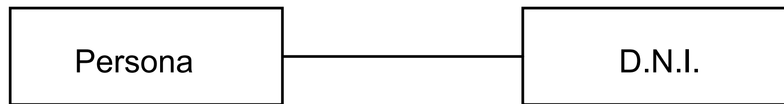
Indica el número de instancias de una clase que se asocian con las instancias de la otra clase. Tipos de multiplicidad:

- **uno_a_uno**

ejemplo (primera notación):

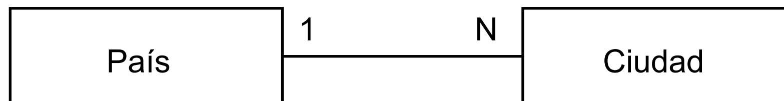


ejemplo (segunda notación):

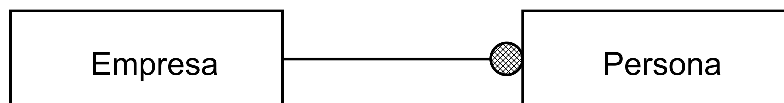


- **uno_a_muchos**

ejemplo (primera notación):

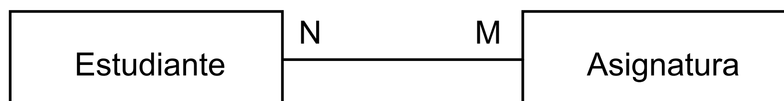


ejemplo (segunda notación):

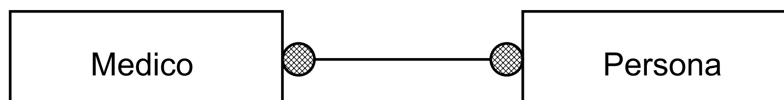


- **muchos_a_muchos**

ejemplo (primera notación):



ejemplo (segunda notación):



Tema 6: Clases y Objetos en C++

Clases y miembros

En C++ una clase es similar a una estructura, de hecho un **struct** es una clase.
Podemos añadir a los elementos de la estructura los prototipos de las funciones miembro que los manipulen. Ejemplo:

```
struct fecha {
    int dia, mes, año;
    void fijar_fecha(int, int, int);
};
```

En la implementación de las funciones se exige calificar el nombre de la función miembro con el nombre de la clase a la que pertenece. Ejemplo:

```
void fecha::fijar_fecha(int d, int m, int a)
{
    dia = d;
    mes = m;
    año = a;
}
```

Para definir un objeto "f" del tipo fecha, basta con una declaración así:

```
fecha f;
```

Para mandar un mensaje al objeto:

```
f.fijar_fecha(3, 2, 1996);
```

Esto plantea un problema desde el punto de vista de la POO: ¡¡todo "f" es visible!!, y operaciones de este tipo estarían permitidas:

```
f.dia = 4;
```

Definición de clases

¿Cómo ocultar la estructura ? Con una estructura **class**. Sintaxis:

```
class <identificador> {
    [ public:
        [ <miembros_dato> ]
        [ <funciones_miembro> ] ]
    [ protected:
        [ <miembros_dato> ]
        [ <funciones_miembro> ] ]
    [ private:
        [ <miembros_dato> ]
        [ <funciones_miembro> ] ]
};
```

Aclaración: Las partes public, protected y private pueden o no aparecer, y no importa su orden.

Notación:

POO	C++
atributo	miembro_dato
método	función_miembro

Miembros de acceso privado

Por defecto, en una clase, todo es privado.

Los miembros sólo pueden ser accedidos por:

i Las funciones miembro de la propia clase

i Las funciones amigas de la propia clase

Su definición está delimitada por:

Desde...	Hasta...
"{"	**public**: ó **protected**: ó **"}"**

Miembros de acceso protegido

Los miembros sólo pueden ser accedidos por:

- Las funciones miembro de la propia clase
- Las funciones amigas de la propia clase
- Las funciones miembro de clases derivadas

Su definición está delimitada por:

Desde...	Hasta...
protected:	**public**: ó **private**: ó **"}"**

Miembros de acceso público

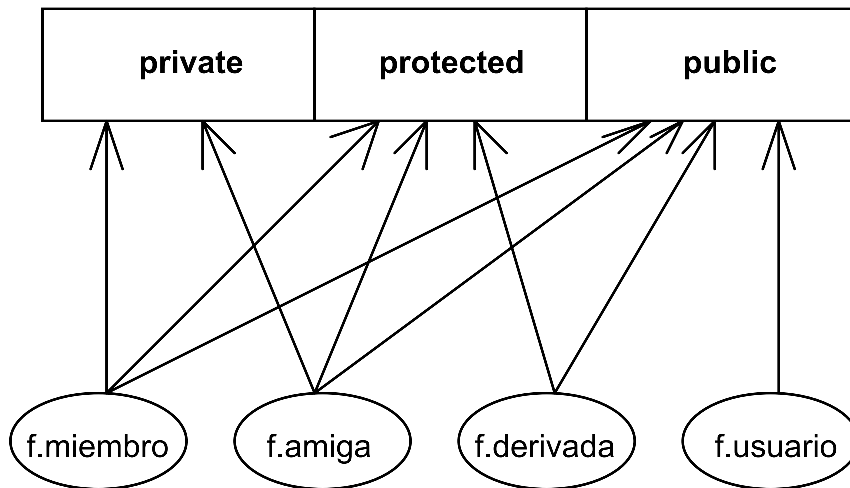
Los miembros pueden ser accedidos por:

- Las funciones miembro de la propia clase
- Las funciones amigas de la propia clase
- Las funciones miembro de clases derivadas
- Cualquier usuario de la clase

Su definición está delimitada por:

Desde...	Hasta...
public:	**private**: ó **protected**: ó **"}"**

Resumen de los modos de acceso a miembros



Ejemplo:

```
class fecha {
    int dia, mes, año; //privado
public:
    void fijar_fecha(int, int, int);
}
```

La única manera de asignar valores a un objeto fecha ahora es a través de la función pública (interface del objeto). Paso de mensajes:

```
fecha f;
f.fijar_fecha(3, 2, 1996);
```

Reglas de ocultación

1. Los miembros_dato que no se quiere que se modifiquen desde el exterior, deberían ser privados o protegidos.
2. Los constructores/destructores deben ser públicos, en la mayoría de las ocasiones.
3. Las funciones que sean usadas para la manipulación interna de datos o que son invocadas por funciones de la propia clase deben ser privadas o protegidas.
4. El resto será público.

Tipos de funciones miembro definibles en una clase

1. Sencillas: normales, nada especial.
2. Estáticas: acceden a los miembros estáticos de una clase.
3. Constantes: devuelven constantes.
4. En Linea.
5. Especiales: constructores y destructores.
6. Virtuales.

Las funciones amigas no son miembros de la clase de la cual son amigas.

Funciones en linea (inline)

Las llamadas a las funciones de tipo **inline** son sustituidas por el código de la propia función, es decir, no se produce cambio de contexto.

Funcionan de forma similar a las MACROS, pero en las funciones **inline** se evalúan realmente los parámetros.

Es especialmente útil en funciones cuya implementación es 1 sola línea, ya que ahorramos tiempo de ejecución al no haber cambios de contexto.

Dentro de las clases, se pueden definir de dos formas:

Incluyendo el cuerpo entre llaves tras la declaración, indicamos al compilador que dé tratamiento "en linea" a las llamadas a esa función:

```
class ejemplo {
    <...>
public:
    tipo funcion(parametros) { <cuerpo>; }
}
```

Con la palabra reservada inline:

```
class ejemplo {
    <...>
public:
    inline tipo funcion(parametros);
}

//restricción: la implementación debe ponerse en
//el mismo archivo donde se define la clase

inline tipo ejemplo::funcion(parametros)
{
    <cuerpo>;
}
```

Funciones amigas (friend)

Las funciones amigas NO SON FUNCIONES MIEMBRO. El prototipo estará definido dentro de la clase, pero con la palabra reservada **friend**. La definición de la función puede estar en cualquier sitio. Ejemplo:

```
class ejemplo {
    <...>
    friend tipo funcion(parametros);
}
```

No es deseable abusar de las funciones amigas. El uso fundamental que tendrán será el de poder realizar operaciones con objetos de 2 clases distintas. Por ejemplo (multiplicar una matriz por un vector):

```
class vector {
    float v[4];
public:
    float &elemento(int);
    //devuelve la dirección de uno de los elementos
};

class matriz {
    vector m[4];
public:
    float &elemento(int, int);
};

vector multiplicar(const matriz &m, const vector &v)
{
    vector r;
    int i, j;

    for (i=0; i<3; i++)
         r.elemento(i) = 0.0;

    for (i=0; i<3; i++)
         for (j=0; j<3; j++)
              r.elemento(i) +=    m.elemento(i,j) *
                                  v.elemento(k);
    return r;
}
```

Una forma de mejorar la eficiencia es disminuir el número de llamadas, accediendo a los miembros_dato de las clases con una función amiga:

```
class matriz;       //declaración incompleta necesaria

class vector {
    float v[4];
public:
    float &elemento(int);
    friend vector multiplicar(    const matriz &,
                             const vector &);
};

class matriz {
    vector m[4];
public:
    float &elemento(int, int);
    vector matriz::mult(     const matriz &,
                             const vector &);
};
```

```
vector matriz::multiplicar(const vector &v)
{
    vector r;
    int i, j;

    for (i=0; i<3; i++)
         r.elemento(i) = 0.0;

    for (i=0; i<3; i++)
         for (j=0; j<3; j++)
              r.v[i] += m[i][j] * v.v[j];
    return r;
}
```

Constructores y destructores

Mecanismo de C++ para inicializar y liberar la memoria asociada al objeto. Son clases miembro especiales, que se diferencian en su sintáxis y en que son invocadas implícitamente (aunque también explícitamente).

Constructores

Miembro de la clase que captura la mempria asociada al objeto.

Características

- Tienen en mismo nombre que la clase
- No devuelven valores
- Pueden tener parámetros (y valores por defecto)
- Pueden ser definidos en linea o normalmente
- Pueden existir más de un constructor para una clase

Cuando declaramos un objeto de una clase, el constructor es invocado automáticamente.

Si no definimos uno, el compilador pone el suyo por defecto.

```
class circulo {
    float x_centro, y_centro;
    float radio;
public:
    circulo(float x, float y, float r)
    { x_centro = x; y_centro = y; radio = r }
};

circulo c;              //error faltan parámetros
circulo c(2, 3, 1.5);   //OK!
circulo c = circulo(2, 3, 1.5);   //OK!


class fecha {
    int dia, mes, año;
public:
    fecha(int, int, int);    //dados dia, mes, año
    fecha(int d, int m);     //dados dia y mes
    fecha(int d);            //dado el dia
    fecha();                 // sin parámetros
};

class fecha {
    int dia, mes, año;
public:
    fecha(int d = 1, int m = 1, int a = 1996);
};

fecha primer_dia_del_año;
fecha examen_c_mas_mas(20, 4);
fecha segundo_dia_del_año(2);
fecha hoy(19, 4, 1996);
```

Constructores de copia

Crean un objeto a partir de otro que ya existe. Se invócan implícitamente cuando se hacen una de estas cosas:

- Asignaciones entre objetos
- Paso de parámetros-objeto por valor

Por defecto, C++ pone un constructor de copia por defecto que copia byte a byte el objeto en sí, pero que da problemas cuando tenemos datos dinámicos: hay duplicación de punteros pero no de la información apuntada.

Se caracterizan por tener como parámetro una referencia constante a un objeto de la clase a la que pertenecen.

```
class complejo {
    double re, im;
public:
    complejo(const complejo &c)
    {
    //"c" es el objeto a la derecha de la asignación

         re = c.re; im = c.im;
    }
    complejo(double r=0.0, double i=0.0)
    { re = r; im = i; }
};

complejo c1;
complejo c2 = c1; //invoca al constructor de copia
```

Destructores

Miembro de la clase que libera la mempria asociada al objeto de la clase. Se invocan implícitamente cuando acaba el ámbito del objeto. Pueden ser invocados explícitamente con el operador **delete**. No tienen sentido si no hay memoria dinámica en el objeto.

Características

- Tienen en mismo nombre que la clase precedido de tilde "~"
- No devuelven valores
- No pueden tener parámetros
- Pueden ser definidos en linea o normalmente
- En una clase puede existir como máximo un constructor.

```
class pila_char {
    int tam;
    char *pila;
    char *cima;
public:
    pila_char(int t)
    { cima = pila = new char[tam = t]; }

    ~pila_char()
    { delete []pila; }

    void meter(char c) { *cima++ = c; }
    char sacar()       { return *--cima; }
};

void f()
{
    pila_char p(3);
}
//aquí se invoca el destructor de la pila

void g()
{
    pila_char p(3);
    //...
    delete p;
    // aquí se invoca el destructor de la pila
}
```

Autoreferencia (puntero "this")

```
class ejemplo {
    int n;
public:
    int leer() { return n; }
};
```

Toda función miembro de una clase C++ recibe un argumento implícito: un puntero al objeto que la invoca.

```
ejemplo * const this; //así sería la declaración
```

Const ahí hace que se pueda modificar el contenido apuntado pero no la dirección.

Ejemplos de uso:

```
int leer() { return this->n; }

void X::g(X &a, X &b)
{
    a = *this;
    *this = b;
}
```

this	puntero al objeto
*this	contenido del objeto
this->miembro	acceso a miembro

Una función amiga no tiene puntero **this**.

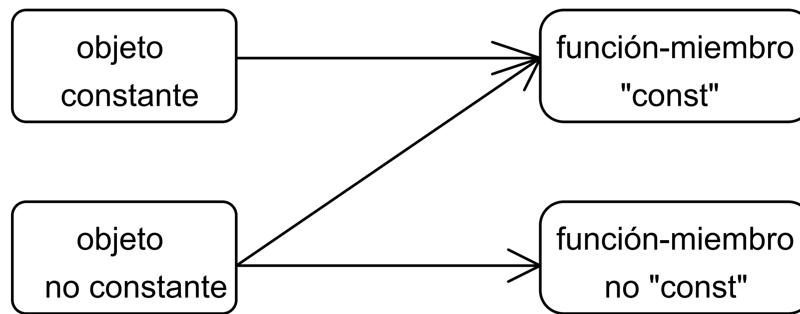
```
class doble_enlace {
    //..
    doble_enlace *prev;
    doble_enlace *sig;
public:
    void anexar(doble_enlace *);
};

void doble_enlace::anexar(doble_enlace *p)
{
    p->sig = sig;
    p->prev = this;
    sig->prev = p;
    sig = p;
}
```

Funciones "const"

Una función-miembro "const" no puede modificar el objeto para la cual ha sido llamada. Se declara poniendo la palabra reservada "const" como sufijo del prototipo.

```
class X {
    int m;
public:
    int leer() const { return m; }
    void escribir(int i) { m = i; }
};
```

No pueden ser invocadas por cualquier objeto:



Una función-miembro "no-const" no puede nunca ser invocada por un objeto constante:

```
void f(X &alterable, const X &constante)
{
    int j;
    j = constante.leer;         //OK!!
    j = alterable.leer;         //OK!!
    constante.escribir(j);      //***ERROR***
    alterable.escribir(j);      //OK!!
}
```

Clases anidadas

Es posible en C++ anidar clases de forma análoga al anidamiento de estructuras:

```
class X {
    class Y {
         int m;
    }
public:
    class Z {
         int m;
    }
    Y f(Z);
};

Y y1;          //***ERROR*** por no ambito
X::Y y1;       //***ERROR*** por ser privada
Z z1;          //***ERROR*** por no ambito
X::Z z1;       //OK!!
```

Objetos de clases como miembros

```
class ejemplo {
    int tam;
    tabla t;
    //...
public:
    ejemplo(int);
    ~ejemplo();
};

ejemplo::ejemplo(int t):t(tam)
{
    tam = t;
}

ejemplo e(10);
```

Primero invoca el constructor del objeto contenido (t de la clase "tabla") y luego al constructor del objeto contenedor (de la clase "ejemplo").

Implementación del constructor de la clase contenedora:

clasecontenedora::clasecontenedora(parametrosformales:lista_de_objetos) {}
lista_de_objetos ::= nombreobjeto(expresion) { , nombreobjeto(expresion) }*

Array de objetos o Punteros a clases

Hay que definir en la clase necesariamente un constructor con valores por defecto, ya que en la declaración de arrays y punteros no hay forma sintáxtica para poder pasar valores de inicialización.

```
tabla t[100];
tabla *t1, *t2;
t1 = new tabla;
t2 = new tabla[10];
```

Miembros estáticos

Son miembros que van a ser compartidos por todos los objetos de la clase. Las reglas de visivilidad se siguen cumpliendo. Sólo se pueden referenciar através del nombre de la clase:

```
class E {
    int x;
public:
    static int valor;
    static void f();
};

E::valor = 3;
```

Las funciones estáticas son compartidas por la clase como función miembro, pero no tienen asociado puntero "this". La función sólo puede acceder a los miembros estáticos de la clase: x no puede ser accedido desde f().

Las se cualifican con el nombre de la clase, aunque se permite también usar el nombre del objeto.

```
E e;
E::f();  //OK!!
e::f();  //no daría error
```

Clases amigas

Cuando todas las funciones de una clase X son amigas de la clase Y, se dice que X es una clase amiga de Y.

```
class X {
    //...
    void f();
    void g();
};

class Y {
    friend void f();
    friend void g();
};
```

Se puede entonces hacer una declaración equivalente de esta forma:

```
class Y {
    friend class Y;
}
```

Sobrecarga de operadores y conversión de tipos

Los operadores sólo se pueden sobrecargar para operendos que sean clases.

Reglas

1. Todo operador C++ puede ser sobrecargado excepto:

.	acceso a miembro
. *	referencia puntero a miembro
: :	ambito
? :	condicional
#	preprocesador

2. Un operador puede ser sobrecargado con funciones ordinarias, amigas o miembro de una clase.
3. No se puede cambiar la precedencia y la asociatividad de los operadores.
4. No se puede cambiar la anidad (nº de operandos) de los operadores
5. La sobrecarga de los siguientes operadores sólo puede hacerse mediante funciones-miembro:

( )	llamada
[ ]	array
=	asignación
->	acceso a miembro desde puntero

6. Los operadores sobrecargados no pueden tener argumentos por defecto

Sintáxis para sobrecargar un operador

```
tiporetorno operator@ (lista_de_argumentos)
```

donde:

@	Es cualquier operador permitido
lista_de_argumentos	Es la lista de operandos
operator	Es una palabra reservada

Operadores binarios: aa @ bb

- **Función-miembro**

  Tiene 1 argumento, el operando derecho.

  El operando izquierdo es this.

```
aa.operator@(bb);
```

- **Función no-miembro**

  Tiene 2 argumentos, el primero es el operando izquierdo.

  El segundo es el operando derecho.

```
operator@(aa, bb);
```

Operadores unarios prefijos: @ aa

- **Función-miembro**

  No tiene argumentos

```
aa.operator@();
```

- **Función no-miembro**

  Tiene 1 argumento: el operando.

```
operator@(aa)
```

Operadores unarios postfijos: aa@

En la lista de argumentos se añade uno instrumental, de tipo entero, el cual no se utilizará, y es tan sólo una marca para el compilador. Por supuesto, no se requiere en la llamada.

• **Función-miembro**

```
tipo operator++(int); //int es instrumental
```

• **Función no-miembro**

```
tipo operator++(tipo, int); //int es instrumental
```

```
class X {
    //...
    X *operator&();      //sobrecarga el op. dirección
    X operator&(X);      //sobrecarga el AND de bits
    X operator++(int);  //sobrecarga el postfijo++
    X operator+(X);      //sobrecarga el + binario
};

X operator-(X);          //sobrecarga el - unario
X operator-(X, int);     //sobrecarga X - entero

class tres_d {
    int x, y, z;
public:
    tres_d operator+(tres_d);
    tres_d operator=(tres_d);     //siempre =miembro
    tres_d operator++();
};

tres_d tres_d::operator+(tres_d t)
{
    tres_d temp;
    temp.x = x + t.x;
    temp.y = y + t.y;
    temp.z = z + t.z;
    return temp;
}
```

```
tres_d tres_d::operator=(tres_d t)
{
    x = t.x;
    y = t.y;
    z = t.z;
    //para permitir sentencias como: a=b=c;
    return *this;
}

tres_d tres_d::operator++()
{
    ++x;
    ++y;
    ++z;
    return *this;
}
```

Cuando usamos funciones amigas para la sobrecarga de operadores unarios, necesariamente el argumento hay que pasarlo por referencia:

```
tres_d operator++(tres_d &t, int)
{
    t.x++;
    t.y++;
    t.z++;
    return *this;
}
```

Si no hubiera pasado "t" por referencia, el retorno sería correcto, pero "t" no quedaría modificado, sino su copia.

Para mezclar operandos de distintas clases, donde el operando izquierdo es por ejemplo un entero y el derecho un objeto de tipo X, es necesario emplear funciones amigas:

```
class complejo {
    double re, im;

friend complejo operator+(complejo, complejo);
friend complejo operator+(complejo, double);
friend complejo operator+(double, complejo);
//...
};
```

Si queremos evitar el tener que implementar cientos de funciones para las operaciones aritméticas sobre complejos con doubles, utilizaremos un constructor especial.

Constructor de conversión

Tiene un parámetro del tipo que queremos convertir:

```
class complejo {
    double re, im;
public:
    complejo(double f) { re = f; im = 0.0; }
    complejo operator=(complejo &);

friend complejo operator+(complejo, complejo);
};

complejo c;
double x;

c = c + x;    //construye c + complejo(x, 0)
c = x;        //sería correcto
x = c;        //***ERROR***
```

Una solución al error producido por "x=c" sería...

Operador de conversión de tipos

operatorT( )

donde "T" es el nombre del tipo al que queremos hacer la conversión.

- Un operador de conversión siempre tiene que ser función miembro de la clase a la que va a ser convertida.
- No debe especificar ningún tipo de retorno.
- No debe tener argumentos.

```
class complejo {
    //...
    operator double() { return; }
    //se implementa con un return
};

complejo c;
double x;

x = c; //OK!!
```

Ejemplo:

```
class Polar;

class Rect { //coordenadas rectangulares
    double xco, yco;
public:
    //Este constructor de conversión
    //hace lo mismo que el
    //operador de conversión
    Rect(Polar p)
    {
         xco=p.radio * cos(p.angulo);
         yco=p.radio * sin(p.angulo);
    }
};

class Polar { //coordenadas polares
    double radio, angulo;
public:
    //Este operador de conversión
    //hace lo mismo que el
    //constructor de conversión
    //de la clase Rect
    operator Rect()
    {
         double x = radio * cos(angulo);
         double y = radio * sin(angulo);
         return Rect(x, y);
    }
};

Rect r;
Polar p;

r = p;   //primero convierte, luego asigna
```

Si lo vamos a usar a menudo conviene...

Sobrecarga del operador de Asignación-Conversión

Este operador tiene como argumento la clase ajena.

```
class Rect {
    double xco, yco;
public:
    Rect operator= (Polar p)
    {
         xco = p.radio * cos(p.angulo);
         yco = p.radio * sin(p.angulo);
         return *this;
    }
};
```

Este es un buen método para evitar crear objetos temporales: copias extra.

NOTA: El constructor de copia sólo es invocado cuando hay una asignación en la declaración del objeto:

```
Rect r1;
Polar p1;
Rect r = r1;  //OK!!
Rect r = p1;  //***ERROR**
```

"Rect r = p1" producirá un error si no tengo un operador de conversión o un constructor de conversión, ya que se necesita la conversión antes de construir la copia.

Operadores de Inserción y Extracción

i No pueden ser funciones-miembro de una clase.
i Han de ser funciones amigas (si es que acceden a miembros privados).
i Devuelven una referencia a flujo.

Sintáxis:

```
ostream &operator<<(ostream &, clase &);
istream &operator>>(istream &, clase &);
```

Como el primer argumento no es la clase, no se pueden implementar como funciones-miembro.

```
class tres_d {
    int x, y, z;
public:
    //...
friend ostream &operator<<(ostream &, tres_d &);
friend istream &operator>>(istream &, tres_d &);
};

ostream &operator<<(ostream &os, tres_d &td)
{
    os << td.x << ", ";
    os << td.y << ", ";
    os << td.z << endl;
    return os; //permite encadenar varios "<<"
}

istream &operator>>(istream &is, tres_d &td)
{
    is >> td.x >> td.y >> td.z;
    return is; //permite encadenar varios ">>"
}

tres_d t1, t2;
cin >> t1;
cout << t1 << t2;
```