UD1: El lenguaje de programación C++

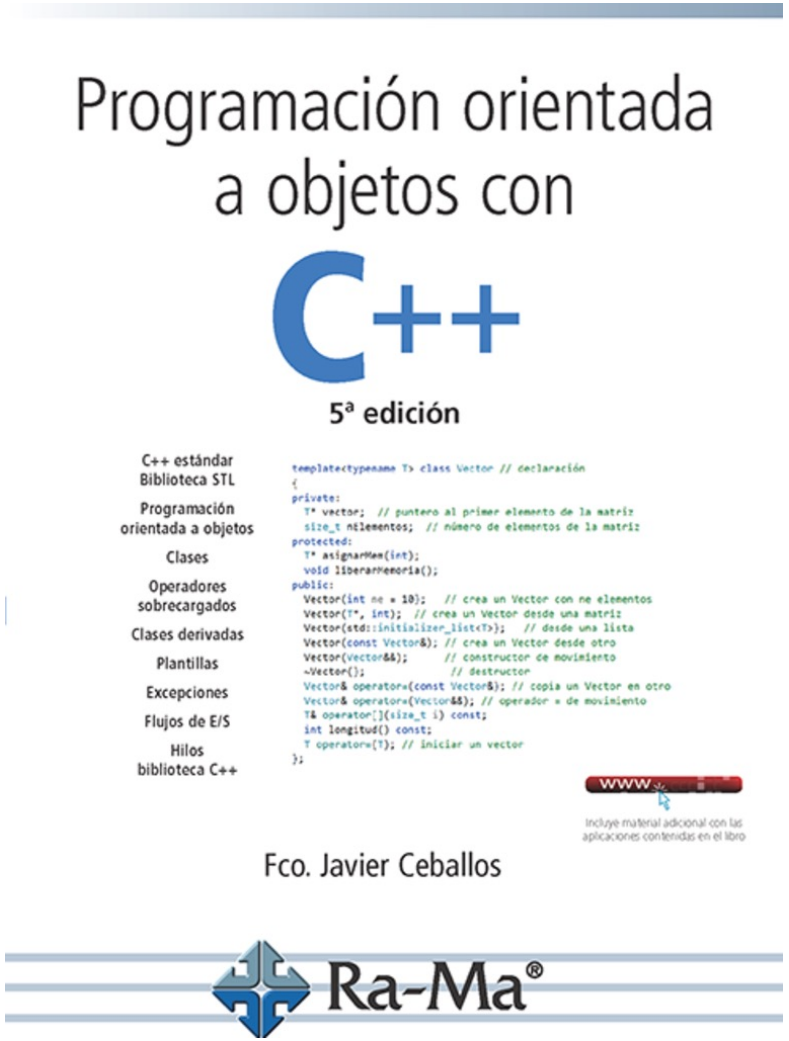
Tema 3: Programación Orientada a Objetos

« Me lo explicaron y lo olvidé, lo vi y lo aprendí, lo hice y lo entendí.»

- Confucio -

Bibliografía Ceballos

- Capítulos 2, 5, 6 y 7, 9
  - Los capítulos 2 y 5 introducen la orientación a objetos
  - El capítulo 6 trata sobrecarga de operadores (se trata por encima en la asignatura)
  - El capítulo 7 describe la herencia en la OO
  - El capítulo 9 trata la gestión de excepciones



Ejemplo: abstracción y encapsulamiento

Personaje de un juego de aventuras	Post en redes sociales
¿Atributos (valores/variables)?	¿Atributos (valores/variables)?
¿Operaciones (funciones, métodos)?	¿Operaciones (funciones, métodos)?

Resto del juego	Resto de la red social
¿Qué más clases harían falta? ¿Qué relaciones guardan entre ellas?	¿Qué más clases harían falta? ¿Qué relaciones guardan entre ellas?

¿Es necesario que el programador sepa exactamente cómo funciona cada parte del código?
¿Es necesario que todas las variables sean accesibles por cualquier código?

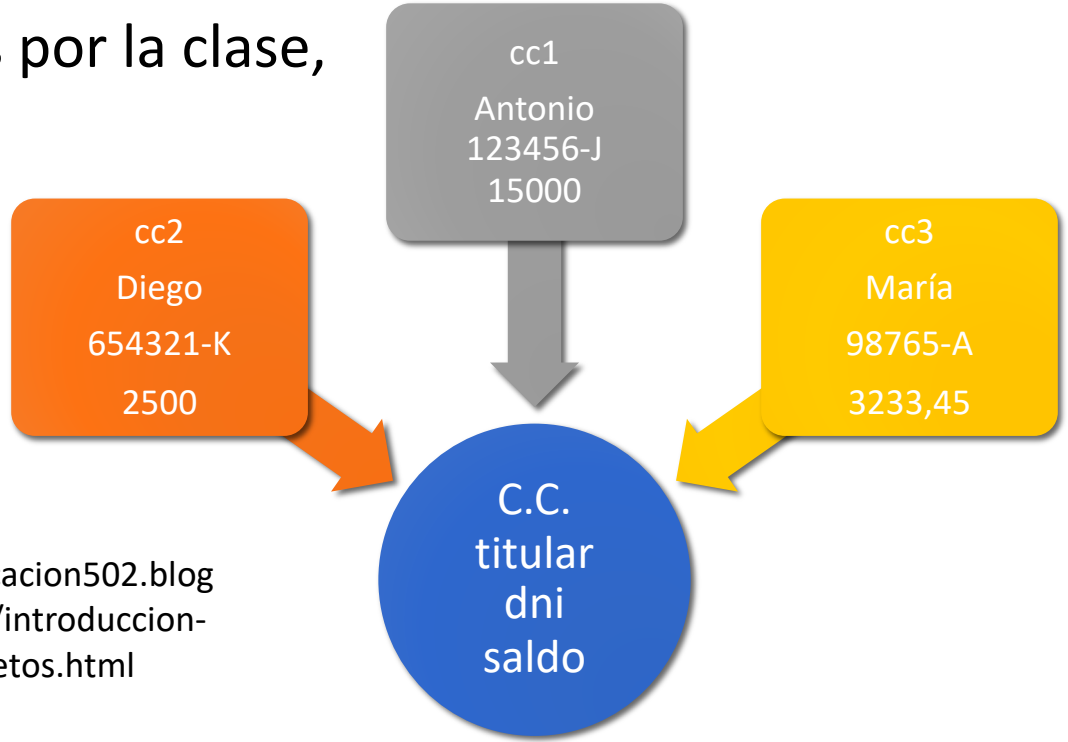
Se entra en el complicado mundo del **DISEÑO**

Conceptos fundamentales de OO (I)

Un programa es un conjunto de objetos que se comunican entre sí mediante mensajes. Una clase agrupa datos junto a las funciones (métodos) para modificarlos

- **Clase**: definición de las propiedades y comportamiento de un nuevo tipo de dato, como si fuera un molde. La creación de un objeto a partir de una clase se llama "instanciación"
- **Objeto**: instancia concreta y diferenciable de una clase. Tiene las propiedades (variables) definidas por la clase, pero con distintos valores

https://cienciaeducacion502.blogspot.com/2014/01/introduccion-las-clases-y-los-objetos.html



Conceptos fundamentales de OO (II)

- **Método**: algoritmo asociado a una clase, ejecutado tras la recepción de un mensaje. Representan lo que la clase puede hacer. Un método puede cambiar las propiedades del objeto, realizar un cálculo o generar mensajes para otros objetos. Son funciones, algunas con propósito particular (constructores, destructor)

- **Mensaje**: invocación de un método de un objeto por parte de otro

- **Propiedad o atributo**: variables (de cualquier tipo de dato, incluido otra clase) que almacenan el estado del objeto. Es posible controlar su visibilidad y accesibilidad: públicas o privadas. Generalmente son privadas, solo modificables mediante métodos definidos en la clase

Conceptos fundamentales de OO (III)

- **Herencia**: facilidad por la cual se pueden declarar nuevas clases a partir de otras ya existentes, heredando atributos y métodos. Es luego posible añadir nuevos atributos y métodos, así como sobrescribir los métodos heredados
- **Polimorfismo**: capacidad de tratar de manera homogénea (por ejemplo, almacenarlas en un vector) un conjunto de objetos instancias de clases que guardan entre sí una relación de herencia, pero luego cada uno ejecuta el método concreto definido en su clase concreta
- **Composición**: cuando alguna de las propiedades de una clase tiene como tipo otra clase

```
class Vehiculo {
 ...
}
```

```
class Rueda {
 ...
}
```

Herencia

```
class Coche : public Vehiculo {
Rueda r1, r2, r3, r4;
 ...
}
```

Composición

Anatomía de una clase

- Una definición de clase introduce un nuevo tipo de dato en el programa

```
class Nombre {
public:
  // métodos (funciones) y atributos (variables) de clase y objeto
protected:
  // métodos (funciones) y atributos (variables) de clase y objeto
private:
  // métodos (funciones) y atributos (variables) de clase y objeto
}; // ojo al punto y coma final!
```

- **public**, **protected** y **private** controlan la visibilidad de métodos y atributos. Es decir, qué objetos tienen acceso a ellos: **public** → cualquiera; **protected** → solo desde clases hijas (herencia); **private** → solo objetos de la clase
- Se puede utilizar **class** o **struct**. La única diferencia es la visibilidad por defecto: **class** → **private**; **struct** → **public**
- Atributos y métodos de una clase tienen ámbito de clase: son visibles entre ellos independientemente del punto en que estén declarados dentro de la clase

Atributos de una clase/objeto

- **Atributos de clase**: valores compartidos por todos los objetos de una clase. Se declaran como variables `static` dentro de la clase
- **Atributos de objeto**: valores individuales de cada objeto que almacenan su estado. Se declaran como variables no `static` dentro de la clase. Son los habituales
- Salvo razón de peso, ambos tipos de atributo se declaran como `private`
  - Es habitual que alguna característica de la clase esté representada por varios atributos que deben ser modificados de forma coordinada (e.g., número DNI y letra)
  - O que no todos los valores sean adecuados (e.g., la edad no puede ser negativa)
- Haciendo los atributos privados nos aseguramos de que solo van a poder ser modificados cuándo y cómo nosotros decidamos:
  - Solo mediante funciones que proporciona el desarrollador (lo habitual)
  - Solo de manera interna (sin que otros objetos externos puedan solicitarlo)
  - Solo cuando se construye el objeto (*objetos inmutables*)

Código: ejemplo atributos (de objeto)

```
class C_Punto {
private:
  double x_, y_;
};
```

```
class C_Num_Complejo {
  double re_;
  double img_;
};
```

Convención: nombres de clase empiezan por **C_** y atributos terminan con **_**

Sé que `std::vector<C_Punto>` es mejor

```
class C_Segmento {
  C_Punto ini_, fin_;
  double longitud_, ang_;
};
```

¿Existe alguna relación entre los atributos de tipo `C_Punto` y los de tipo **double**?

Invariante

```
class C_Malla {
  C_Punto centro_;
  C_Punto puntos_[100];
  int num_puntos_;
};
```

¿Existe relación entre `puntos_` y `num_puntos_`?

Funciones dentro de una clase (I)

```
mods1 tipo_dato_retorno id_funcion (lista_parametros) mods2
```

opcional — obligatorio — opcional

- Todas las funciones declaradas dentro de una clase, salvo las que tienen *mods1* {**`static`** o **`friend`**}, se denominan *funciones miembro o métodos* (Java) y se invocan utilizando la notación `obj.metodo()`. Los métodos públicos constituyen la **interfaz de la clase**
- mods1:
  - **`virtual`**: indica que la función es polimórfica (usado en herencia)
  - **`friend`**: la función tiene acceso a la parte **`private`**/**`protected`** sin ser función miembro
  - **`static`**: indica que la función es de clase, no de objeto
  - **`explicit`**: se utiliza en constructores de un parámetro, para indicar que solo se puede utilizar al construir un objeto y no en conversiones implícitas
- mods2:
  - **`const`**: indica que la función no modifica atributos del objeto, y por tanto se puede invocar en objetos constantes (o que se han pasado mediante una referencia constante)
  - **`final`**: indica que la función no puede ser sobrescrita (usado en herencia)
  - **`override`**: indica que esta función debe sobrescribir una función **`virtual`** (usado en herencia)

No son palabras reservadas

Funciones dentro de una clase (II)

- Las funciones **`static`** son funciones de clase y se invocan utilizando la sintaxis `nombreClase::funcion()`. Solo tienen acceso a funciones y atributos de clase
- Desde una función miembro (de objeto) se tiene acceso a todo el contexto estático de la clase (atributos y funciones), pero no viceversa
- Las funciones **`friend`** tienen acceso al contenido de una clase sin formar realmente parte de su interfaz. Se utiliza mucho para sobrecargar *operadores* (`+`, `!=`, `<<`, etc.). La visibilidad de las funciones **`friend`** es siempre **`public`**
- Son funciones miembro: constructores, destructor y funciones normales
  - **Constructores**: definen cómo se instancia un objeto de una clase, i.e., con qué valores. Constructor sin argumentos (*constructor por defecto*), constructores delegados (utilizan otros constructores de la clase), constructores "normales"
  - **Destructor**: solo se puede definir uno; especifica cómo se terminan los objetos de una clase. Salvo que se reserve memoria dinámicamente (con **`new`**) o se esté definiendo una jerarquía de clases (en herencia), no es necesario preocuparse por él

Código: ejemplo atributo de clase (static)

```
class C_Punto {
  static int n_puntos_;
  double x_, y_; // atributo de objeto
public:
  static int get_n_puntos() { return n_puntos_; }

  C_Punto () { ++n_puntos_; }
};

int C_Punto::n_puntos_ = 0;

int main() {
 C_Punto a, b; // ctor
 std::cout << "Se han creado: " << C_Punto::get_n_puntos() << '\n';
 C_Punto c, d;
 std::cout << "Se han creado: " << C_Punto::get_n_puntos() << '\n';
 return 0;
}
```

atributo de clase

método de clase

constructor

*inicialización de atributo de clase (obligatorio). Excepción a la regla de que no se puede acceder a los elementos **private** de una clase*

invocación de método de clase

```
Se han creado: 2
Se han creado: 4
```

Dos ficheros: cabecera (.h) e implementación (.cpp)

c_punto.h

```
#ifndef C_PUNTO_H // evita 'n' inclusiones
#define C_PUNTO_H
class C_Punto {
  double x_, y_;
public:
    double get_x() const { return x_; }
    double get_y() const;
    void desplazar_rel (const double d_x,
            const double d_y);
    void desplazar_abs (const double x,
            const double y)
        { x_ = x; y_ = y; }
    void solapar (const C_Punto &p)
        { x_ = p.x_; y_ = p.get_y(); }
}; // ojo al ';' final!!!
#endif
```

Interfaz

c_punto.cpp

```
#include "c_punto.h"

double C_Punto::get_y() const { return y_; }

void C_Punto::desplazar_rel (const double d_x,
 const double d_y) { x_ += d_x; y_ += d_y; }
```

main.cpp

```
#include "c_punto.h"
int main () {
    C_Punto a, b, c;
    a.desplazar_rel (1.1, 2.2);
    a.desplazar_abs (b.get_x(), b.get_y());
    c.solapar(a);
}
```

- Las funciones "cortas" se suelen implementar en el .h
- Módulos (C++20) al rescate

Código: ejemplo constructor (I)

- El compilador sabe cómo inicializar los tipos de datos que proporciona el lenguaje, pero desconoce cómo hacerlo con los tipos definidos por el usuario
- Los objetivos del *constructor* de una clase es construir objetos correctamente:
  1. Creando primero cada una de las partes (atributos) que lo forman
  2. Estableciendo el **invariante de la clase**: propiedad/es de la clase que debe/n mantenerse siempre, aunque se invoquen funciones (e.g., número y letra NIF)
- El constructor es una función cuyo nombre coincide con el de la clase y no retorna valor. Tiene dos partes: *inicializador* (opcional, pero necesario) y el código final de inicialización

inicializador

código constructor

```
class C_Punto {
  double x_, y_; // atributos
public:
  C_Punto (double x, double y) : x_{x}, y_{y} {} // constructor normal
  C_Punto () : C_Punto{1.0, 2.0} {} //constructor por defecto, que delega en otro
  C_Punto (double x) : x_{x}, y_{0.0} {}
  // mejor alternativa, delegando: C_Punto (double x) : C_Punto{x, 0.0} {}
};
```

Inicializador y constructor

- El *inicializador* permite especificar con qué valores se va a invocar el constructor de cada uno de los atributos de una clase
- Sin inicializador, se invocan *constructores por defecto*. Esto no es necesariamente malo, pero como regla general, es mejor utilizar siempre el inicializador
- El inicializador es, además, la única forma de inicializar un atributo constante
- Los atributos se deben inicializar siempre en el mismo orden en que aparecen en la definición de la clase. Eso es lo que hace el constructor por defecto
- Cuando se invoca el código del constructor, todos los atributos de la clase han sido ya construidos. Por eso, muchas veces se deja vacío el bloque { }

```
class C_Punto1 {
  double x_, y_;
public:
  C_Punto1 (double x, double y)
   : x_{x}, y_{y} {}
};
```

con inicializador

```
x_ = x; y_ = y;
```

```
class C_Punto2 {
  double x_, y_;
public:
  C_Punto2 (double x, double y)
   { x_ = x; y_ = y; }
};
```

sin inicializador

```
1) x_ = 0; y_ = 0;
2) x_ = x; y_ = y;
```

Constructor delegado y valores por defecto

- Es buena práctica de programación tener un constructor completo y proporcionar constructores con menos parámetros, que delegan en el complejo para inicializar correctamente el objeto, en lugar de repetir el código del inicializador (máxima de programación *Don't Repeat Yourself*)
- Un constructor (y cualquier función, realmente) puede tener cualquier número de parámetros con valores por defecto, pero deben aparecer los últimos
- Si no se define ningún constructor, el compilador crea uno 'por defecto'

```
class C_Punto1 {
  double x_, y_;
public:
  C_Punto1 (double x, double y)
   : x_{x}, y_{y} { }
  C_Punto1 () // ctor por defecto
   : C_Punto {0.0, 0.0} { }
};
```

```
class C_Punto2 {
  double x_, y_;
public:
  C_Punto2 (double x=0.0, double y=0.0)
   : x_{x}, y_{y} { }
};
```

Ambas clases proporcionan un constructor por defecto (C_Punto2 de forma indirecta)

Código: ejemplo constructor (II)

```
class C_Segmento {
  C_Punto ini_, fin_;
  double longitud_, ang_;
public:
  C_Segmento (double x1, double y1, double x2, double y2)
   : ini_ {x1, y1}, fin_ {x2, y2}, ang_{atan2(y2-y1, x2-x1)},
    longitud_{hypot(x1-x2, y1-y2)} { }
  C_Segmento (double x2, double y2) : C_Segmento {0.0, 0.0, x2, y2} { }
  C_Segmento (C_Punto ini, C_Punto fin) : ini_{ini}, fin_{fin} { }
  C_Segmento (C_Punto fin) : ini_{0.0, 0.0}, fin_{fin} { }
  // mejor, delegación: C_ Segmento (C_Punto fin):C_Segmento{{0.0, 0.0}, fin} { }
};
```

¿Qué pasa con `longitud_` y `ang_` en los últimos dos constructores?

¿Puedo acceder a las coordenadas x_, y_ del C_Punto en los dos últimos constructores?

No → `ang_{ atan2(fin.get_y()-ini.get_y(), fin.get_x()-ini.get_x()) }`

Código: ejemplo constructor (III)

```
#include <cmath>
class C_Punto {
    double x_, y_; // atributos
public:
    C_Punto (double x, double y) : x_{x}, y_{y} {} // constructor normal
    C_Punto () : C_Punto{1.0, 2.0} {} //constructor por defecto (delega)
    double get_x() const { return x_; }
    double get_y() const { return y_; }
};
class C_Segmento {
    C_Punto ini_, fin_;
    double longitud_, ang_;
public:
    // Resto de constructores
    C_Segmento(C_Punto ini, C_Punto fin) : ini_{ini}, fin_{fin},
        ang_{ atan2(fin.get_y()-ini.get_y(), fin.get_x()-ini.get_x()) },
        longitud_ {hypot (ini.get_x()-fin.get_x(), ini.get_y()-fin.get_y())}
    {}
};
```

Mejor utilizar `ini`, `fin` (los parámetros) que `ini_`, `fin_` (los atributos). Los parámetros tienen valor, pero los atributos no está tan claro que lo tengan en este punto

Constructores, destructor y operador asignación

- Existen 3 constructores especiales (dentro de lo especiales que son): por defecto (sin parámetros), de copia (*copy*) y de movimiento (*move*)

```
C_Punto ()              // default constructor
C_Punto (const C_Punto& rhs) // copy constructor
C_Punto (C_Punto&& rhs)   // move constructor
```

- El destructor (solo hay uno) es el encargado de liberar los recursos cogidos por un objeto (memoria, sockets, ficheros, etc.) cuando termina el ámbito del objeto (el bloque en que está definido)

```
~C_Punto ()
```

- El operador de asignación (*operator*=). De copia o de movimiento:

```
C_Punto& operator= (const C_Punto& rhs)// copy assignment
C_Punto& operator= (C_Punto&& rhs)   // move assignment
```

Funciones especiales y el compilador de C++

- El compilador de C++ invoca el constructor de copia cuando
  - Se crea un objeto a partir de otro que ya existe (uso normal de ctor.)
  - En funciones, cuando se pasa un parámetro por copia o retorna un valor, si no se utilizan punteros ni referencias
- El compilador de C++ invoca el constructor y el operador= de movimiento cuando están involucrados valores temporales
- El compilador de C++ invoca el destructor cuando el objeto sale del ámbito en que está declarado
- El compilador de C++ genera automáticamente las 6 funciones anteriores, pero el programador puede:
  - Solicitar que el compilador genere un error si se invoca el método → al final de la declaración se pone `=delete`
  - Forzar la generación por defecto → al final de la declaración `=default`

Ejemplo constructores y destructores

Universidad Politécnica de Cartagena

```
class C_Punto {
  double x_, y_;
public:
  C_Punto (double x, double y) : x_{x},
     y_{y} { cout << "Ctor normal\n"; }
  C_Punto () : x_{0}, y_{0}
     { cout << "Ctor por defecto\n"; }
  C_Punto (const C_Punto &c) : x_{c.x_},
     y_{c.y_} { cout << "Ctor copia\n"; }
  C_Punto& operator= (const C_Punto& rhs)
     { x_ = rhs.x_; y_ = rhs.y_;
      cout << "Operator=\n"; return *this; }
  ~C_Punto() { cout << "Dtor (" << x_
          << " " << y_ << ")\n"; }
};
C_Punto f (C_Punto a) {
    cout << "Dentro de f, antes de salir\n";
    return a;
}
```

El 99% de las veces no hay que definirlos. El compilador hace el trabajo

```
int main() {
    C_Punto a, b{1,2}, c{b};
    b=a;
    return 0;
}
```

```
Ctor por defecto
Ctor normal
Ctor copia
Operator=
Antes de invocar f()
Ctor copia
Dentro de f, antes de salir
Ctor copia
Operator=
Dtor (1 2)
Dtor (1 2)
Antes de terminar
Dtor (1 2)
Dtor (0 0)
Dtor (1 2)
```

this *es un puntero al objeto sobre el que se ha invocado el método y solo puede utilizarse dentro del código de una clase*

Reglas para constructor/destructor/op=

- El compilador genera un ‘constructor por defecto’ automáticamente si la clase no proporciona ningún constructor. Este constructor invoca los constructores por defecto de cada uno de los atributos de la clase, en el orden en que aparecen declarados los atributos. Se puede forzar su creación: `C_Punto() = default`
- Respecto a las otras funciones especiales (destructor, constructor de copia/movimiento y operador= de copia/movimiento):
  - El compilador las genera automáticamente, con comportamiento por defecto (invoca esas operaciones en cada uno de los atributos de la clase) y correcto, si no se define ninguno de los 5 en la clase
  - Solo si la clase contiene punteros o referencias es necesario redefinirlos. Y en este caso, se recomienda utilizar *smart pointers* (definidos en `<memory>`) para gestionar la memoria correctamente

Siempre que se define uno, hay que definirlos todos

Universidad Politécnica de Cartagena

https://www.youtube.com/watch?v=7Qgd9B1KuMQ

Generación de constructor/destructor/op=

https://blog.feabhas.com/2015/11/becoming-a-rule-of-zero-hero/

Siempre que se define uno, hay que pensarlos todos

The compiler supplies ...	If you write ...							
	None	**Def Ctor**	**Any Ctor**	**Dtor**	**Copy-ctor**	**Copy-op=**	**Move-ctor**	**Move-op=**
Def Ctor	✓	–	☒	✓	☒	✓	☒	✓
Dtor	✓	✓	✓	–	✓	✓	✓	✓
Copy-ctor	✓	✓	✓	✓[1]	–	✓[1]	✗	✗
Copy-op=	✓	✓	✓	✓[1]	✓[1]	–	✗	✗
Move-ctor	✓	✓	✓	☒[1]	☒[1]	☒[1]	–	✗
Move-op=	✓	✓	✓	☒[1]	☒[1]	☒[1]	✗	–

- Las operaciones de movimiento se han definido para mejorar el rendimiento de la aplicación, evitando copias. Pero no pasa nada si faltan

✓[1] El compilador genera *warnings*. Se recomienda declarar el método explícitamente, aunque sea **`=default`**

✗ El compilador lo declara con **`=delete`** → participa en la selección de la función que debe ser invocada (*overload resolution*)

☒ El compilador no declara el método → no participa en *overload resolution*

☒ [1] La operación de movimiento se convierte en copia

Universidad Politécnica de Cartagena

https://www.youtube.com/watch?v=vLinb2fgkHk

blog.feabhas.com/2015/11/becoming-a-rule-of-zero-hero/

Clases que no se pueden copiar ni mover

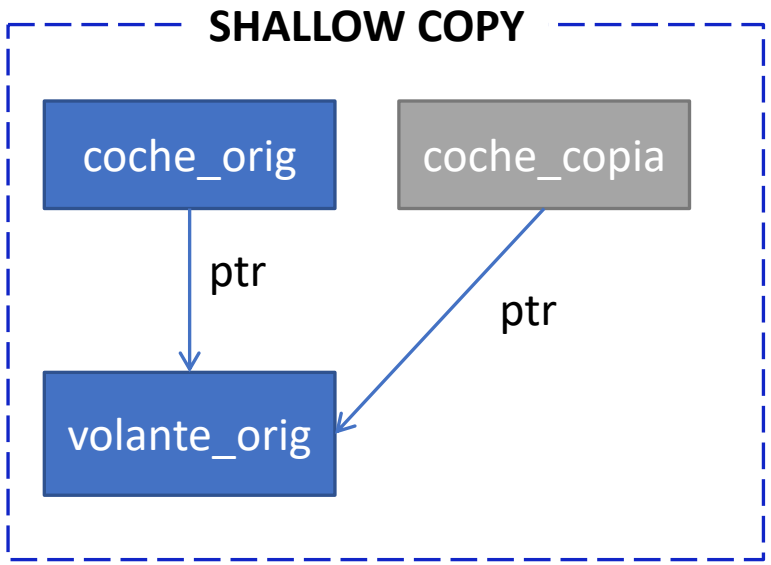
```
class No_Move {
  // atributos y métodos privados
public:
  No_Move (const No_Move& rhs) = default;
  No_Move& operator= (const No_Move& rhs) = default;
  // resto de métodos públicos
};
// los movimientos se convierten en copias
```

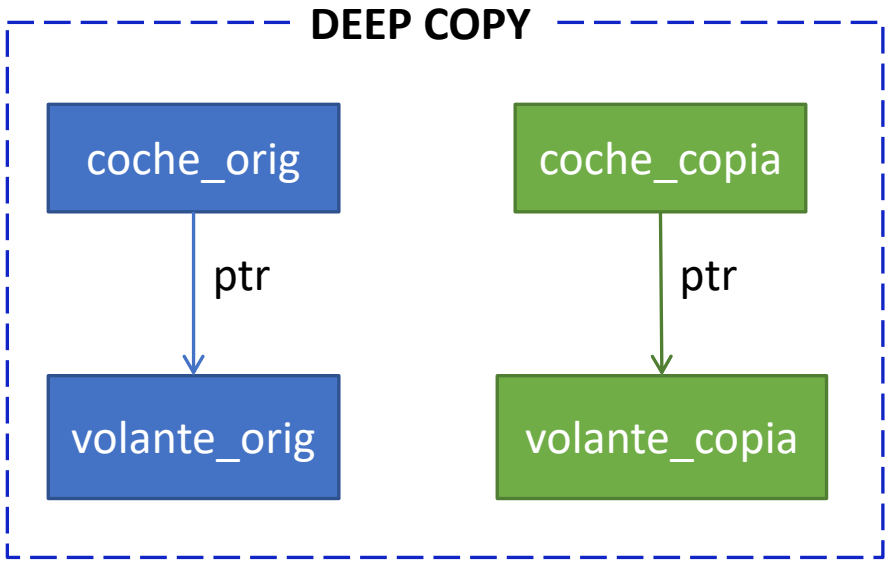
- Las operaciones especiales se pueden, además, declarar en la parte privada o protegida para restringir su visibilidad
- Las operaciones de movimiento no se deben **=delete** → pueden producir errores en otras partes del código. Es mejor no declararlas para que el compilador invoque las operaciones de copia

```
class No_Move_No_Copy {
  // atributos y métodos privados
public:
  No_Move_No_Copy (const No_Move_No_Copy& rhs) = delete;
  No_Move_No_Copy& operator= (const No_Move_No_Copy& rhs) = delete;
  // resto de métodos públicos
};
// los movimientos y copias generan errores de compilación
```

Deep copy vs. shallow copy

- La diferencia aparece **cuando entre los atributos de una clase hay punteros**, ya que en el resto de casos la copia es siempre *deep copy* (invocación del constructor de copia para cada atributo)





- La implementación por defecto cuando hay punteros es *shallow copy*, que generalmente no es la semántica esperada
- En general, no debe utilizar punteros “crudos” sino *smart pointers*

Resources Acquisition is Inicialization (RAII)

- Técnica básica en C++ para evitar pérdidas de recursos (*resource leaks*)
  - Recurso es todo elemento del sistema informático que tiene que ser gestionado correctamente (adquirido y liberado), como memoria, ficheros, etc.
- Técnica empleada por las clases de la librería estándar
- Consiste en que los recursos no se adquieren directamente, sino que están gestionados por una clase (*handler*) → el constructor los adquiere y en el destructor los libera automáticamente
- Utilizando de forma sistemática clases que ya emplean RAII, es posible utilizar C++ sin recolector de basura (*garbage collector*) fácilmente
  - Gestión dinámica de memoria: smart pointers
  - Colecciones de datos: string, array, vector, map, etc.
  - Otros: ficheros (fstream), hilos (thread), mutex, etc.

Adición de nuevas funciones miembro (métodos)

- Es posible añadir cualquier número de métodos a una clase, respetando las reglas generales que aplican a funciones
- Dentro de una función se tiene acceso a todos los atributos del objeto de forma implícita, sin necesidad de pasarlos como parámetros. Y también se puede invocar cualquier función del objeto poniendo solo su nombre
- La palabra reservada **`this`** es una *referencia al objeto* sobre el que se invoca un método. Existe únicamente en el contexto de un método de objeto
- Los métodos se pueden marcar como **`const`** para indicar que no modifican el estado del objeto (i.e., sus atributos) y que, por tanto, pueden utilizarse con objetos (o referencias) *constantes* (y con objetos no constantes también)
  - El modificador **`mutable`** se puede usar para marcar atributos que pueden ser modificados incluso aunque el objeto sea **`const`**. Utilizar con cuidado

Clase C_Punto: getters y setters, operator<<

```
class C_Punto {
public:
  double get_x () const { return x_; }
  double get_y () const { return y_; }
  void set_x (double x) { x_ = x; }
  void set_y (double y) { y_ = y; }
  double dist_al_origen () const {
    return sqrt(x_*x_ + y_*y_);
  }

  double dist_a_punto (const C_Punto& p) const {
    const double xs = x_ - p.x_;
    const double ys = y_ - p.y_;
    return sqrt(xs*xs + ys*ys);
    // return hypot(xs, ys);
  }
  C_Punto () : C_Punto(-10.0,-10.0) {}
  C_Punto (double x, double y) :
     x_{x}, y_{y} {}
```

getters y setters

constructores

```
  friend ostream& operator<< (ostream& os,
   const C_Punto &p) {
    os << "Punto en (" << p.x_ << ", "
     << p.y_ << ')';
    return os;
  }
private:
  double x_;
  double y_;
};

int main () {
  C_Punto p1, p2 {3.0,1.0}; // construccion
  cout << p1 << endl << p2 << endl;
  p2.set_x(22.22); // envío de mensaje
  cout << p1.dist_al_origen() << " p1-p2: "
     << p1.dist_a_punto(p2) << endl;
  return 0;
}
```

atributos

```
Punto en (-10, -10)
Punto en (3, 1)
14.1421 p1-p2: 34.046
```

¿Y si C_Punto no tuviera *setters*?

Código: clase C_Segmento

```
class C_Segmento {
 C_Punto ini_, fin_;     double longitud_, ang_;
 void actualizar () { // puede ser invocada siempre que se cambie un punto
   const double x1 = ini_.get_x(), x2=fin_.get_x();
   const double y1 = ini_.get_y(), y2=fin_.get_y();
   ang_ = atan2(y2-y1, x2-x1);
   longitud_ = hypot(x1-x2, y1-y2);
 }
public:
 C_Segmento(double x1,double y1,double x2,double y2):
       C_Segmento{ {x1,y1}, {x2,y2} } {}
 C_Segmento(double x2, double y2) : C_Segmento {0.0, 0.0, x2, y2} {}
 C_Segmento(C_Punto ini, C_Punto fin):ini_{ini}, fin_{fin} { actualizar(); }
 C_Segmento(C_Punto fin) : C_Segmento { {0.0, 0.0}, fin} {}
 double get_longitud () const { return longitud_; }
 C_Punto get_ini () const { return this->ini_; }
};
```

El operador **->** *es como el* **.** *pero cuando se tiene un puntero en lugar de una variable*

No necesario (se puede retornar `ini_`*). Recuerde que* ***this*** *es un puntero*

Código: mejorando clase C_Segmento

```
class C_Segmento{
  ...
  void actualizar () {
    const double x1 = ini_.get_x(), x2 = fin_.get_x();
    const double y1 = ini_.get_y(), y2 = fin_.get_y();
    ang_ = atan2(y2-y1, x2-x1);
    longitud_ = ini_.dist_punto(fin_);
  }
public:
  ...
  void cambiar_punto_ini (const C_Punto &p) {
   ini_ = p; // operador asignación por copia
   actualizar();
  }
  void desplazar (const double d_x, const double d_y) {
   ini_ = C_Punto { ini_.get_x()+d_x, ini_.get_y()+d_y }; // opcion 1
   fin_.set_x(fin_.get_x()+d_x); // opcion 2
   fin_.set_y(fin_.get_y()+d_y); // opcion 2
  }
};
```

¿Por qué no invoco actualizar()?

Operadores

- C++ permite redefinir los operadores del lenguaje (`+`, `*`, `<<`, `==`, etc.) para los tipos de usuario, de forma que los programas pueden ser más expresivos
- Se puede conseguir el mismo objetivo añadiendo funciones. Pero:
  - Es más cómodo hacer `cout << mi_punto << '\n';` que `imprimir (mi_punto);`
  - Es más cómodo `if(punto1==punto2)` que `if(son_iguales(punto1, punto2))`
- La definición del operador está forzada por la forma en que se utiliza
- Que sea una opción no quiere decir que se tenga que utilizar siempre
- Hay que redefinirlos con criterio y manteniendo la semántica habitual
- Lamentablemente, no todos los operadores se definen de la misma forma, y hay además una serie de consideraciones en el diseño de la clase que pueden influir en la forma elegida

http://en.cppreference.com/w/cpp/language/operators
http://stackoverflow.com/questions/4421706/operator-overloading/4421719#4421719

Código: operator<< y operator>> (I)

Aprender “de memoria”

```
class C_Punto {
  ...
  friend ostream& operator<< (ostream &os, const C_Punto &p) {
    os << '(' << p.x_ << ", " << p.y_ << ')';
    return os;
  }
  friend istream& operator>> (istream &is, C_Punto &p) {
    double x, y;
    is >> x; is >> y; // falta gestion de errores en E/S
    if (/* No se han leído correctamente los valores de entrada */) {
      is.setstate(std::ios::failbit);
    } else {
      C_Punto tmp {x, y};
      swap (tmp, p); // un patrón interesante
    }
    return is;
  }
};
```

`friend` indica que una función (o una clase) tiene acceso a la parte privada, pero no forma parte de la interfaz de la clase en cuestión

Código: operator<< y operator>> (II)

```
class C_Segmento {
  ...
  friend ostream& operator<< (ostream& os, const C_Segmento& r) {
    os << r.ini_ << " ---> "  << r.fin_ << " [";
    os << r.longitud_ << '@'  << r.ang_ << ']';
    return os;
  }
  ...
  friend istream& operator>> (istream& is, C_Segmento& r) {
    C_Punto a, b;
    is >> a;
    if (!is.fail()) {
      is >> b;
      if (!is.fail()) {
       C_Segmento tmp {a, b};
       swap (tmp, r);
       return is;
      }
    }
    is.setstate(std::ios::failbit);
    return is;
  }
};
```

Código: operadores lógicos (I)

```
class C_Punto {
  ...
  friend bool operator== (const C_Punto &lhs, const C_Punto &rhs) {
       if (&lhs == &rhs) return true; // comparo direcciones memoria
       if ((lhs.x_ == rhs.x_) && (lhs.y_ == rhs.y_) ) return true;
       return false;
  }
};
// Las siguientes funciones están definidas fuera del contexto de la clase:
bool operator< (const C_Punto &lhs, const C_Punto &rhs) {
  if (lhs.get_x() < rhs.get_x()) return true; // demasiado simple :-)
  return false;
}
bool operator>= (const C_Punto &lhs, const C_Punto &rhs) {
  return !(lhs<rhs);
     // reutilización ante todo → si hay error, solo hay que corregir 1 sitio
}
```

*Como existen funciones `get_x()` y `get_y()`, puedo hacer estos operadores **friend** o no. Lamentablemente, no pueden ser funciones miembro*

*C++20 define el spaceship operator, **<=>**, para gestionar comparaciones con 1 solo operador*

Código: operadores lógicos (II)

```
int main () {
  C_Punto a, b{3,3}, c;
  cout << (a==a ? "a igual a OK\n" : "a no igual a KO\n");
  cout << (a==b ? "a igual b KO\n" : "a no igual b OK\n");
  cout << (a==c ? "a igual c OK\n" : "a no igual c KO\n");
  cout << (a>=b ? "a >= b KO\n" : "not (a >= b) OK\n");
  return 0;
}
```

```
a igual a OK
a no igual b OK
a igual c OK
not (a>=b) OK
```

- El operador ternario `( exprLógica ? expr1 : expr2 )` retorna siempre una expresión
- Es una versión simplificada y restringida del **`if/else`**
- Retorna `expr1` si `exprLógica` se evalúa a **`true`**, y `expr2` si `exprLógica` se evalúa a **`false`**

Ejemplo uso de **this**

- **this** es un puntero al objeto. Es la única forma en que un objeto se puede referir a sí mismo
- Solo se puede utilizar en una función miembro (método), en un objeto

```
class C; void imprimir (const C &c); // declaración por adelantado
class C { int a_;   public:
  C (int a=3) : a_{a} {}
  void llama_imprimir() const { imprimir (*this); }
  void soy_yo (const C &c) const
    { cout << (&c == this? "SI" : "NO") << " soy yo mism@\n";  }
  friend ostream& operator<< (ostream &os, const C &c)
    { os << "VAL: " << c.a_;  return os;  }
};
void imprimir (const C &c) { cout << "Desde la función: " << c << endl; }
int main() {
  C a, b{7}; a.llama_imprimir();
  b.llama_imprimir(); a.soy_yo(b); a.soy_yo(a);
  return 0;
}
```

```
Desde la función: VAL: 3
Desde la función: VAL: 7
NO soy yo mism@
SI soy yo mism@
```

Objetos constantes

- Con objetos constantes solo se pueden invocar métodos **`const`**, que son aquellos que no cambian el estado del objeto sobre el que se invocan (i.e., no modifican el valor de sus atributos). **`mutable`** permite establecer una excepción a esta regla
- En general, hay que marcar variables/objetos y métodos como **`const`** siempre que sea posible y tenga sentido → nuestro código es más seguro y el compilador puede optimizarlo

```
class C_Punto {
  double x_, y_;
public:
  C_Punto (double x=0.0, double y=0.0) :
       x_{x}, y_{y} {}
  void set_x (const double x) { x_ = x; }
  double get_x () { return x_; }
  double get_x_const () const { b x_; }
  friend ostream& operator<< (ostream &os,
   const C_Punto &c) {
    os << "Punto (" << c.get_x_const()
      << ", " << c.y_ <<')’;
    /* os << "Punto (" << c.get_x() << ", "
      << c.y_ <<')'; // error const! */
    return os;
  }
};
int main() {
  const C_Punto a {1,2};
  cout << a << '\n';
  //a.set_x(3);
  //double la_x = a.get_x();
  double la_x_const = a.get_x_const();
  cout << "C. x -> " << la_x_const << '\n';
  return 0;
}
```

Colecciones y constructores

```
class Test { int a_, b_;
public:
  Test() { cout << "--> Default ctor!\n"; }
  Test(int a):a_{a} {cout <<"--> Ctor param!\n";}
  Test(int a, int b) : a_{a}, b_{b} {
    cout << "--> Ctor paramsss!\n";
  }
  Test(const Test &c){cout<<"--> Ctor copia!\n";}
  Test(Test &&c) { cout << "--> Ctor move!\n"; }
};

int main() {
 const int TAM = 3;
 cout << "======= a1 ======\n";
 array<Test, TAM> a1; // ctor defecto
 cout << "======= a2 ======\n";
 array<Test,TAM> a2 {{Test{5,6},Test{8},Test{}}};
 cout << "====== v1 =======\n";
 vector<Test> v1 (TAM);
 cout << "====== v2 =======\n";
 vector<Test> v2;
 v2.reserve(TAM);
```

Para el caso general, es la forma correcta

push_back: se pasa un objeto ya creado
emplace_back: se pasan los parámetros para construirlo en el sitio

```
 for (int i=0; i<TAM; ++i)
  v2.emplace_back(i, i);
 cout << "======= v3 ======\n";
 vector<Test> v3;
 v3.reserve(TAM);
 for (int i=0; i<TAM; ++i) {
   v3.push_back(Test{i,i});
 }
 cout << "======= v4 ======\n";
 vector<Test> v4;
 v4.reserve(TAM);
 for (int i=0; i<TAM; ++i) {
   v4.push_back(v3.at(i));
 }
 cout << "======= v5 ======\n";
 vector<Test> v5 {Test{5,6},
   Test{8}, Test{}};
}
```

```
======= v5 ======
--> Ctor paramsss!
--> Ctor param!
--> Default ctor!
--> Ctor copia!
--> Ctor copia!
--> Ctor copia!
```

```
======= a1 ======
--> Default ctor!
--> Default ctor!
--> Default ctor!
======= a2 ======
--> Ctor paramsss!
--> Ctor param!
--> Default ctor!
====== v1 =======
--> Default ctor!
--> Default ctor!
--> Default ctor!
====== v2 =======
--> Ctor paramsss!
--> Ctor paramsss!
--> Ctor paramsss!
======= v3 ======
--> Ctor paramsss!
--> Ctor move!
--> Ctor paramsss!
--> Ctor move!
--> Ctor paramsss!
--> Ctor move!
======= v4 ======
--> Ctor copia!
--> Ctor copia!
--> Ctor copia
```

Conceptos fundamentales de OO (III)

- **Herencia**: facilidad por la cual se pueden declarar nuevas clases a partir de otras ya existentes, heredando atributos y métodos. La clase hija puede añadir nuevos atributos y métodos, así como sobrescribir los métodos heredados del padre
- **Polimorfismo**: capacidad de tratar de manera homogénea (por ejemplo, almacenarlas en un vector) un conjunto de instancias de clases que guardan entre sí una relación de herencia, pero sin perder el tipo concreto de cada objeto
- **Composición**: cuando alguna de las propiedades de una clase tiene como tipo otra clase

```
class C_Vehiculo {
 ...
}
```

```
class C_Rueda {
 ...
}
```

Herencia

```
class C_Coche : public C_Vehiculo {
 C_Rueda r1, r2, r3, r4;
 ...
}
```

Composición

Herencia

```
class C_Persona {
};
class C_Empleado : public C_Persona {
};
class C_Estudiante : public C_Persona {
};
```

Herencia. Tipos de herencia:

- **public**: se mantiene el nivel de visibilidad de las características heredadas. Se puede ampliar y reducir la visibilidad
- **private**: todas las características de Persona se heredan como privadas. No se pueden realizar asignaciones polimórficas (la nueva clase no sería una Persona). Útil para herencia de implementación: heredar solo para reutilizar la implementación, no la interfaz

- La herencia es siempre una extensión, no una restricción: la clase hija es una versión de la clase padre con capacidades adicionales
- **Principio de Liskov**: *las superclases deben poder ser sustituibles por cualquier subclase en el código*
- Las clases 'hija' tiene una copia de todos los atributos de las clases padre (es posible heredar de más de una clase padre; ojo con esto: problema *herencia en diamante*) y de los métodos no privados, salvo constructores, destructor, **operator**= y funciones **friend**
- La inicialización se hace por capas: se van creando los objetos desde el ancestro más lejano hasta la clase actual. "Desde mi tatarabuelo hasta mí"

Ejemplo herencia (I)

```
class C_Persona {
  string nombre_;
public:
  C_Persona (string n) : nombre_{n} {}
  string get_nombre () const { return nombre_; }
  friend ostream& operator<< (ostream &os, const C_Persona &p)
    { os << "PERSONA:: " << p.nombre_; return os; }
};
class C_Empleado : public C_Persona {
  string empresa_;
public:
  C_Empleado (string nombre, string empresa) : C_Persona (nombre), empresa_{empresa} {}
  string get_empresa () const { return empresa_; }
};
ostream& operator<< (ostream &os, const C_Empleado &e)
  { os << "EMPLEADO:: " << e.get_nombre() << " | " << e.get_empresa(); return os; }
```

Solo se requiere acceso de lectura a los atributos del objeto

Habitualmente, **operator<<** se declara como **friend** para tener acceso directo a los atributos y poder imprimirlos. La referencia **const** asegura que no se van a modificar

La clase hija puede añadir métodos nuevos

El constructor no se hereda. Hay que invocar el de la clase padre. Y debe ser el primero

Versión alternativa. Puesto que existen funciones que me retornan los valores de los atributos, puedo hacer que **operator<<** sea una función externa

Ejemplo herencia (II)

```
class C_Estudiante : public C_Persona {
  string universidad_;
public:
  C_Estudiante (string nombre, string univ) : C_Persona (nombre), universidad_{univ} {}
  string get_uni () const { return universidad_; }
  string get_nombre () const { return '#' + C_Persona::get_nombre(); }
};
int main () {
  C_Persona p {"Persona"};
  C_Empleado em {"Empleado", "Empresa"};
  C_Estudiante es {"Estudiante", "UPCT"};
  cout << "PERSONA:: " << p.get_nombre() << endl;
  cout << "EMPLEADO:: " << em.get_nombre() << " | " << em.get_empresa() << endl;
  cout << "ESTUDIANTE:: " << es.get_nombre() << " | " << es.get_uni() << endl;
  cout << p << endl;
  cout << em << endl;
  return 0;
}
```

Sobrescribo (redefino) un método heredado.
El atributo nombre_ no es visible en la clase hija (está definido como **private**, no **protected**), así que hay que invocar el método get_nombre() de la clase padre

Gracias a **operator<<**

```
PERSONA:: Persona
EMPLEADO:: Empleado | Empresa
ESTUDIANTE:: #Estudiante | UPCT
PERSONA:: Persona
EMPLEADO:: Empleado | Empresa
```

Uso de **final**

Mediante el uso de **final** se indica que no se puede heredar de una clase o que el método no puede ser redefinido

```
class A final {
  void f(int);
};

class B {
  void f() final;
};
```

```
class C : public A {}; // Error

class D : public B {
    void f() {}; // Error
};
```

Polimorfismo (I)

- El mecanismo de herencia permite reutilizar código y establecer relaciones de jerarquía entre ellas, pero ¿qué pasa si necesito procesar colecciones de objetos del tipo C_Persona y sus subclases?

```
vector<C_Persona> vps; vector<C_Empleado> vems; vector<C_Estudiante> vess;
```

- Pero esto me obliga a tener colecciones **distintas**. El principio de Liskov dice que *"las superclases deben poder ser sustituibles por cualquier subclase en el código"*; entonces, ¿esto funciona?

```
vps.push_back(p); vps.push_back(em); vps.push_back(es);
for (const auto &i : vps) cout << i << endl;
```

```
PERSONA:: Persona
PERSONA:: Empleado
PERSONA:: Estudiante
```

- Sí y no. Puedo almacenar objetos de distintos tipos (siempre que tengan una relación de herencia) ☺, pero todos se comportan como si fueran del tipo C_Persona ☹. ¿Y esto?:

```
vems.push_back(p); vems.push_back(em); vems.push_back(es);
for (const auto &i : vems) cout << i << endl;
```

Compilation failed due to following error(s).

```
main.cpp: In function 'int main()':
main.cpp:66:21: error: no matching function for call to 'std::vector::push_back(C_Persona&)'
     vems.push_back(p); vems.push_back(em); vems.push_back(es);
                        ^
```

Polimorfismo (II)

- Solución: C++ identifica el tipo correcto (e invoca el método correcto) en una colección mixta si:
  1. El acceso a dicho objeto se realiza a través de un puntero/referencia a la clase base
  2. La clase base marca como **`virtual`** los métodos que van a particularizar las clases hijas. Estos son los métodos que realmente justifican la herencia, ya que definen la diferencia de comportamiento. Los métodos no marcados como **`virtual`** no producen *dynamic dispatch*, sino que se ejecutaría el método de `C_Persona`

```
class C_Persona { public:
  virtual string get_type() const { return "PERSONA:: "; }
};
class C_Empleado : public C_Persona { public:
  string get_type() const override { return "EMPLEADO:: "; }
};
class C_Estudiante : public C_Persona { public:
  string get_type() const override { return "ESTUDIANTE:: "; }
};
int main () {
  // se instancian los objetos de C_Persona, C_Empleado y C_Estudiante como antes
  vector<C_Persona*> vps_p;
  vps_p.push_back(&p); vps_p.push_back(&em); vps_p.push_back(&es);
  for (const auto i : vps_p) cout << i->get_type() << endl;
  for (const auto i : vps_p) cout << i->get_nombre() << endl;
}
```

```
PERSONA::
EMPLEADO::
ESTUDIANTE::
```

```
Persona
Empleado
Estudiante
```

La `->` es como el `.` pero cuando se tiene un puntero

Polimorfismo y operator<< (y otras situaciones)

- Contratiempo: las funciones **friend** **NO** forman parte de la interfaz de una clase y no son **virtual**

```
class C_Persona { protected: string nombre_;
  friend ostream& operator<< (ostream &os, const C_Persona &p)
    { os << p.to_string(); return os; }
    virtual string to_string() const
    { return {"PERSONA:: " + nombre_}; }  };
class C_Empleado : public C_Persona { protected: string empresa_;
  string to_string() const override
    { return {"EMPLEADO:: " + nombre_ + " | " + empresa_}; }   };
class C_Estudiante : public C_Persona { protected: string universidad_;
  string to_string() const override
    { return {"ESTUDIANTE:: " + get_nombre() + " | " + universidad_}; }   };
int main () {
  C_Persona p {"Persona"}; C_Empleado em {"Empleado", "Empresa"};
  C_Estudiante es {"Estudiante", "UPCT"};
  vector<C_Persona*> vps_p;
  vps_p.push_back(&p); vps_p.push_back(&em); vps_p.push_back(&es);
    for (const auto i : vps_p) cout << *i << endl;
}
```

```
PERSONA:: Persona
EMPLEADO:: Empleado | Empresa
ESTUDIANTE:: #Estudiante | UPCT
```

Otro ejemplo, código completo (I)

```
class C_Persona {
protected:
  string nombre_;
  virtual string to_string() const { return "PERSONA:: " + nombre_; }
public:
  C_Persona (string n) : nombre_{n} { }
  string get_nombre () const { return nombre_; }
  virtual string get_type() const { return "PERSONA:: "; }
  friend ostream& operator<< (ostream &os, const C_Persona &p)
    { os << p.to_string(); return os; }
};

class C_Empleado : public C_Persona {
protected:
  string empresa_;
  string to_string() const override { return "EMPLEADO:: " + nombre_ + " | " + empresa_; }
public:
  C_Empleado (string nombre, string empresa) : C_Persona (nombre), empresa_{empresa} { }
  string get_empresa () const { return empresa_; }
  string get_type() const override { return "EMPLEADO:: "; }
};
```

```
class C_Estudiante : public C_Persona {
protected:
  string universidad_;
  string to_string() const override
    { return "ESTUDIANTE:: " + get_nombre() + " | " + universidad_; }
public:
  C_Estudiante (string nombre, string univ) : C_Persona (nombre), universidad_{univ} {}
  string get_uni () const { return universidad_; }
  string get_nombre () const { return '#' + C_Persona::get_nombre(); }
  string get_type() const override { return "ESTUDIANTE:: "; }
};
void clase_base (const C_Persona p) {
  cout << "CLASE BASE\n";
  cout << "\tget_nombre() [NP]-> "<< p.get_nombre() << endl; // [NP]: no polimórfico
  cout << "\tget_type() [P] -> " << p.get_type() << endl; // [P]: polimórfico (virtual)
  cout << "\toperator<< [P] -> " << p << endl;
}
void clase_base_ref (const C_Persona &p) {
  cout << "CLASE BASE REF\n";
  cout << "\tget_nombre() [NP]-> "<< p.get_nombre() << endl;
  cout << "\tget_type() [P] -> " << p.get_type() << endl;
  cout << "\toperator<< [P] -> " << p << endl;
}
```

Otro ejemplo, código completo (III)

```
int main () {
  C_Persona p {"Antonio Persona"};
  C_Empleado em {"Juan Empleado", "Empresa"};
  C_Estudiante es {"Lucia Estudiante", "UPCT"};
  clase_base(p);
  clase_base(em);

  clase_base(es);
```

```
CLASE BASE
        get_nombre() [NP]-> Antonio Persona
        get_type() [P] -> PERSONA::
        operator<< [P] -> PERSONA:: Antonio Persona
CLASE BASE
        get_nombre() [NP]-> Juan Empleado
        get_type() [P] -> PERSONA::
        operator<< [P] -> PERSONA:: Juan Empleado
CLASE BASE
        get_nombre() [NP]-> Lucia Estudiante
        get_type() [P] -> PERSONA::
        operator<< [P] -> PERSONA:: Lucia Estudiante
```

```
  clase_base_ref(p);

  clase_base_ref(em);

  clase_base_ref(es);
  return 0;
}
```

```
CLASE BASE REF
        get_nombre() [NP]-> Antonio Persona
        get_type() [P] -> PERSONA::
        operator<< [P] -> PERSONA:: Antonio Persona
CLASE BASE REF
        get_nombre() [NP]-> Juan Empleado
        get_type() [P] -> EMPLEADO::
        operator<< [P] -> EMPLEADO:: Juan Empleado | Empresa
CLASE BASE REF
        get_nombre() [NP]-> Lucia Estudiante
        get_type() [P] -> ESTUDIANTE::
        operator<< [P] -> ESTUDIANTE:: #Lucia Estudiante | UPCT
```

Static binding vs Dynamic binding

- Cuando se invoca una función, el *binding* (unión o enlace) es el proceso por el cual se elige la función concreta (y correcta) que debe ser invocada
- **Static/early binding → enlace estático/temprano**
  - Se resuelve y hace en tiempo de compilación
  - Es el método por defecto en C++, salvo cuando un parámetro es un objeto
  - POO añade el enlace dinámico, junto con la posibilidad de elegir entre ambos
  - Enlace estático se activa si no se tiene una referencia/puntero al objeto, o si se invoca un método no **`virtual`**

```
void clase_base (const C_Persona p);
```

- **Dynamic/late binding → enlace dinámico/tardío**
  - Se resuelve y hace en tiempo de ejecución, por lo que no es tan rápido como el estático
  - Se activa cuando se tiene un puntero/referencia a objeto y se invoca un método **`virtual`**

```
void clase_base_ref (const C_Persona &p);
```

Clases abstractas y métodos puros virtuales

- Una clase debe ser **abstracta** cuando declara el comportamiento común de la jerarquía de herencia y el diseñador quiere evitar que se pueda instanciar
- Una clase es abstracta cuando tiene un **método puro virtual**, que es un método virtual sin implementación heredable por defecto. Por tanto, las clases hijas están forzadas a implementarlo

```
virtual XXX funcion_pura_virtual (YYY) ZZZ = 0;
```

- La herencia (con el *static/dinamic binding*) se une a las reglas de generación de los métodos especiales (destructor, funciones de copia y movimiento) para crear una "tormenta perfecta y caótica"
  - El destructor de la clase base debe ser **`virtual`** para asegurar que los objetos de las clases hija se van a destruir adecuadamente
  - Si se declara el destructor, se deshabilita la generación de los métodos de movimiento
  - Y si se declara cualquiera de ellos, se deshabilita la generación de los métodos de copia

Clase base abstracta con operaciones especiales

```
class Base {
  // métodos y atributos privados
public:
  virtual ~Base() = default;
  Base(Base&&) = default;
  Base& operator=(Base&&) = default;
  Base(const Base&) = default;
  Base& operator=(const Base&) = default;

  // resto de métodos públicos,
  // alguno puro virtual para que la
  // clase sea abstracta
};
```

```
class Hija1 : public Base {
  // implementa, como mínimo,
  //  los métodos puros virtuales
  //  de la clase Base

  // el compilador genera
  //  automáticamente el destructor
  //  y las operaciones de copia y
  //  movimiento (salvo que se
  //  defina alguno en la clase
  //  Hija, en cuyo caso hay que
  //  redefinir los que hagan falta)
};
```

Upcasting y downcasting

- Son conversiones (*casts*) en la jerarquía de herencia: hacia arriba (*upcasting*) o hacia abajo (*downcasting*) en la jerarquía
- El compilador hace la conversión hacia arriba de forma automática y siempre funciona (como convertir un **`double`** en un **`int`**). La información del tipo que sobra, se elimina
  - A veces, puede ser un problema, si no se va con cuidado. Fuente de sorpresas:

```
void clase_base (const C_Persona p) { }
C_Empleado em; clase_base(em); // upcasting. Puede que se quiera hacer
```

  - Pero, aunque use la versión con puntero de la función, solo se resuelven las llamadas polimórficas (funciones **`virtual`**)

```
void clase_base_ref (const C_Persona &p)
    { p->get_uni(); // error, aunque el tipo real sea C_Estudiante }
```

- La conversión hacia abajo permite recuperar el tipo real. Pero solo se puede utilizar cuando se tienen punteros o referencias

Downcasting, **typeid** y **dynamic_cast**<T>

```
int main() {
 C_Persona p {"Antonio Persona"}; C_Empleado em {"Juan Empleado", "Empresa"};
 C_Estudiante es {"Lucia Estudiante", "UPCT"}; vector<C_Persona*> vps_p;
 vps_p.push_back(&p); vps_p.push_back(&em); vps_p.push_back(&es);
 for (const auto i : vps_p) { // es un vector de punteros, por eso es i, no &i
  if(typeid(*i) == typeid(C_Persona)) { // typeid retorna el tipo del argumento
   cout << "Es una persona!!! \n";
  } else if(typeid(*i) == typeid(C_Empleado)) {
   cout << "Es un empleado!!! \n";
  } else if(typeid(*i) == typeid(C_Estudiante)) {
   cout << "Es un estudiante!!! \n";
   //cout << '\t' << i->get_uni() << endl; // Da error! ☹
   C_Estudiante *ptr = dynamic_cast<C_Estudiante*>(i);
   cout << '\t' << ptr->get_uni() << endl; // Ahora no hay error ☺
  }
 }
}
```

```
Es una persona!!!
Es un empleado!!!
Es un estudiante!!!
    UPCT
```

Mismo orden en que han sido introducidos los objetos en el vector

dynamic_cast<T>

- **dynamic_cast**<T> retorna un puntero nulo si no puede hacer la conversion
- Otra forma de conseguir lo mismo que en la transparencia anterior … pero a costa de hacer muchas conversiones dinámicas (que son “lentas”)
- **dynamic_cast**<T> es el PlanB. Si se utiliza mucho, es <u>signo de mal diseño</u>

```
for (const auto i : vps_p) {
  C_Estudiante *ptr_es = dynamic_cast<C_Estudiante*>(i);
  C_Empleado *ptr_em = dynamic_cast<C_Empleado*>(i);
  if(ptr_em){
    cout << "Es un empleado!!! \n";
    cout << '\t' << ptr_em->get_empresa() << endl;
  } else if(ptr_es) {
    cout << "Es un estudiante!!! \n";
    cout << '\t' << ptr_es->get_uni() << endl;
  } else {
    cout << "Es una persona!!! \n";
  }
}
```

```
Es una persona!!!
Es un empleado!!!
    Empresa
Es un estudiante!!!
    UPCT
```

Consejos para la creación de clases (I)

- Una clase puede tener <u>todos</u> los atributos y funciones que estime necesarios
- No hay que crear clases enormes (con mucho estado). Es mejor tener muchas clases pequeñas que pueda combinar, que pocas clases y grandes (además, serán difíciles de utilizar y reutilizar). Como las piezas de Lego
- El estado de una clase (Σ atributos) se almacena en la parte **privada** para tener control sobre su acceso y modificación
- El conjunto de métodos **públicos** se denomina *interfaz*
- Los atributos y métodos **privados** solo pueden ser utilizados internamente
- Hay que prestar atención a los constructores de la clase

Consejos para la creación de clases (II)

- Todos los métodos de una clase tienen acceso al estado y al resto de métodos
- Idealmente, solo las funciones que necesitan acceso directo al estado deberían ser métodos, mientras que las demás deberían ser funciones externas que utilizan la interfaz
  - Potencia la encapsulación
  - Disminuye el conjunto de funciones que deben ser revisadas en caso de modificación del estado
  - Es posible combinar ambas restricciones: métodos que acceden al estado a través de otros métodos. Sobre todo, cuando se tenga que realizar acceso de escritura

Make Interfaces Easy to Use Correctly and Hard to Use Incorrectly

- El diseño de la interfaz es importante. Si lo hace bien, las interfaces serán un placer de usar y aumentarán la productividad de los demás. Si lo hace mal, sus interfaces serán fuente de frustración y errores.
  - **Fácil de usar correctamente** porque ese es el camino de menor resistencia. Casi siempre pasan los parámetros correctos con los valores correctos, porque eso es lo más natural: las cosas simplemente funcionan. Una buena forma de diseñar interfaces que sean fáciles de usar correctamente es desarrollarlas desde el punto de vista del usuario. Ejercítelos antes de que existan. Analice casos de uso comunes y especifique cómo desea que se comporte la interfaz
  - **Difícil de usar incorrectamente** porque anticipan los errores que se pueden cometer y los hacen difíciles, idealmente imposibles de cometer. Una GUI puede deshabilitar o eliminar comandos que no tienen sentido en el contexto actual, por ejemplo, o una API puede eliminar los problemas de ordenamiento de argumentos al permitir que los parámetros se pasen en cualquier orden. También debe observar cómo se usa incorrectamente una interfaz durante el lanzamiento temprano y modificar la interfaz para evitar tales errores. La mejor forma de evitar un uso incorrecto es imposibilitarlo

Sobre todo, recuerde que las interfaces existen para la conveniencia de sus usuarios, no de sus implementadores

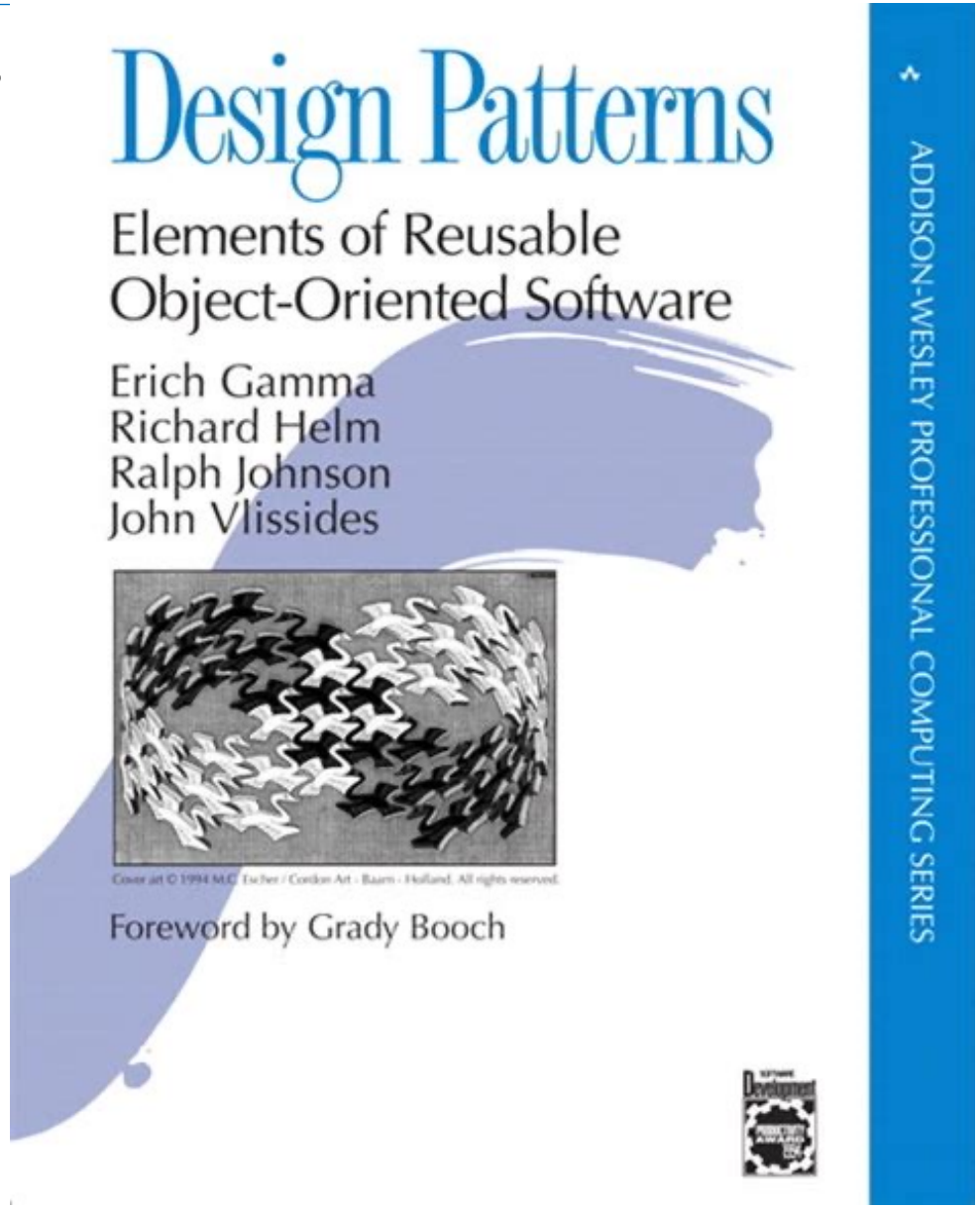
© Scott Meyers

Principios del diseño orientado a objetos

Principio	Enunciado
Abierto/Cerrado (Open-Closed)	Un módulo debe estar abierto para su extensión y cerrado para su modificación, debido a las dependencias que puede crear
De Reemplazabilidad (Liskov)	Las superclases deben poder ser sustituibles por las subclases.
De inversión de dependencia	Depender de abstracciones (interfaces), no de concreciones (clases).
De segregación de interfaces	Mejor muchas interfaces pequeñas específicas para diversos clientes que una interfaz grande de propósito general
De diseño de interfaces	Programe una interfaz, no una implementación.
No superclases concretas	Evitar mantener superclases concretas. Nunca una clase abstracta puede ser hija de una concreta
Favorece composición frente a herencia	Los cambios en la clase padre afectan a **todas** las clases hijas Es difícil saber qué hace una clase hija (hay que revisar la cadena de herencia)
Responsabilidad única	Una clase debe ser responsable hacer una sola cosa Simplifica su diseño y potencia su reutilización

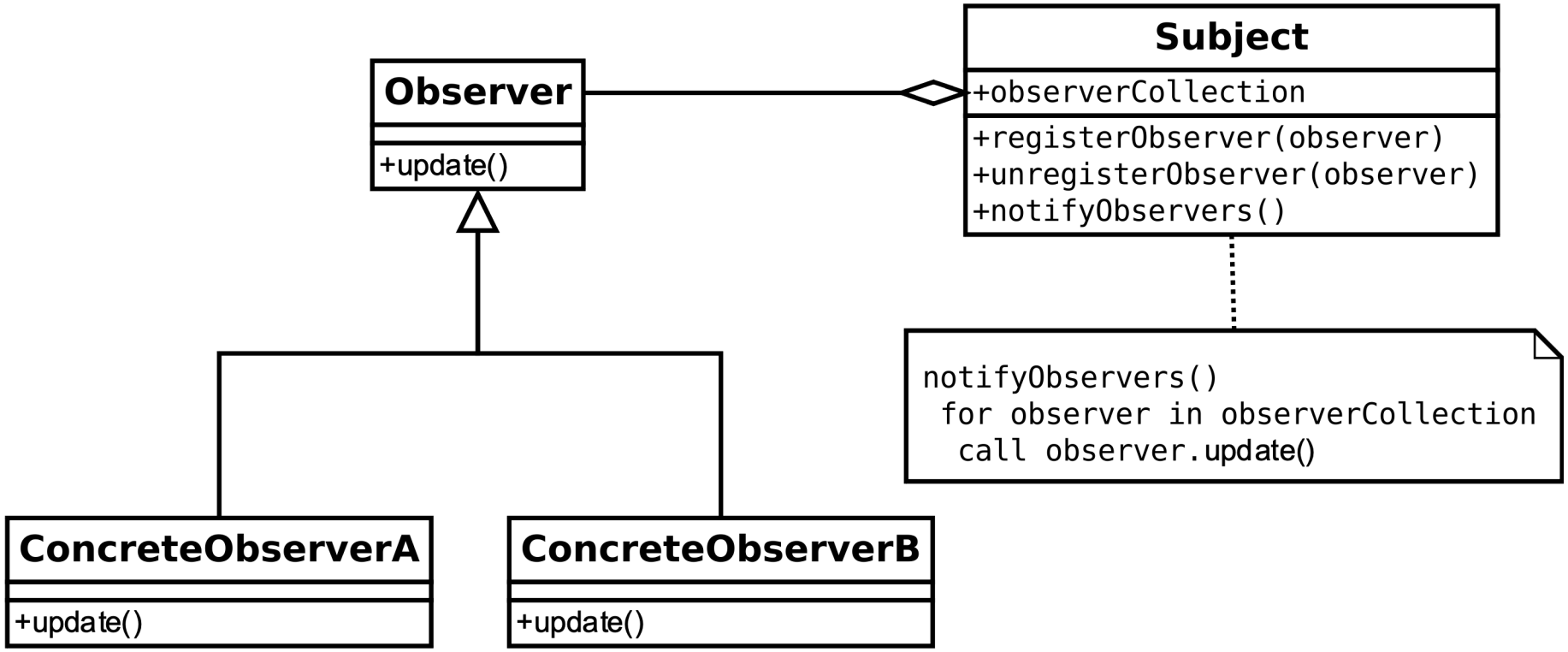
Solo es la punta del iceberg

- Solo hemos visto los principios y mecanismos básicos de la POO para crear objetos, herencia y polimorfismo
- Pero queda saber cómo utilizarlos correctamente y cómo hacer un diseño de clases
- En cuanto el problema crece, es necesario utilizar soluciones conocidas → el libro de patrones de diseño ofrece soluciones OO **probadas**



Ejemplo: patrón observador (I)

- Permite que una clase notifique a otras que su estado ha cambiado, en lugar de que sean estas las que están preguntando constantemente (*polling*)



Ejemplo: patrón observador (II)

```
struct I_Observable { virtual void notificar (string msg) = 0; }; // pure virtual class
class C_Observador1 : public I_Observable { const string m_ {"C_Notificado1 :: "};
  public: void notificar (string msg) override { cout << m_ << msg << endl; }
};
class C_Observador2 : public I_Observable { const string m_ {"C_Notificado2 :: "};
  public: void notificar (string msg) override { cout << m_ << msg << endl; }
};
class C_Sujeto { vector<I_Observable*> clientes_;
 public: C_Sujeto (vector<I_Observable*> cs) : clientes_{cs} {}
      void anyadir (I_Observable *obs) { clientes_.push_back(obs); }
      void notificar() { for (const auto c : clientes_) c->notificar ("Notificacion");  }
};
int main() {
  C_Observador1 n1a, n1b, n1c; C_Observador2 n2a, n2b, n2c;
  C_Sujeto sj { vector<I_Observable*> {&n1a, &n1b, &n2a, &n2b, &n2c} };
  // alternativa: C_Sujeto sj { {&n1a, &n1b, &n2a, &n2b, &n2c} };
  sj.anyadir(&n1c); sj.notificar(); return 0;
}
```

```
C_Notificado1 :: Notificacion
C_Notificado1 :: Notificacion
C_Notificado2 :: Notificacion
C_Notificado2 :: Notificacion
C_Notificado2 :: Notificacion
C_Notificado1 :: Notificacion
```

Errores y fallos de programación

- Inevitablemente, los programas se encontrarán con problemas en tiempo de ejecución: fichero inexistente, conexión de red inestable, valores incorrectos, operaciones no realizables porque el sistema no se encuentra en el estado adecuado, fallos a la hora de mantener el invariante de la clase, etc.
- Estos **errores** son diferentes de fallos de programación (*bugs*), fallos en la lógica: un algoritmo incorrecto o mal implementado, un código que realmente no soluciona el problema, valores sin cubrir (**`if/else`**), etc.
- Además de los *códigos de error* (ahora mejorados gracias a `<utility> y <optional>`), C++ proporciona ***excepciones***: un mecanismo que trabaja a nivel de función (unidad mínima de trabajo de un programa) y que el programador no puede ignorar (los códigos de error se ignoran a menudo)
- Pero las excepciones son un mecanismo que añade sobrecarga en tiempo de ejecución, por lo que se reservan para situaciones realmente excepcionales. En sistemas de tiempo-real, a veces su uso está prohibido

https://docs.microsoft.com/en-us/cpp/cpp/exception-handling-in-visual-cpp?view=msvc-160

http://www.drdobbs.com/when-and-how-to-use-exceptions/184401836

Programación de excepciones (I)

Rol 1: programador de la función

Detecta la condición de error (requiere estudio previo de pre/post condiciones e invariantes):

```
double dividir (double n, double d)
 noexcept(<expresión_lógica>) {
  if (d == 0)
   throw invalid_argument
    ("division por cero");
  else return n/d;
}
```

para plantillas

`noexcept` es optativo e indica si pueden escapar excepciones de la función (lanzadas por la función o por otras funciones invocadas por esta)

Mediante un comentario, o en la documentación de la función, se suele especificar qué excepciones puede lanzar la función, para que el usuario de la función pueda actuar en consecuencia

Rol 2: usuario de la función

Gestiona la excepción en la función invocante (o en alguna de las funciones de la pila de llamadas):

```
void funcion () {
 double a, b, c;
 bool error = true;
 do { cin >> a;       cin >> b;
  try {
   c = dividir (a, b);
   error = false; // solo sin excepción
  } catch (const invalid_argument &error) {
          // siempre 'const &'
   // gestionar el error
   cerr << error.what() << endl;
  }
 } while (error);
 cout << "Division: " << c << endl;
}
```

Programación de excepciones (II)

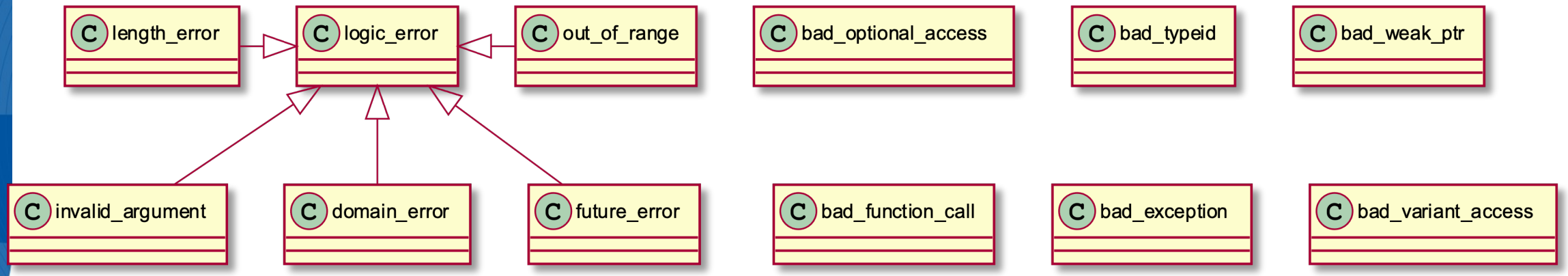
- Es posible lanzar más de una excepción en una función y es posible capturar cualquier excepción (como un '*else*')

```
try {
    c = f(); // puede lanzar excepción
} catch (const invalid_argument &e) {
 // captura excepción invalid_argument
} catch (const overflow_error &e) {
 // captura excepción overflow_error
} catch (...) {
 // captura resto de de excepciones. Siempre es el último 'catch'
}
```

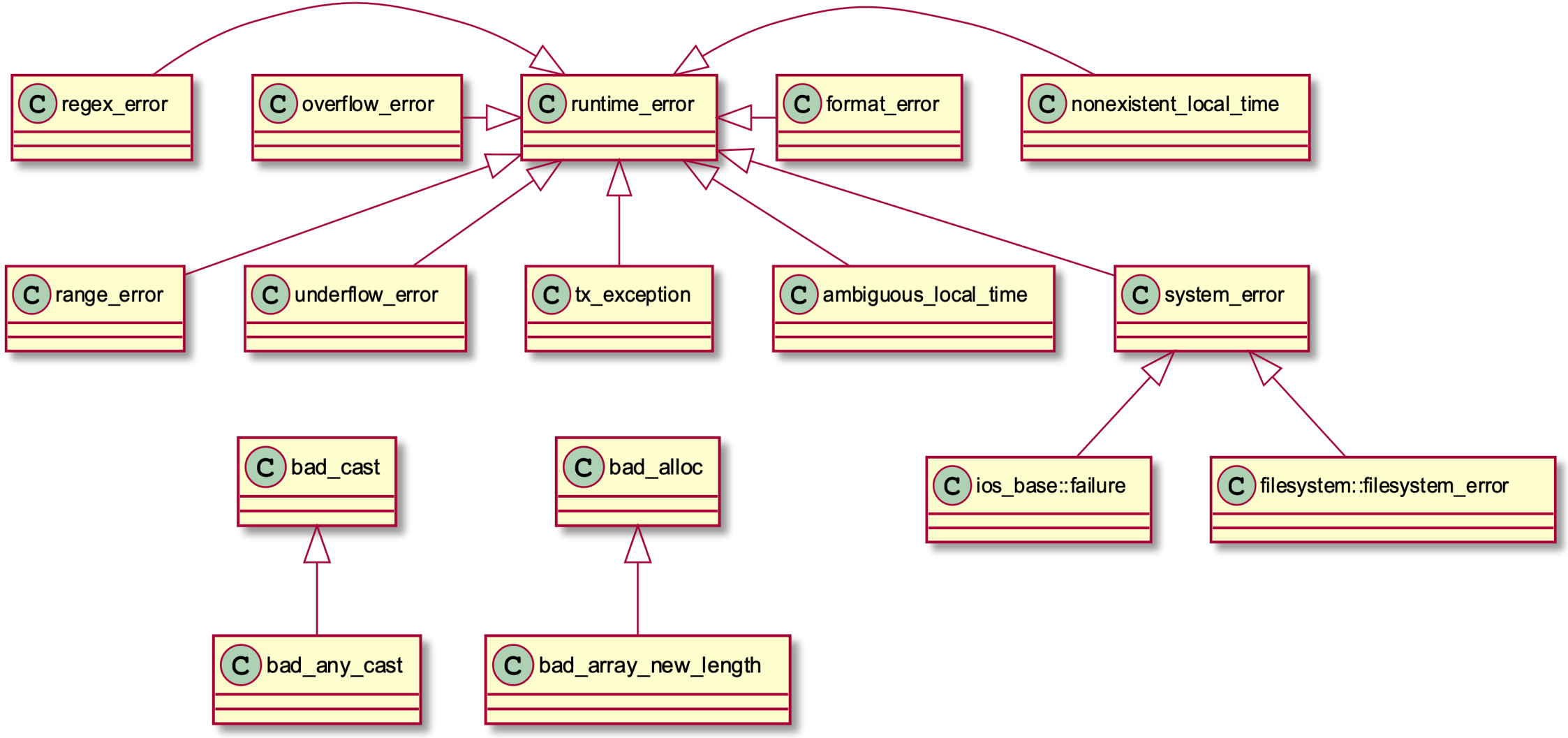
- Los bloques **try/catch** se pueden colocar en secuencia y se pueden anidar
- Comportamiento de **noexcept**:
  - En funciones **`noexcept(false)`** o en las que no aparezca: las excepciones se propagan hacia atrás en la cadena de invocación de funciones. Si llegan sin ser tratadas hasta el *main*, abortan el programa
  - En funciones **`noexcept`** o **`noexcept(true)`** : si una excepción "sale" de la función, el programa se aborta. El compilador emite *warnings* si detecta esta situación

Jerarquía de excepciones (I)

- C++ define un conjunto de excepciones básicas en `<stdexcept>` y funciones de utilidad en la librería `<exception>`
- Todas las excepciones heredan de la clase base `exception`, tienen un constructor que acepta un *string*, y una función *what()* para recuperar el mensaje de error
- Es posible definir nuevos tipos de excepciones utilizando herencia



Jerarquía de excepciones (II)



Lanzando excepción personalizada

```
#include <iostream>
#include <stdexcept>
using namespace std;

class Mi_exc : public bad_alloc {
  int val_;
  const char* msg_;
public :
  Mi_exc(const char* msg, int v=1)
     : msg_{msg}, val_{v} {}
  int get () const noexcept { return val_; }
  const char* what() const noexcept{
    return msg_;
  }
};
```

Puede ser cualquier clase de la jerarquía

Siempre debe tener un método **what()**

```
void lanza () {
  cout << __FUNCTION__
     << "() --> antes de excepcion\n";
  throw Mi_exc("Se ha lanzado mi excepcion", 8);
}
int main(){
  try {
   lanza();
  } catch (const Mi_exc &e) {
   cout << __FUNCTION__
      << "() --> excepcion |"
      << e.what() << "@"<< e.get() << "|\n";
  }
  cout << "Main terminando\n";
  return 0;
}
```

```
lanza() --> antes de excepcion
main() --> excepcion |Se ha lanzado mi excepcion@8|
Main terminando
```

Gestión de excepciones

- Cuando un constructor no puede construir el objeto, lo normal es lanzar una excepción. En este caso, el destructor no se ejecuta, así que hay que tener cuidado

  https://isocpp.org/wiki/faq/exceptions#ctors-can-throw

- Cuando se lanza una excepción, se puede tratar en varios puntos:
  - En el lugar donde se tiene suficiente conocimiento para poder arreglar la situación o tomar acciones correctivas/paliativas. Tratamiento de excepciones por niveles. Cuanto más cerca de la excepción, mejor
  - En límites claramente definidos: función *main* o la función principal de un hilo o la función principal de un sub-sistema
- Dentro de un bloque `catch` se puede utilizar `throw` para volver a lanzar la excepción recibida, para que la gestione la función invocante
- En general, la gestión de errores (al intentar construir un objeto, una función que no puede realizar su cometido, valores de entrada incorrectos, fichero que no existe, etc.) es compleja

Abrahams guarantees for exceptions

0. **Sin garantía:** ☹
1. **Garantía básica**: si se lanza una excepción, no se pierden recursos y los objetos permanecen en un estado destructible y utilizable, pero no necesariamente predecible. Este nivel es apropiado cuando el código del cliente puede hacer frente a operaciones fallidas que ya han cambiado el estado de los objetos
2. **Garantía sólida**: si se lanza una excepción, el estado del programa permanece sin cambios. Este nivel siempre implica una semántica global de compromiso o retroceso (commit-or-rollback), que incluye que no se invaliden las referencias o iteradores en un contenedor si falla una operación. Además, ciertas funciones deben proporcionar una garantía aún más estricta para hacer posibles los niveles de seguridad de excepción anteriores.
3. **Garantía Nothrow**: la función no emitirá una excepción en ningún caso. Resulta que a veces es imposible implementar la garantía fuerte o incluso la básica a menos que se garantice que ciertas funciones no arrojarán (por ejemplo, destructores, funciones de des-asignación)

Se supone que hay que poner un **try/catch** siempre que se invoque una función que no sea **noexcept**

Universidad Politécnica de Cartagena

http://www.gotw.ca/gotw/059.htm