

Tema 4: Herencia y Polimorfismo

Tecnología de la Programación de Videojuegos 1
Grado en Desarrollo de Videojuegos

Miguel Gómez-Zamalloa Gil

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

Introducción

- ✦ La herencia y el polimorfismo son las características propias más importantes de la POO
- ✦ La herencia permite reutilizar código, definiendo nuevas clases a partir de otras ya existentes, formando jerarquías
- ✦ Si la clase B hereda de la clase A:

`class B : public A {...};`

 - ▶ La clase A se denomina **clase base** o **padre** de B, mientras que B se denomina **clase derivada**, **subclase** o **hija** de A.
 - ▶ La clase B hereda todos los métodos y atributos de la clase A
 - ▶ Puede definir nuevos atributos y/o métodos, o también redefinir métodos heredados
- ✦ Se puede heredar de una clase (**herencia simple**) o de varias clases (**herencia múltiple**)

Sintaxis de la Herencia en C++

✦ Sintaxis de herencia simple:

```
class B : TipoHerencia A {...};
```

- ▶ Indica que la clase B hereda de A
- ▶ **TipoHerencia** especifica el tipo de herencia, que en C++ puede ser: **public**, **protected** y **private**

✦ Por ahora nos limitamos a la herencia simple y pública (la más habitual en orientación a objetos):

```
class B : public A {...};
```

Cuidado: Por defecto la herencia es privada en C++

Niveles de Protección

- ✦ Además de los niveles de protección **private** y **public**, existe el nivel **protected** para métodos y atributos
 - ▶ El nivel **protected** indica que éstos pueden ser utilizados dentro de la implementación de una clase que herede de ella
 - ▶ En cualquier otra situación se comportan como privados
- ✦ En la implementación de una subclase podemos utilizar todos los elementos públicos y protegidos (**protected**) de la clase base, pero no los privados.

Niveles de Protección

♦ Para herencia pública:

Declaración de método o atributo	Accesible desde la propia clase	¿Accesible desde subclases?	¿Accesible para las demás?
<code>public</code>	SI	SI	SI
<code>protected</code>	SI	SI	NO
<code>private</code>	SI	NO	NO

Ejemplo

```
class Base {  
private: // no accesible desde las subclases  
    int x, y;  
protected: // accesible desde las subclases  
    int nx, ny;  
    void init() { x = 0; y = 0; nx = 0; ny = 0; }  
public: // accesible para todos  
    Base() : x(0), y(0), nx(0), ny(0) {}  
    Base(int a, int b) : x(a), y(b), nx(a), ny(b) {}  
  
    void set(int a, int b, int na, int nb) {  
        x = a; y = b;  
        nx = na; ny = nb;  
    }  
  
    int getX() const { return x; }  
    int getY() const { return y; }  
    int getNX() const { return nx; }  
    int getNY() const { return ny; }  
};
```

Ejemplo

```
class Hija: public Base { // Hija tiene los atributos y métodos de Base y
private:
    Vector2D p1, p2; // nuevos atributos
    int z;
public:
    void actualizaN() { // nuevo método
        init();          // Correcto, init es protected
        p1.setX(nx);     // Correcto, nx es protected
        z = y + ny;      // INCORRECTO -> z = getY() + ny;
        set(2, 3, z, z); // Correcto
    }

    Vector2D getP1() const { return p1; }
    const Vector2D& getP2() const { return p2; }
};
```

Constructoras

- ♦ La constructora de una subclase invoca, implícita o explícitamente, en primer lugar, a una constructora de la clase base.
 - ▶ **Explícitamente:** mediante llamada a una constructora de la clase base en la secuencia de inicialización
 - ▶ **Implícitamente:** Si no se especifica una, se invoca automáticamente a la constructora por defecto de la clase base. Si no existe -> **Error**

Orden en la construcción de un objeto:

1. Una constructora de la clase base
2. Las constructoras de los nuevos atributos (de tipo clase) de la clase (y/o la secuencia de inicialización)
3. El bloque de código de la constructora de la clase

Recuerda que al crearse un objeto con new, se ejecuta la constructora del objeto que se crea

Constructoras

- ♦ Supongamos que declaramos la variable:

```
Hija h;
```

- ▶ ¿Qué constructora se ejecuta?
- ♦ Como la clase **Hija** no tiene definida ninguna constructora, se ejecuta su constructora por defecto:
 1. La constructora por defecto de la clase **Base**
 - ▶ Si no existe -> Error de compilación
 2. Las constructoras por defecto de los nuevos atributos de **Hija**, en el orden textual de la declaración: p1 y p2 (z es básico, no se inicia por defecto).
 - ▶ Si alguna no existe -> Error de compilación

Constructoras

- ♦ Si añadimos a la clase Hija una constructora sin argumentos:

```
Hija() { set(1,1,1,1); z = 7; }
```

- ♦ Al declarar **Hija** h; se ejecutarán:
 1. La constructora por defecto de la clase **Base**
 - ▶ Si no existe -> **Error de compilación**
 2. Las constructoras por defecto de los nuevos atributos de **Hija**, en el orden textual de la declaración: p1 y p2 (z es básico, no se inicia por defecto)
 - ▶ Si alguna no existe -> **Error de compilación**
 3. El bloque de instrucciones de la constructora de **Hija**
- ♦ El compilador completa automáticamente la sec. de inicialización:

```
Hija() : Base(), p1(), p2() { set(1,1,1,1); z = 7; }
```

Constructoras

- ◆ Podemos hacer una llamada explícita a la constructora con dos argumentos de la clase **Base**:

```
Hija(): Base(2,3), p1(2,2), p2(), z(7) {};  
Hija(int a, int b): Base(a,b), p1(a+2,b+1), p2() { z=7; };
```

- ▶ Posibles declaraciones: `Hija h2(2,3); Hija h;`

- ◆ O no hacerla, en cuyo caso el compilador completa la lista de inicializaciones con **Base()**:

```
Hija(int a): Base(), p1(a+2,1), p2(), z(0) {};
```

- ▶ Si no existe -> **Error de compilación**

Destructoras

- ♦ La destructora de una subclase invoca automáticamente, en último lugar, a la destructora de la clase base
 - ▶ Siempre de forma implícita (solo existe una destructora por clase)

Orden en la destrucción de un objeto:


1. El bloque de instrucciones de la destructora de la clase hija. (Las variables creadas con new, hay que liberarlas con delete, y entonces se ejecuta la destructora del objeto liberado)
2. Las destructoras de los nuevos atributos de la clase hija, en el orden inverso al de la construcción
3. La destructora de la clase base

Observa que el orden es precisamente el orden inverso al orden de construcción

Redefinición de Métodos

- ♦ Una subclase puede redefinir métodos de la clase base
 - ▶ El nuevo método puede invocar al método heredado

```
class Base {  
    int x, y;  
    int nx, ny;  
    ...  
    int suma(){  
        return x + y + nx + ny;  
    }  
};  
  
class Hija: public Base {  
    int z;  
    ...  
    int suma(){ // Redefinición para incluir al nuevo atributo z  
        // return getX() + getY() + nx + ny + z; // Mejorable ...  
        return Base::suma() + z; // Así mucho mejor  
    }  
};
```


Invocación al método heredado

Enlace Estático vs. Dinámico

¿Qué método se ejecuta, el heredado o el redefinido?

- ♦ Por defecto se hace **enlace estático** (se decide en tiempo de compilación)
- ♦ Se ejecuta el método de la clase declarada para la variable

Ejemplo:

```
Hija h(2,3);  
int n = h.suma(); // Se ejecuta Hija::suma()  
h.actualizaN();  // Se ejecuta Hija::actualizaN()
```

```
Base b(2,3);  
n = b.suma();    // Se ejecuta Base::suma()  
b.actualizaN();  // Error de compilación. Base no tiene el método!
```

Castings (static_cast)

Casting ascendente (seguro)

- ♦ Al igual que con tipos básicos, el casting ascendente es seguro y automático
- ♦ Todo objeto de la clase hija lo es también de la clase base
- ♦ Ejemplos:

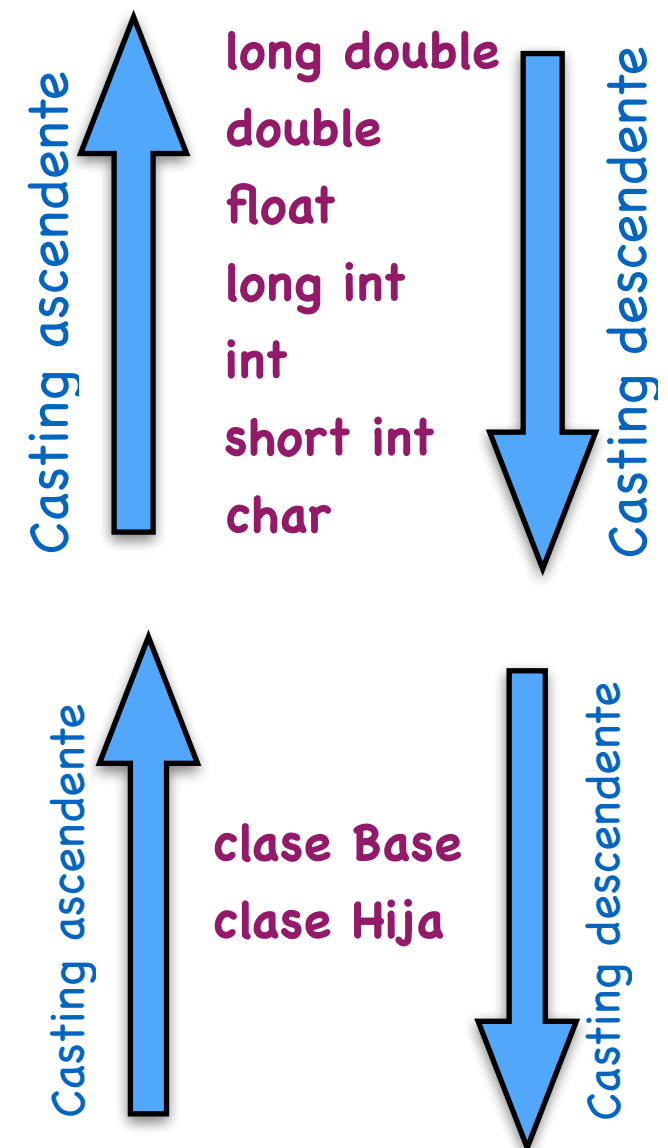
```
Base b(2,3);  
Hija h(2,3);  
int n = static_cast<Base>(h).suma();
```

```
Base b(2,3);  
Hija h(2,3);  
b = static_cast<Base>(h);  
// b = h; // Sería igual  
int n = b.suma(); // Base::suma()
```

Casting descendente (no seguro)

- ♦ Si no es con punteros es muy raro
- ♦ Daría error de compilación. Requiere programar proceso de conversión (constructora por copia correspondiente)

```
Base b(2,3);  
Hija h(2,3);  
int n = static_cast<Hija>(b).suma(); // Error de compilación
```



Polimorfismo

Una variable **Base*** var; puede apuntar a un objeto de cualquiera de las subclases de **Base**

- ♦ El polimorfismo se utiliza con punteros o referencias
- ♦ El operador de asignación para punteros simplemente copia la dirección en la variable
- ♦ Ejemplos:

```
Base b(2,3);  
Hija h(2,3);  
Base* pb = &h; // Hija es subclase de Base  
Hija* ph = &b; // ERROR de compilación  
Hija* ph = pb; // ERROR de compilación  
ph = static_cast<Hija*>(pb); // Ok, pero en general mucho cuidado  
ph = static_cast<Hija*>(&b); // Posibles errores imprevisibles de ejecución!
```


Polimorfismo

- ♦ El `static_cast` simplemente comprueba en compilación que los tipos son compatibles

Su uso correcto es responsabilidad del programador

- ♦ El enlace es estático por defecto

```
Hija h(2,3);  
Base* pb = &h; // Hija es subclase de Base  
int n = pb->actualizaN(); // ERROR, Base no tiene el método actualizaN  
n = static_cast<Hija*>(pb)->actualizaN(); // OK  
n = pb->suma(); // Enlace estático, se ejecuta Base::suma
```

- ♦ Para que se ejecute el método `suma` de `Hija` (tipo del objeto apuntado por `pb`) hay que utilizar `enlace dinámico` (mediante métodos virtuales)

Enlace Dinámico (métodos virtual)

- ✦ El enlace de los métodos **virtual** es **dinámico**: se ejecuta el método de la clase del objeto apuntado en el momento de la ejecución

```
class Base {  
    ...  
    virtual int suma(){  
        return x + y + nx + ny;  
    }  
};  
  
class Hija: public Base {  
    ...  
    virtual int suma() { // Redefinición para incluir al nuevo atributo z  
        return Base::suma() + z;  
    }  
};
```

```
Hija h(2,3);  
Base* pb = &h;  
int n = pb->suma(); // Se ejecuta el suma del objeto real en ejecución
```

Enlace Dinámico (métodos virtual)

- ✦ Los métodos virtual se heredan virtual
- ✦ Por convenio se pone virtual además de en la clase en todas las redefiniciones del método
 - ▶ Así el código es más legible
- ✦ En lenguajes como Java o C# el enlace es dinámico por defecto (de hecho no existe el enlace estático)

Polimorfismo y Destructoras

- ♦ Recuerda que al final de la destructora de una clase se invoca automáticamente a la destructora de su clase base
- ♦ **Importante:** Como con todos los métodos, por defecto la destructora se resuelve por enlace estático
- ♦ Esto puede provocar que quede basura

```
Hija* h = new Hija(2,3);
```

```
Base* pb = h;
```

```
int n = pb->suma(); // Se ejecuta el suma del objeto real en ejecución
```

```
delete pb; // Se ejecuta la destructora de la clase Base!
```

Polimorfismo y Destructoras

```
class Base {  
private:  
    T* pt; // Objeto dinámico de clase T  
public:  
    Base(){pt = new T(...); ... }  
    ...  
    ~Base(){ delete pt; pt = nullptr; }  
};  
  
class Hija : public Base {  
private:  
    M* pm; // Objeto dinámico de clase M  
public:  
    Hija() : Base(){ pm = new M(...); ... }  
    ...  
    ~Hija(){ delete pm; pm = nullptr; } // Se llama a la destructora de Base  
};
```

Polimorfismo y Destructoras

```
Hija* h = new Hija(2,3);
```

```
Base* pb = h;
```

```
int n = pb->suma(); // Se ejecuta el suma del objeto real en ejecución
```

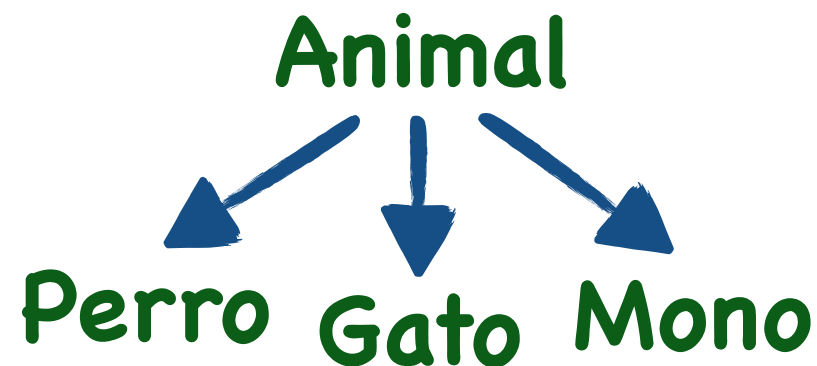
```
delete pb; // Se ejecuta la destructora de la clase Base!
```

- ✦ Quedaría sin destruir la memoria apuntada por **pm**
- ✦ Cada objeto debe responsabilizarse de borrar la memoria que él ha creado

La destructora siempre debe ser virtual, para que se resuelva siempre por enlace dinámico

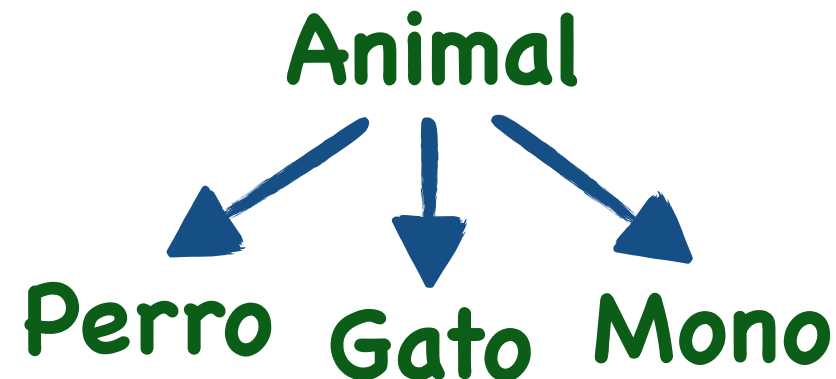
Jerarquías de Clases Polimórficas

- ♦ La herencia y el polimorfismo permiten diseñar e implementar sistemas extensibles con gran facilidad
- ♦ Los programas procesan los objetos de todas las clases existentes en una jerarquía de forma **genérica**
- ♦ Para que una clase sea **polimórfica** basta con que tenga un método virtual. La destructora siempre debe ser virtual.



Jerarquías de Clases Polimórficas

- ♦ Un puntero de un tipo puede apuntar a cualquier objeto de sus subclases



- ♦ Se establece una relación de tipo "es un"

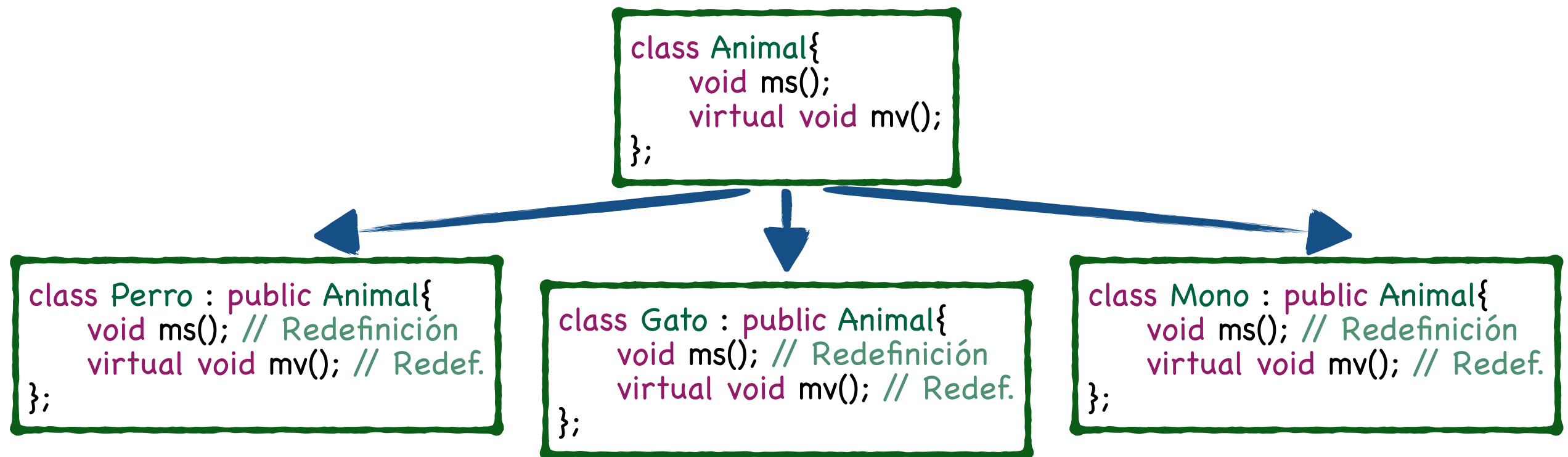
- ▶ ¿Un perro es un animal? ➡ SI

```
Animal* a = new Perro(); // Correcto
```

- ▶ ¿Un animal es un perro? ➡ NO

```
Perro* p = new Animal(); // Incorrecto! Error de compilación
```


Jerarquías de Clases Polimórficas



`Animal* as[N];` // array de elementos polimórficos (Animal es una clase polimórfica)

`Perro* p = new Perro();`

`Gato* g = new Gato();`

`as[0] = p;` // perro es un animal

`as[1] = g;` // gato es un animal

`as[2] = new Mono();` // mono es un animal

...
`for (Animal* a : as){`

`a->ms();` // ms no es virtual -> enlace estático -> Siempre A::ms

`a->mv();` // mv es virtual -> enlace dinámico -> Ejecuta mv del objeto apuntado

`}`

Clases Abstractas

- ✦ Una clase **abstracta** es una clase que no se puede instanciar, es decir, no se pueden tener objetos de esa clase, ni estáticos ni dinámicos
- ✦ Se usan como clases base de las cuales heredan otras clases para definir jerarquías
- ✦ Una clase es abstracta si no tiene constructora pública, o si tiene un método abstracto (método virtual puro)

```
virtual tipoResultado nombreMetodo(parámetros) = 0;
```

- ✦ Una clase abstracta deba tener la destructora virtual
- ✦ Una clase abstracta pura (también llamada interfaz) consta exclusivamente de métodos abstractos

Clases Abstractas

```
class Animal{  
    virtual void m(){...};  
    virtual void mover() = 0;  
};
```

```
class Perro : public Animal{  
    void m(){...}; // Redefinición  
    virtual void mover(){...};  
};
```

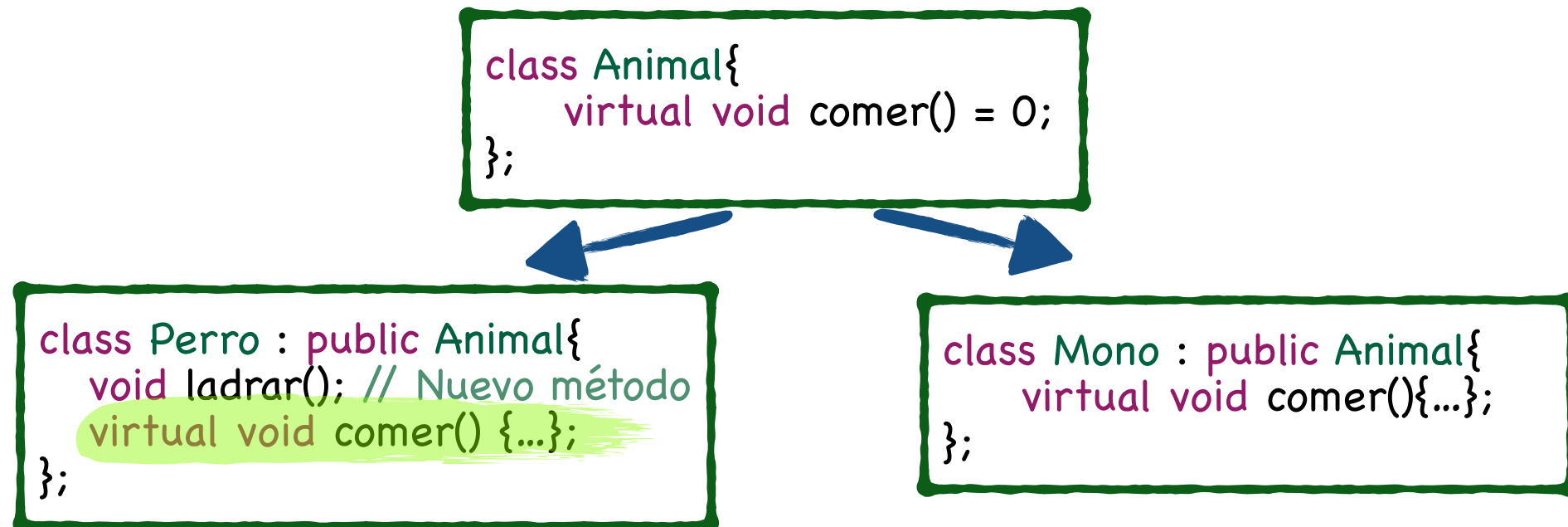
```
class Gato: public Animal{  
    void m(){...}; // Redefinición  
    virtual void mover(){...};  
};
```

```
class Mono: public Animal{  
    void m(){...}; // Redefinición  
    // No implementa mover()  
};
```

```
Animal a; // Error: No se puede construir un objeto de Animal (es abstracta)  
Animal* pa; // Correcto  
pa = nullptr; // Correcto  
pa = new Animal(); // Error: No se puede construir un objeto de Animal (es abstracta)  
pa = new Perro(); // Correcto: Perro es concreta  
pa = new Mono(); // Error: No se puede construir un objeto de Mono (tb es abstracta)
```

Una clase que hereda de una clase abstracta, si no implementa todos los métodos abstractos, sigue siendo abstracta

Polimorfismo y Métodos Nuevos



```
Animal* as[N]; // array de elementos polimórficos (Animal es una clase polimórfica)  
Gato* g = new Gato();  
as[0] = new Perro(); // perro es un animal  
as[1] = g; // gato es un animal  
  
as[0]->ladrar(); // Error de compilación: Animal no sabe ladrar  
static_cast<Perro*>(as[0])->ladrar(); // Se ejecuta Perro::ladrar pero cuidado!
```

No se chequea que `as[0]` apunte a un objeto de tipo `Perro`. Si no habrá problemas impredecibles. Es responsabilidad del programador!

Casting Dinámico (dynamic_cast)

- ✦ Solo se debe usar **static_cast** cuando hay seguridad 100% de que el objeto es de una clase compatible

Casting dinámico (dynamic_cast)

- ✦ Con **dynamic_cast** se comprueba en ejecución que el tipo es compatible, si no devuelve **nullptr**
- ✦ Eso permite hacer castings más seguros:

```
Animal* as[N]; // Array de elementos polimórficos (Animal es una clase polimórfica)
as[0] = new Perro(); // perro es un animal

...
Perro* p = dynamic_cast<Perro*>(as[0]); // Se intenta dependiendo del tipo real
if (p != nullptr) p->ladrar(); // Solo se invoca si el tipo es correcto
else throw "Error en casting";
```

- ✦ También se puede preguntar en ejec. por un tipo con **typeid**

```
if (typeid(*as[0]) == typeid(Perro)) ...
```

Herencia y Excepciones

Sintaxis:

```
try {  
    // Código que puede generar excepciones  
    // o que se debe proteger de éstas, evitando su ejecución  
  
} // Después del bloque try, la secuencia de cláusulas catch  
catch(Tipo1[& var]){ // código de tratamiento para Tipo1  
} catch(Tipo2[& var]){ // código de tratamiento para Tipo2  
...  
} catch(TipoN[& var]){ // código de tratamiento para TipoN  
} catch (...) { // código de tratamiento else } // Opcional  
// "fin-try-catch" siguiente instrucción al try-catch
```

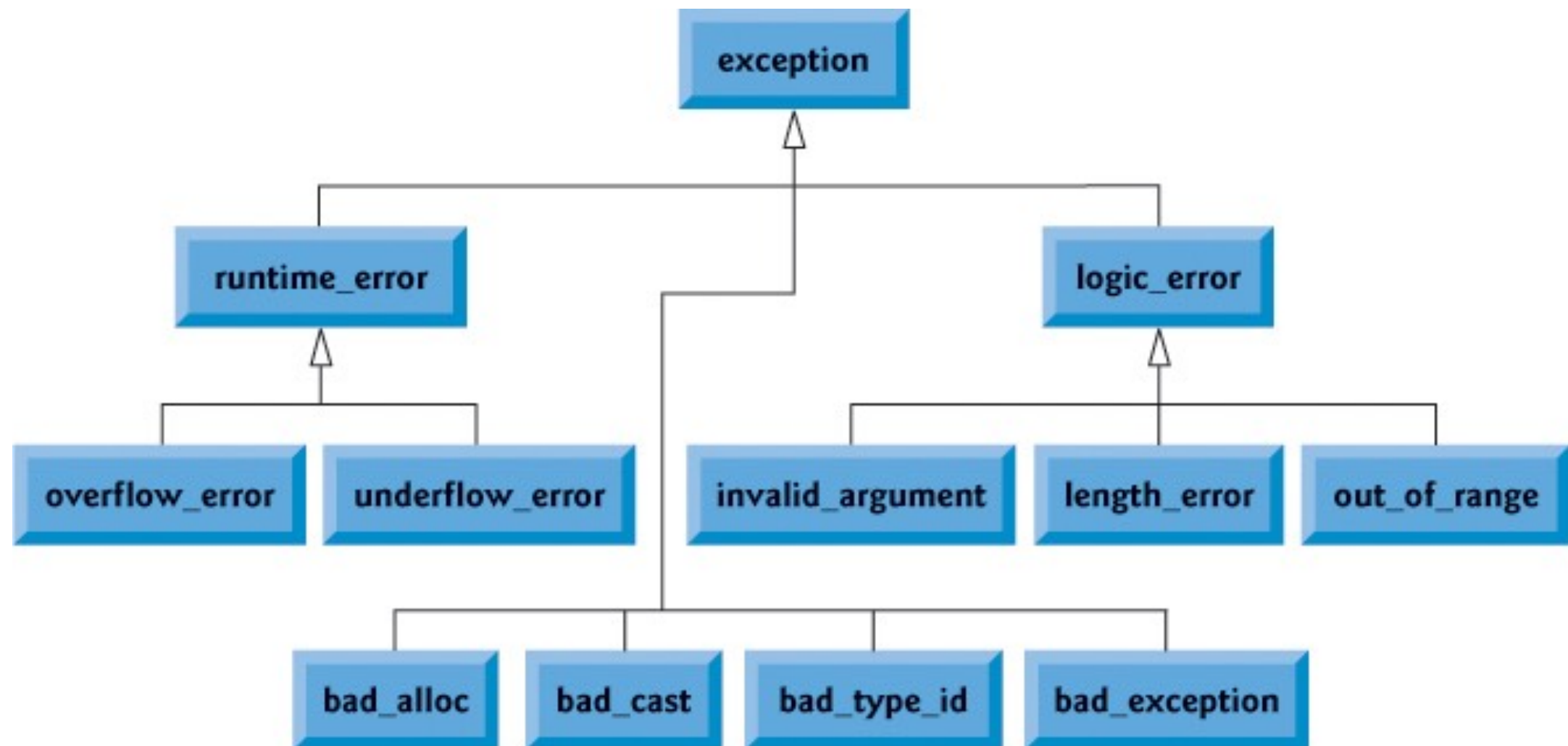
- ♦ Las cláusulas **catch** se prueban secuencialmente hasta que una captura la excepción lanzada, y el control de la ejecución del programa pasa al bloque de código de dicha cláusula.

Herencia y Excepciones

- ✦ Normalmente las excepciones se declaran formando jerarquías
- ✦ En C++ estándar, la clase base de la jerarquía es **exception**, clase definida en la librería **<exception>**, con un método virtual **const char* what()** y constructora sin argumento
- ✦ Algunas subclases de exception: **bad_alloc**, **bad_typeid**, **overflow_error**, **out_of_range**, **invalid_argument**, ...
- ✦ Algunas tienen una constructora con un argumento de tipo **string** o **char***, para el mensaje que describe el error y que se puede consultar con el método **what()**
- ✦ Podemos definir nuestras propias clases de excepciones a partir de alguna de las clases existentes o de forma independiente

Jerarquía de la Librería Estándar

♦ Jerarquía básica de la librería estándar:



♦ En los estándares C++11 y C++17 hay varias más

- ▶ <https://en.cppreference.com/w/cpp/error/exception>

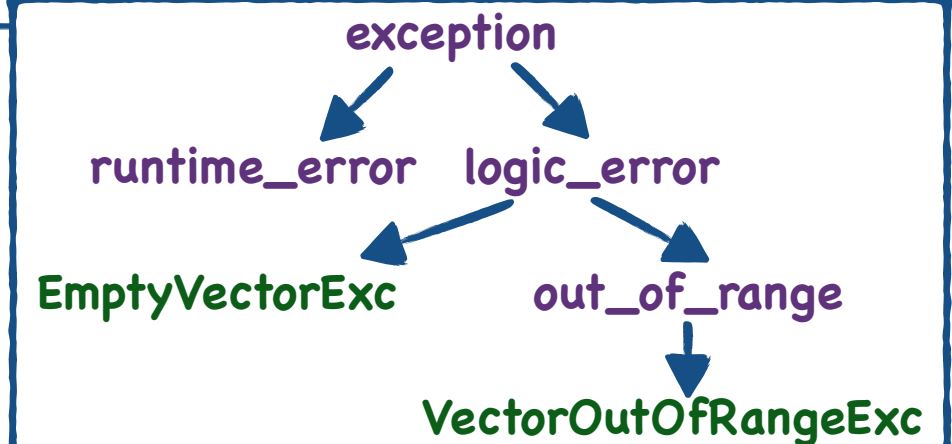
Herencia y Excepciones

- ◆ Podemos simplemente heredar para tener nuevos tipos

```
class EmptyVectorExc : public logic_error {  
public:  
    EmptyVectorExc(const string& m) : logic_error("Empty vector. " + m) {};  
};
```

- ◆ O añadir nuevo comportamiento para hacer cosas más sofisticadas

```
class VectorOutOfRangeExc : public out_of_range {  
protected:  
    int index;  
    int rangeIni;  
    int rangeEnd;  
public:  
    VectorOutOfRangeExc(const string& m, int i, int rIni, int rEnd) :  
        out_of_range(m + "Invalid access at pos. " + to_string(i) + ". Valid range is [" +  
            to_string(rIni) + "," + to_string(rEnd) + "]"),  
        index(i), rangeIni(rIni), rangeEnd(rEnd) {};  
};
```



Herencia y Excepciones

- ♦ Las secuencias de **catch** se suelen ordenar de los tipos más específicos a los más generales

```
try {  
    // Código que puede generar excepciones  
    // o que se debe proteger de éstas  
  
} catch (EmptyVectorExc& e) { // código de tratamiento para EmptyVectorExc  
    ...  
}  
catch (VectorOutOfRangeExc& e) { // tratamiento para VectorOutOfRangeExc  
    ...  
}  
catch (logic_error& e) { // código de tratamiento para logic_error  
    ...  
}  
catch (exception& e) { // código de tratamiento para logic_error  
    ...  
}  
catch (...) { // código de tratamiento para otros errores  
    ...  
}  
// "fin-try-catch" siguiente instrucción al try-catch
```

