8. Polimorfismo

El polimorfismo consiste en utilizar objetos declarados como clase base pero con instancias de clases hijas con el objetivo de que se ejecuten los procedimientos adecuados. Por ejemplo: “si les pedimos a todos los animales que produzcan un sonido: los perros van a ladrar y los gatos maullar”.

Ejemplo: no es polimorfismo

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

class Animal{
  public:
    virtual void sonido(){ cout<<"----"<<endl; }
};
class Gato : public Animal{
  public:
    void sonido(){ cout<<"Miauuuu"<<endl; }
};
class Perro : public Animal{
  public:
    void sonido(){ cout<<"Guau guau"<<endl; }
};

int main(){
   Animal animal;
   Gato gato;
   Perro perro;
   animal.sonido();
   gato.sonido();
   perro.sonido();
   cout<<endl;
   Animal& mascota1 = gato;
   Animal* mascota2 = &perro;
   Mascota1.sonido();
   Mascota2->sonido();
}
```

```
----
Miauuuu
Guau guau

Miauuuu
Guau guau
```

8.1 Funciones vituales

Son funciones de la clase Base que se **sobrescriben** por las clases Hijas con el objetivo de utilizar polimorfismo.

Nota: El polimorfismo sólo funciona en referencias y apuntadores.

Ejemplo:

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

class Base{
public:
    void metodo(){
        cout<<"Base::metodo()"<<endl;
    }
    virtual void metodoVirtual(){
        cout<<"Base::metodoVirtual()"<<endl;
    }
};

class Hija : public Base{
public:
    void metodo(){
        cout<<"Hija::metodo()"<<endl;
    }
    void metodoVirtual(){
        cout<<"Hija::metodoVirtual()"<<endl;
    }
};

int main(){
   Base base;
   Hija hija;
   base.metodo();
   base.metodoVirtual();
   hija.metodo();
   hija.metodoVirtual();
   cout<<endl;

   Base  b1 = hija;
   Base& b2 = hija;
   Base* b3 = &hija;
   b1.metodo();
   b2.metodo();
   b3->metodo();
   b1.metodoVirtual();
   b2.metodoVirtual();
   b3->metodoVirtual();
}
```

```
Base::metodo()
Base::metodoVirtual()
Hija::metodo()
Hija::metodoVirtual()

Base::metodo()
Base::metodo()
Base::metodo()
Base::metodoVirtual()
Hija::metodoVirtual()
Hija::metodoVirtual()
```

8.2 Clases abstractas: métodos virtuales puros

Las clases abstractas NO pueden tener instancias. En C++ una clase abstracta es aquella que:

- Tiene al menos un método virtual puro, es decir, un método virtual sin implementación.
- No se pueden crear instancias de esa clase

El objetivo de este tipo de clases es marcar una pauta hacia clases hijas, es decir, se obliga a las clases hijas a implementar el o los método(s) virtual(es) puro(s). En C++ un método vitual puro es aquel que no tiene código y se iguala a 0.

Reglas:

- Sólo de pueden declarar apuntadores o referencias de las clases abstractas
- En la jerárquia de la herancia, al menos una clase debe implementar el método virtual puro.
  Si la clase abstracta A define el método virtual puro, luego B hereda de A e implementa el método, después C hereda de B. C no tiene la oblicación de implementar el método porque su papá B ya lo hizo.
- Una clase abstracta debe tener al menos un método virtual puro, pero puede tener también otros métodos con implementación.
- Si una clase hija no implementa un método virtual puro de su clase madre, entonces también se vuelve clase abstracta.

Ejemplo:

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

class Figura{
    string nombre;
public:
    Figura(string nombre){ this->nombre=nombre; }
    virtual float getArea() = 0;
    virtual float getPerimetro() = 0;
    void imprimir(){
        cout<<nombre<<" area:"<<getArea()<<" perimetro:"<<getPerimetro()<<endl;
    }
};
class Cuadrado : public Figura{
    float lado;
public:
    Cuadrado(string nombre, float lado):Figura(nombre){ this->lado=lado; }
    float getArea(){ return lado*lado; }
    float getPerimetro(){ return 4*lado; }
};
class Circulo: public Figura{
    static const float PI = 3.1416;
    float radio;
public:
    Circulo(string nombre, float radio):Figura(nombre){ this->radio=radio; }
    float getArea(){ return PI*radio*radio; }
    float getPerimetro(){ return PI*2*radio; }
};
int main(){
    Figura* fig1 = new Cuadrado("cuadradito",5);
    Circulo fig2("circulo 1",4);
    fig1->imprimir();
    fig2.imprimir();
    delete fig1;
}
```

8.3 Clases abstractas: constructores y destructores

Constructores de clases abstractas:

- No pueden mandar llamar a funciones virtuales porque aún no existe su implementación

Destructores de clases abstractas:

- Es mejor dejarlos virtuales ya que de estas clases no se crean implementaciones, es mejor que cada subclase implemente el destructor.
- Si no se generan los destructores virtuales, el destructor llamado por delete es el del tipo del objeto

Ejemplo: Destructor en clase abstracta

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

```
class Figura{
public:
    Figura(string nombre){
        this->nombre=nombre;
        // this->imprimir();  ERROR
        cout<<"Constructor Figura"<<endl;
    }
    ~Figura(){
        cout<<"Destructor Figura"<<endl;
    }
protected:
    string nombre;
    virtual float getArea() = 0;
    virtual float getPerimetro() = 0;
    void imprimir(){
            cout<<nombre<<"
area:"<<getArea();
            cout<<"
perimetro:"<<getPerimetro()<<endl;
    }
};
```

```
class Cuadrado : public Figura{
    float lado;
public:
    Cuadrado(string nombre,float lado)
            :Figura(nombre){
        this->lado=lado;
        cout<<"Constructor Cuadrado"<<endl;
    }
    ~Cuadrado(){
        cout<<"Destructor cuadrado";
    }
    float getArea(){ return lado*lado; }
    float getPerimetro(){ return 4*lado; }
};
```

```
int main(){
    Figura* fig1;
    fig1 = new Cuadrado("cuadradito",5);
    delete fig1;
};
```

```
Constructor Figura
Constructor Cuadrado
Destructor Figura
```

Ejemplo: Destructor en clase abstracta con constructor virtual puro

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

```
class Figura{
public:
    Figura(string nombre){
        this->nombre=nombre;
        cout<<"Constructor Figura"<<endl;
    }
    virtual ~Figura() = 0;
protected:
    string nombre;
    virtual float getArea() = 0;
    virtual float getPerimetro() = 0;
    void imprimir(){
            cout<<nombre<<"
area:"<<getArea();
            cout<<"
perimetro:"<<getPerimetro()<<endl;
    }
};
Figura::~Figura(){}
```

```
class Cuadrado : public Figura{
    float lado;
public:
    Cuadrado(string nombre,float lado)
            :Figura(nombre){
        this->lado=lado;
        cout<<"Constructor Cuadrado"<<endl;
    }
    ~Cuadrado(){
        cout<<"Destructor cuadrado";
    }
    float getArea(){ return lado*lado; }
    float getPerimetro(){ return 4*lado; }
};
```

```
int main(){
    Figura* fig1;
    fig1 = new Cuadrado("cuadradito",5);
    delete fig1;
};
```

```
Constructor Figura
Constructor Cuadrado
Destructor Cuadrado
```

8.4 Sobrecarga (overload) contra Sobreescritura (virtual)

	Sobrecarga	Sobreescritura (virtual)
Nombre	Debe ser el mismo	Debe ser el mismo
¿Cómo?	Fuera de clases En una clase Herencia	Sólo por herencia
Parámetros	Deben cambiar, ya sea en: • Tipo • Número	Deben ser los mismos parámetros o subtipos de ellos
Retorno	Puede cambiar	Debe ser el mismo, o en caso de ser una referencia o apuntador puede ser referencia o apuntador de una clase derivada
Nivel de acceso (public, protected, private)	Puede cambiar	En la clase hija debe ser igual o menos restrictivo

```
#include <iostream>   #include <sstream>  #include <string>  using namespace std;
class Figura{
public:
    Figura(string nombre){
        this->nombre=nombre;
    }
    // Sobrecarga -------------------------
    void imprimir(){
        cout<<nombre<<" area:"<<getArea();
        cout<<" perimetro:"<<getPerimetro();
        cout<<endl;
    }
    void imprimir(int n){
        cout<<" area:"<<getArea();
        cout<<" perimetro:"<<getPerimetro();
        cout<<endl;
    }
protected:
    string nombre;
    // Sobreescritura ---------------------
    virtual float getArea() = 0;
    virtual float getPerimetro() = 0;
    virtual Figura* copiar() = 0;
};
```

```
class Cuadrado : public Figura{
    float lado;
public:
    Cuadrado(string nombre,float lado)
         :Figura(nombre){ this->lado=lado; }
    // Sobreescritura ---------------------
    float getArea(){ return lado*lado; }
    float getPerimetro(){ return 4*lado; }
    Cuadrado* copiar(){
        Cuadrado *c;
        c = new Cuadrado(this->nombre,
                         this->lado);
        return c;
    }
    // Sobrecarga -------------------------
    using Figura::imprimir;
    string imprimir(string titulo){
        stringstream s;
        s<<titulo<<nombre;
        s<<" area:"<<getArea();
        s<<" perimetro:";
        s<<getPerimetro()<<endl;
        return s.str();
    }
};
```

```
int main(){
    Figura* fig1 = new
    Cuadrado("cuadradito",5);
    fig1->imprimir();

    Cuadrado c("cuad", 10);
    cout<<"area:"<<c.getArea()<<endl;

    Cuadrado *c2 = c.copiar();
    (*c2).imprimir();
    (*c2).imprimir(1);
    cout<<(*c2).imprimir("c2 ")<<endl;
    delete fig1;
    delete c2;
}
```

8.5 Casting: Upcasting, downcasting

Upcasting: Consiste en la creación de un objeto del tipo de la clase padre con una instancia de una clase hija. Sólo se puede realizar cuando la herencia es pública.

Suponiendo la siguiente estructura:

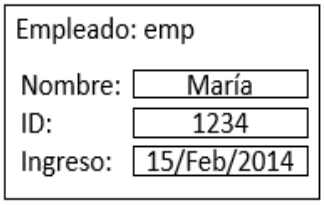
```
class Fecha{
  public:
    int dia,mes,anio;
    Fecha(int d,int m,int a){
       dia=d;mes=m; anio=a; }

class Persona{
  private:
    string nombre;
  public:
    void setNombre(string n){ nombre=n; }
    string getNombre(){ return nombre; }
    Persona(string nombre){
        this->nombre = nombre;
    }
};
```

```
class Empleado : public Persona {
  private:
    int ID;   Fecha ingreso;
  public:
    void setID(int id){ ID=id; }
    int getID(){ return ID; }
    void setFecha(int d,int m, int a){
        ingreso.dia=d; ingreso.mes=m;
        ingreso.anio=a; }
    Fecha getFecha(){ return ingreso; }
    Empleado(string nombre,int ID,
             int d,int m,int a)
             : Persona(nombre),ingreso(d,m,a){
        this->ID = ID;
    }
};
```

Upcasting sería:

Creamos un objeto de tipo empleado:

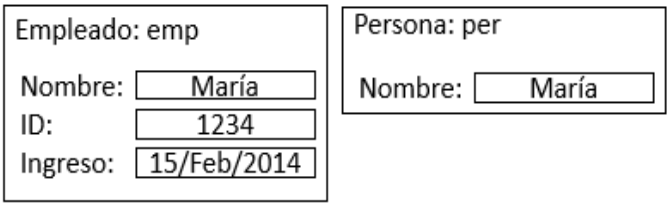
Empleado emp(“María”, 1234, 15, 02, 2014);



Formas de Upcasting:

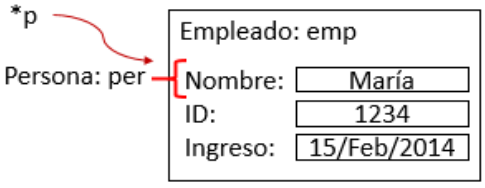
1. Si creamos un objeto tipo persona y le asignamos la instancia emp:
   - Se hace una copia del objeto emp pero sólo con la información correspondiente a la clase persona
   - No se puede utilizar polimorfismo

Empleado emp(“María”, 1234, 15, 02, 2014);
Persona per = emp;



2. Si creamos una referencia o un apuntador tipo persona hacia la instancia emp:
   - Podemos acceder a la información de la instancia emp correspondiente a la clase persona
   - Se puede utilizar polimorfismo.

Empleado emp(“María”, 1234, 15, 02, 2014);
Persona& per = emp;
Persona* p = &emp;



Entones el upcasting se podría hacer de dos formas:

1. Tipo de dato: clase padre, instancia: clase hija
   - Se copia la información
   - Sólo se puede acceder a los atributos y métodos de la clase padre
   - No se puede utilizar el polimorfismo
2. Tipo de dato: referencia o apuntador a la clase padre, instancia: clase hija
   - Sólo se puede acceder a los atributos y métodos de la clase padre
   - Se puede utilizar el polimorfismo

Downcasting: Consiste en la creación de un objeto del tipo de la clase hija con un objeto de la clase padre. Este proceso es más peligroso porque podemos tener una clase que no corresponda.

El downcasting en C++ se realiza con la instrucción

dynamic_cast<tipo de la clase hija>(objeto);

- El cast se verifica en tiempo de ejecución, en caso de que la instancia no corresponda al tipo que se requiere regresa un 0.
- Sólo se puede utilizar si la clase es polimórfica, es decir, si existe al menos un método virtual en la clase padre.
- Sólo se puede convertir a apuntadores o referencias, sin embargo, se recomienda utilizar apuntadores para verificar si el resultado es diferente de 0.

Suponiendo la siguiente estructura:

```
class Figura{
protected:
 string nombre;
public:
 Figura(string nombre){ this->nombre=nombre; }
    virtual float getArea() = 0;
    virtual float getPerimetro() = 0;
    virtual void imprimir(){
       cout<<nombre<<" area:"<<getArea()<<" perimetro:"<<getPerimetro()<<endl; }
};
class Cuadrado : public Figura{
    float lado;
    public:
    Cuadrado(string nombre, float lado):Figura(nombre){ this->lado=lado; }
    float getArea(){ return lado*lado; }
    float getPerimetro(){ return 4*lado; }
    void imprimir(){
    cout<<"CUADRADO "<<nombre<<" area:"<<getArea()<<" perimetro:"<<getPerimetro()<<endl; }
};
class Circulo: public Figura{
    static const float PI = 3.1416;
    float radio;
    public:
    Circulo(string nombre, float radio):Figura(nombre){ this->radio=radio; }
    float getArea(){ return PI*radio*radio; }
    float getPerimetro(){ return PI*2*radio; }
    void imprimir(){
    cout<<"CIRCULO "<<nombre<<" area:"<<getArea()<<" perimetro:"<<getPerimetro()<<endl; }
};
```

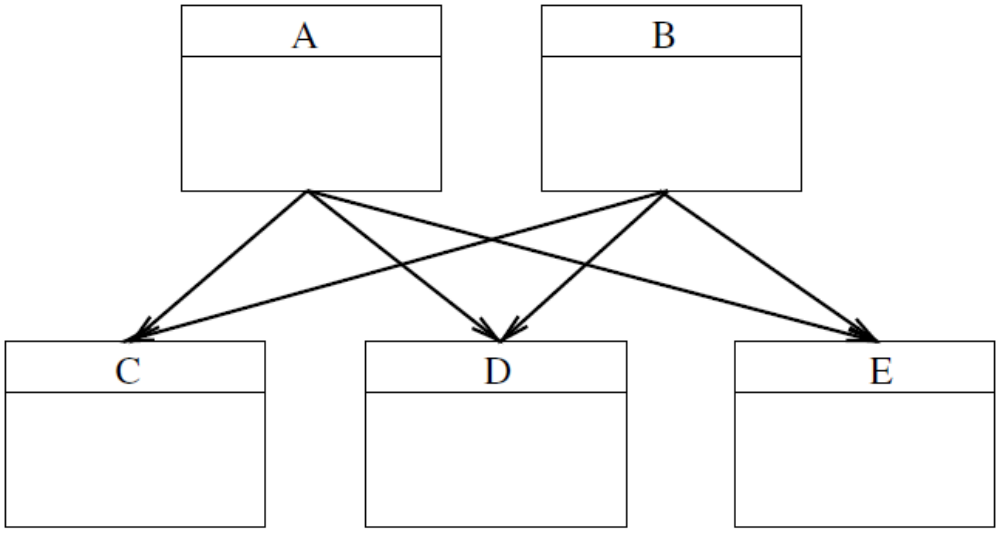
Downcasting sería:

```
Figura *f1 = new Circulo("circ",2);
Figura *f2 = new Cuadrado("cuad", 4);

Cuadrado* c1 = dynamic_cast<Cuadrado*>(f1);
Cuadrado* c2 = dynamic_cast<Cuadrado*>(f2);

if(c1!=0) c1->imprimir();
if(c2!=0) c2->imprimir();

delete f1;
delete f2;
```

```
Figura *f2 = new Cuadrado("cuad", 4);
Cuadrado& c2 = dynamic_cast<Cuadrado&>(*f2);
c2.imprimir();
delete f2;
```

9. Herencia múltiple

La herencia múltiple consiste en heredar de dos o más clases.



(+) Puede ser útil
(-) Puede provocar “problemas”, por eso otros lenguajes Orientados a Objetos no lo permiten

Ejemplo:

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

class Papa1{
public:
    int var1;
    void metodo1(){ cout<<"metodo 1"<<endl; }
};
class Papa2{
public:
    int var2;
    void metodo2(){ cout<<"metodo 2"<<endl; }
};
class Hijo: public Papa1, public Papa2{
public:
    int varHijo;
    void metodoHijo(){ cout<<"metodo H"<<endl; }
};
int main(){
    Hijo h;
    h.var1 = 10;
    h.var2 = 10;
    h.varHijo = 10;
    h.metodo1();
    h.metodo2();
    h.metodoHijo();
    cout<<sizeof(Papa1)<<endl;
    cout<<sizeof(Papa2)<<endl;
    cout<<sizeof(Hijo)<<endl;
}
```