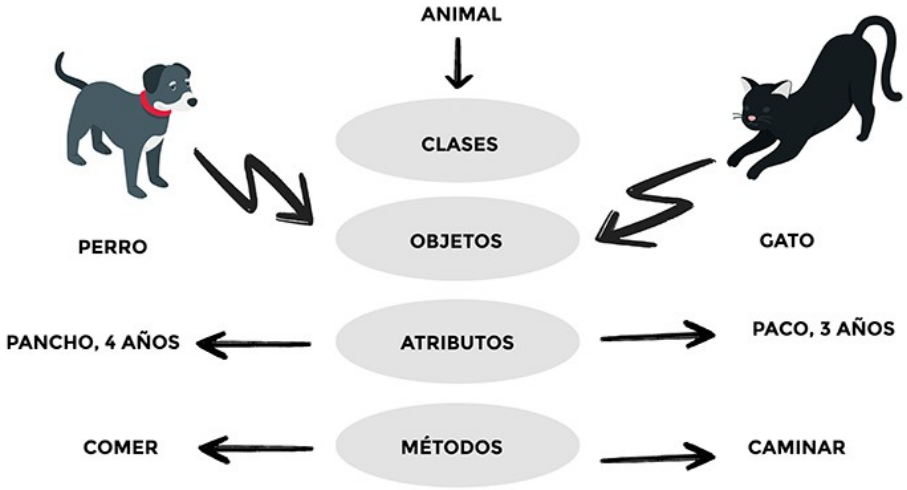
Programación orientada a obxectos

A programación orientada a obxectos (POO) busca levar á programación a filosofía dos obxectos da vida cotidiana. Básase en crear unha **clase** que terá **obxectos** cunhas características (**atributos**) e unhas accións (**métodos**) comúns. Os atributos concretánse con variables e os métodos con funcións. No seguinte exemplo temos unha clase Animal que ten dous obxectos, un perro e un gato que teñen os atributos comúns do nome e a idade e os métodos camiñar e comer.

Este proceso de determinar as características (atributos) e as accións (metodos) esenciais de algo, ignorando os detalles superfluos é o que coñecemos como **abstracción**.



Aqui temos un exemplo de como crear unha clase cos seus atributos e métodos

- As clases creanse con class e un nome que comeza con maiúscula. Dentro das chaves poñemos os atributos e os métodos. Despois das chaves poñemos punto e coma.

class Persoa{…};

- Aos atributos dámoslle a condición de private (privados, de tal xeito que só poderemos utilizar exclusivamente os atributos cos métodos desta clase.

- Dentro dos métodos debemos incluir un **método construtor** que nos permite inicializar os atributos da clase e sempre debe levar o nome da mesma, deste xeito a nosa clase sabe cantos atributos ten que recibir.

Persoa::Persoa(int _idade,string _nome){...}

- Unha vez declarada a clase, desenvolvemos a programación de todos os métodos.

- Os obxectos creamolos na función principal int main() comezando co nome da clase e a continuación o nome do obxecto e por último entre parentese os atributos do obxecto.

Persoa p1(20,"Alejandro");

- Por último neste programa, executamos o método ler no obxecto p1 e o método correr no obxecto p2.

p1.ler();
p2.correr();

```
//POO: Crear unha clase


#include<iostream>
using namespace std;

//Creamos a clase persoa cos seus atributos e os seus métodos

class Persoa{
private:   //Atributos
        int idade;
        string nome;

public:   //Métodos
        Persoa(int,string);   //Método construtor
        void ler();
        void correr();
};

//Método construtor, permite inicializar os atributos

Persoa::Persoa(int _idade,string _nome){
idade=_idade;
nome=_nome;
}

//Método ler

void Persoa::ler(){
cout<<"Son "<<nome<<" e estou lendo un libro"<<endl;
}

//Método correr

void Persoa::correr(){
cout<<"Son "<<nome<<" e estou correndo unha maraton"<<" e teño "<<idade<<" anos "<<endl;
}


int main()
{
Persoa p1(20,"Alejandro"); //Creamos o obxecto p1
Persoa p2(24,"Teresa");  //Creamos o obxecto p2
p1.ler();   //O obxecto p1 executa o método ler
p2.correr();   //O obxecto p2 executa o método correr


return 0;
}
```

Exercicio: Crea unha clase chamada Rectangulo que teña os seguintes atributos: longo e ancho, e os seguintes métodos: perimetro() e area() que calculen o perímetro e a área dos obxectos.

```
//Clase rectángulo

#include<iostream>
using namespace std;

class Rectangulo{
private:
	float longo,ancho;

public:
	Rectangulo(float,float);
	void perimetro();
	void area();

};
Rectangulo::Rectangulo(float _longo,float _ancho){
longo=_longo;
ancho=_ancho;
}

void Rectangulo::perimetro(){
float perimetro;
perimetro=2*longo+2*ancho;
cout<<"O perimetro e: "<<perimetro<<endl;
}

void Rectangulo::area(){
float area;
area=longo*ancho;
cout<<"A area e: "<<area<<endl;
}
int main()
{
float l,a;
cout<<"Introduce o longo: ";
cin>>l;
cout<<"Introduce o ancho: ";
cin>>a;
Rectangulo r1(l,a);
r1.perimetro();
r1.area();

return 0;
}
```

O método construtor non ten porque ser único, podemos inicializar os atributos con dous ou máis construtores diferentes (**sobrecarga de construtores**), teñen que ter distinto número de parámetros ou distintos tipos de parámetros. Isto aplicámolo no seguinte programa que nos permite visualizar unha data con dous tipos de entrada distintas (9,2,2024) ou (20240209)

```
//POO: Sobrecarga de construtores

#include<iostream>
using namespace std;

class Data{
private:
	int dia;
	int mes;
	int ano;
public:
	Data(int,int,int); //construtor 1
	Data(long); //construtor 2
	void amosarData();

};

//   (9,2,2024)   ---   20240209  (Dúas formas de dar as datas, necesitamos dous construtores)

//Construtor 1
Data::Data(int _dia,int _mes,int _ano){
dia=_dia;
mes=_mes;
ano=_ano;
}

//Construtor 2
Data::Data(long data){
ano=int(data/10000);
mes=int((data-ano*10000)/100);
dia=int(data-ano*10000-mes*100);
}

//Función amosardata
void Data::amosarData(){
cout<<"A data de hoxe e: "<<dia<<"/"<<mes<<"/"<<ano<<endl;
}

int main()
{
Data hoxe(9,2,2024);
Data onte(20240209);
hoxe.amosarData();
onte.amosarData();

return 0;
}
```

Exercicio: Crea unha clase Tempo que conteña os seguintes atributos enteiros: horas, minutos e segundos. Fai que a clase conteña 2 constructores, o primeiro debe ter 3 parámetros Tiempo(int,int,int) e o segundo só un campo que serán os segundos e ten que desenvolvelo en horas, minutos e segundos. O formato de saída debe ser como este exemplo 12:34:56.

```
//Tempo con sobrecarga de construtores

#include<iostream>
using namespace std;

class Tempo
{
private:
	int horas;
	int minutos;
	int segundos;
public:
	Tempo(int,int,int);
	Tempo(long);
	void amosarTempo();
};

//1º método construtor

Tempo::Tempo(int _horas,int _minutos,int _segundos){
horas=_horas;
minutos=_minutos;
segundos=_segundos;
}

//2º método construtor

Tempo::Tempo(long tempo){
horas=tempo/3600;
minutos=(tempo%3600)/60;
segundos=(tempo%3600)%60;
}

//Método amosarTempo()

void Tempo::amosarTempo(){
cout<<"O tempo e: "<<horas<<":"<<minutos<<":"<<segundos<<endl;
}


int main()
{
Tempo t1(12,20,44);
Tempo t2(3600);
t1.amosarTempo();
t2.amosarTempo();

return 0;
}
```

Destrutores

Podemos eliminar obxectos da nosa clase cos destrutores, para isto temos que crear un método destrutor con ~Can(); o cal permitiríanos eliminar os obxectos da clase Can.

```
//POO: Destrutores

#include<iostream>
using namespace std;

//Creamos a clase Can cos seus atributos e os seus métodos

class Can{
private:   //Atributos
        string nome,raza;

public:   //Métodos
        Can(string,string);   //Método construtor
        ~Can();  //Destrutor, elimina os obxectos da clase
        void amosardatos();
        void xogar();
};

//Método construtor, permite inicializar os atributos

Can::Can(string _nome,string _raza){
nome=_nome;
raza=_raza;
}
//Destrutor
Can::~Can(){
}

//Método amosar datos

void Can::amosardatos(){
cout<<"Nome: "<<nome<<endl;
cout<<"Raza: "<<raza<<endl;
}

//Método xogar

void Can::xogar(){
cout<<"O can "<<nome<<" esta xogando"<<endl;
}

int main()
{
Can can1("Draco","Labrador"); //Creamos o obxecto can1
can1.amosardatos();   //O obxecto can1 executa o método amosar datos
can1.xogar();   //O obxecto can2 executa o método xogar
can1.~Can();  //Elimina o obxecto
return 0;
}
```

Métodos construtores e modificadores (getters e setters)

Podemos utilizar os setters e getters para respectivamente establecer os valores dos atributos e para visualizar os mesmos. É facer doutro xeito o que realizábamos co método construtor, unha vantaxe que ten é que se temos 20 atributos e queremos inicializar só 10, este é un sistema máis sinxelo. Creanse con get e set e o nome da clase.

```
//POO: Getters e setters
#include<iostream>
using namespace std;

//Creamos a clase punto cos seus atributos e os seus métodos
class Punto{
private:   //Atributos
        int x,y;

public:   //Métodos
        void setPunto(int,int);   //Serven para establecer os valores dos meus atributos
        int getPuntoX();
        int getPuntoY(); //Serven para amosar os valores dos meus atributos
};
void Punto::setPunto(int _x,int _y)  //Establecemos valores aos atributos
{
x=_x;
y=_y;
}
int Punto::getPuntoX(){
        return x;
}
int Punto::getPuntoY(){
        return y;
}
int main()
{
Punto punto1; //Creamos o obxecto punto1
punto1.setPunto(15,10); //Establecemos en 15 e 10 as coordenadas do punto 1
cout<<punto1.getPuntoX()<<endl;
cout<<punto1.getPuntoY()<<endl;
return 0;
}
```

Exercicio: Crea unha clase de naves espaciais, que teña os atributos: Nome da nave, posición da coordenada x, posición da coordenada y, posición da coordenada z e número de vidas; e os métodos: Construtor, desprazamento cara a dereita no eixo das x, desprazamento cara a esquerda no eixo das x, desprazamento cara adiante no eixo das y, desprazamento cara atrás no eixo das y, desprazamento cara arriba no eixo das z, desprazamento cara abaixo no eixo das z, establecer posición, destrutor de vidas (non pode haber número de vidas negativo) e amosar estado da nave (dar posición no espazo e número de vidas). Despois faremos:

a) Crear as naves Enterprise, Endurance e Challenger situadas no orixe (0,0,0) e con tres vidas.
b) Establecer a posición da nave Enterprise en (20,35,-15).
c) Establecer a posición da nave Endurance en (10,24,78)
d) Desplazar posición da nave Enterprise dúas posicións cara arriba.
e) Eliminar unha vida da nave Challenger e dúas da Endurance.
f) Desprazar dúas posicións a nave Enterprise cara a dereita.
g) Amosar o estado das tres naves.

```
//Nave espacial
#include<iostream>
using namespace std;

class Nave{
private:
	string nome;
	int x;
	int y;
	int z;
	int vidas;
public:
	Nave(string,int,int,int,int); //construtor 1
	void subir();
	void baixar();
	void dereita();
	void esquerda();
	void adiante();
	void atras();
	void posicionar();
	void destruir();
	void amosarEstado();

};

//Construtor
Nave::Nave(string _nome,int _x,int _y,int _z,int _vidas){
nome=_nome;
x=_x;
y=_y;
z=_z;
vidas=_vidas;
}
//Subir
void Nave::subir(){
y=y+1;
}
//Baixar
void Nave::baixar(){
y=y-1;
}
//Desprazar á dereita
void Nave::dereita(){
x=x+1;
}
//Desprazar á esquerda
void Nave::esquerda(){
x=x-1;
}
//Desprazar cara adiante
void Nave::adiante(){
y=y+1;
}
//Desprazar cara atrás
void Nave::atras(){
```

```
y=y-1;
}
//Posicionar a nave
void Nave::posicionar(){
int posx,posy,posz;
cout<<"Introduce a coordenada x: ";
cin>>posx;
cout<<"Introduce a coordenada y: ";
cin>>posy;
cout<<"Introduce a coordenada z: ";
cin>>posz;
x=posx;
y=posy;
z=posz;
}
//Destruir vidas
void Nave::destruir(){
if(vidas>0)
{
vidas--;
}
}
//Amosar estado
void Nave::amosarEstado(){
cout<<"Posicion x da nave "<<nome<<" "<<x<<endl;
cout<<"Posicion y da nave "<<nome<<" "<<y<<endl;
cout<<"Posicion y da nave "<<nome<<" "<<z<<endl;
cout<<"Vidas da nave "<<nome<<" "<<vidas<<endl<<endl;
}

int main()
{
Nave Enterprise("Enterprise",0,0,0,3);
Nave Endurance("Endurance",0,0,0,3);
Nave Challenger("Challenger",0,0,0,3);
cout<<"Introduce as coordenadas da nave Enterprise: "<<endl;
Enterprise.posicionar();
cout<<"Introduce as coordenadas da nave Endurance: "<<endl;
Endurance.posicionar();
Enterprise.subir();
Enterprise.subir();
Challenger.destruir();
Endurance.destruir();
Endurance.destruir();
Enterprise.dereita();
Enterprise.dereita();
Enterprise.amosarEstado();
Endurance.amosarEstado();
Challenger.amosarEstado();
return 0;
}
```

Herencia

A herencia é un mecanismo para reutilizar e ampliar as clases existentes sen modificalas, xerando así relacións xerárquicas entre elas. Unhas clases (derivadas ou fillas) van poder heredar atributos e métodos de outra clase (base ou pai).

```
//POO: Herencia en poo, clase pai (base) Persoa e clase filla (derivada) Alumno
#include<iostream>
using namespace std;
//Creamos a clase persoa cos seus atributos e os seus métodos

class Persoa{
private:   //Atributos
        string nome;
        int idade;
public:   //Métodos
        Persoa(string,int);   //Método construtor
        void amosarpersoa();
};
class Alumno: public Persoa{  //Estamos indicando que a clase Alumno é filla da clase Persoa
private:
        string codigoalumno;
        float notafinal;
public:
        Alumno(string,int,string,float);  //Primeiro os da clase pai e despois os desta clase
        void amosaralumno();
};
//Método construtor da clase pai, permite inicializar os atributos
Persoa::Persoa(string _nome,int _idade){
nome=_nome;
idade=_idade;
}
//Método amosar persoa da clase pai-Persoa
void Persoa::amosarpersoa(){
cout<<"Nome: "<<nome<<endl;
cout<<"Idade: "<<idade<<" anos"<<endl;
}
//Método construtor da clase filla Alumno
Alumno::Alumno(string _nome,int _idade,string _codigoalumno,float _notafinal) :
Persoa( _nome,_idade){
codigoalumno=_codigoalumno;
notafinal=_notafinal;
}
//Método amosar alumno da clase filla Alumno
void Alumno::amosaralumno(){
amosarpersoa();
cout<<"Codigoalumno: "<<codigoalumno<<endl;
cout<<"notafinal: "<<notafinal<<endl;
}
int main()
{
Alumno alumno1("Fernando",54,"345d321",8.5);
alumno1.amosaralumno();
return 0;
}
```

Exercicio

Fai un programa en C++ que siga esta xerarquía (herencia amosada na figura), o único método ademais do construtor será amosar datos e os atributos son: Clase Persoa (nome, idade), clase Empleado (soldo), clase Estudante(nota final), clase Universitario (carrera).


```
#include<iostream>
#include<stdlib.h>
using namespace std;

class Persoa{
	private: //Atributos
		string nome;
		int idade;
	public: //Metodos
		Persoa(string,int); //Construtor Persona
		void amosarPersoa();
};

class Empleado : public Persoa{
	private: //Atributos
		float soldo;
	public: //Metodos
		Empleado(string,int,float); //Construtor Empleado
		void amosarEmpleado();
};

class Estudante : public Persona{
	private: //Atributos
		float notaFinal;
	public: //Métodos
		Estudante(string,int,float);
		void amosarEstudante();
};

class Universitario : public Estudante{
	private: //Atributos
		string carrera;
	public:
		Universitario(string,int,float,string); //Construtor Universitario
		void amosarUniversitario();
};

//Construtor da clase Persoa(Clase Pai)
Persoa::Persoa(string _nome,int _idade){
	nome = _nome;
```

```
	idade= _idade;
}

void Persoa::amosarPersoa(){
	cout<<"Nome: "<<nome<<endl;
	cout<<"Idade: "<<idade<<endl;
}

//Construtor da clase Empleado (Clase filla)
Empleado::Empleado(string _nome,int _idade,float _sueldo) : Persoa(_nome,_idade){
	soldo = _soldo;
}

void Empleado::amosarEmpleado(){
	amosarPersoa();
	cout<<"Soldo: "<<soldo<<endl;
}

//Constructor da clase Estudante
Estudante::Estudante(string _nome,int _idade,float _notaFinal) : Persoa(_nome,_idade){
	notaFinal = _notaFinal;
}

void Estudante::amosarEstudante(){
	amosarPersoa();
	cout<<"Nota Final: "<<notaFinal<<endl;
}
//Construtor da clase Universitario(Clase filla)
Universitario::Universitario(string _nome,int _idade,float _notaFinal,string _carreira) :
Estudiante(_nome,_idade,_notaFinal){
	carreira = _carreira;
}
void Universitario::amosarUniversitario(){
	amosarEstudante();
	cout<<"Carreira: "<<carreira<<endl;
}
int main(){
	Empleado empleado1("Juan",35,1300);
	cout<<"-Empleado-"<<endl;
	empleado1.amosarEmpleado();
	cout<<"\n";

	Estudante estudante1("Maria",21, 6.7);
	cout<<"-Estudante-"<<endl;
	estudante1.amosarEstudante();
	cout<<"\n";

	Universitario universitario1("Alejandro",20,15.6,"Informatica");
	cout<<"-Universitario-"<<endl;
	universitario1.amosarUniversitario();
	cout<<"\n";

	system("pause");
	return 0;
}
```

Polimorfismo

Un mismo método pode comportarse de distinto xeito según sexa o obxecto, pensade no método comer(), un humano, un can e unha vaca teñen este método pero os tres fanno de tres xeitos distintos. Aquí temos un exemplo, cunha clase pai Persoa e dúas clases fillas Alumno e Profesor. Ao executar o método amosar() ten que facelo distinto para cada clase, no alumno ten que aparecer a maiores a nota final e no profesor a materia.

```
//Polimorfismo en POO

#include<iostream>
using namespace std;

//Creamos a clase Persoa cos seus atributos e os seus métodos

class Persoa{
private:   //Atributos
        string nome;
        int idade;
public:   //Métodos
        Persoa(string,int);   //Método construtor
        virtual void amosar();  //Virtual indícanos que imos ter polimorfismo
};

//Método construtor clase Persoa
Persoa::Persoa(string _nome,int _idade){
nome=_nome;
idade=_idade;
}
//Método amosar clase Persoa
void Persoa::amosar(){
cout<<"Nome: "<<nome<<endl;
cout<<"Idade: "<<idade<<endl;
}

class Alumno : public Persoa{
private:
        float notaFinal;
public:
        Alumno(string,int,float);
        void amosar();
};
//Método construtor clase Alumno
Alumno::Alumno(string _nome,int _idade,float _notaFinal) : Persoa(_nome,_idade){
notaFinal=_notaFinal;
}
//Método amosar clase Alumno
void Alumno::amosar(){
Persoa::amosar();
cout<<"Nota final: "<<notaFinal<<endl;
}

class Profesor : public Persoa{
private:
        string materia;
```

```
public:
        Profesor(string,int,string);
        void amosar();
};
//Método construtor clase Profesor
Profesor::Profesor(string _nome,int _idade,string _materia) : Persoa(_nome,_idade){
materia=_materia;
}
//Método amosar clase Profesor
void Profesor::amosar(){
Persoa::amosar();
cout<<"Materia: "<<materia<<endl;
}

int main()
{
Alumno al1("Pepe",23,8.5);
Profesor pr1("Luis",63,"Historia");
al1.amosar();
cout<<"\n";
pr1.amosar();
return 0;
}
```

/*Exercicio 4: Crear un programa en C++ que teña a seguinte xerarquía de clases: Animal(Clase Pai) -> Humano e Can (Clases fillas), e facer polimorfismo co método comer(). Na clase Animal teño que poñer “eu como “, na clase Humano “na mesa, sentado nunha silla” e na clase Can “nun plato no chan”.

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

class Animal{
        private:
                int idade;
        public:
                Animal(int);
                virtual void comer();
};

class Humano : public Animal{
        private:
                string nome;
        public:
                Humano(int,string);
                void comer();
};

class Can : public Animal{
        private:
                string nome,raza;
        public:
                Can(int,string,string);
                void comer();
};
```

```
//Constructor da clase Animal
Animal::Animal(int _idade){
	idade = _idade;
}

void Animal::comer(){
	cout<<"Eu como ";
}

//Constructor da clase Humano
Humano::Humano(int _idade,string _nome) : Animal(_idade){
	nome = _nome;
}

void Humano::comer(){
	Animal::comer();
	cout<<"na mesa, sentado nunha silla"<<endl;
}

//Constructor da clase Can
 Can::Can(int _idade,string _nome,string _raza) : Animal(_idade){
	nome = _nome;
	raza = _raza;
}

void Can::comer(){
	Animal::comer();
	cout<<"nun plato no chan"<<endl;
}

int main(){
	Humano h1(21,"Luis");
	Can c1(4,"Bobby","Pastor Aleman");
	h1.comer();
	c1.comer();

	system("pause");
	return 0;
}
```