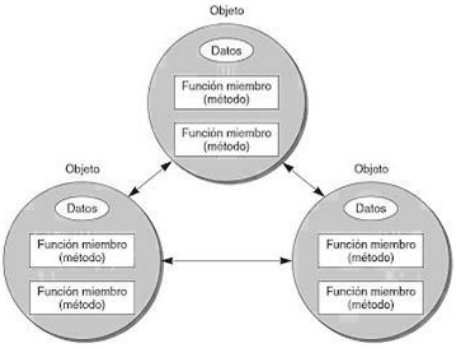
CLASES EN C++

Objetivo: *conocer los fundamentos de C++ y familiarizase con su entorno de trabajo.*

Sobrecarga de métodos



Sabemos que una **clase** está constituida por una serie de **atributos** (variables) y **métodos** (funciones). Los **atributos** deben ser ***privados*** y los **métodos** pueden ser ***públicos*** y ***privados*** según si se van a poder llamar desde fuera de la clase o no.
En C++ podemos *definir* ***varios métodos*** *con el* ***mismo nombre***. Cuando definimos dos o más métodos con el mismo nombre decimos que los estamos **sobrecargando**.
La **restricción para la sobrecarga de métodos** es que los mismos deben diferir en cantidad o tipo de parámetros. Es decir, podemos definir dos métodos con el mismo nombre pero uno tenga por ejemplo 3 parámetros y otro tenga 2 parámetros:

```
void mayor(int x1,int x2,int x3)
void mayor(int x1,int x2)
```

O que tengan la misma cantidad de parámetros pero sean de distinto tipo:

```
void mayor(int x1,int x2)
void mayor(char nombre1[40],char nombre2[40])
```

Ejemplo 1:

Crear una clase llamada **Matematica** que implemente cuatro **métodos** llamados **mayor**. El *primero* que reciba como parámetros dos enteros y devuelva el mayor de ellos. El *segundo* que reciba tres enteros y devuelva el mayor. Los mismo deben hacer los siguientes *dos métodos* pero recibiendo parámetros de tipo float.

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

class Matematica {
public:
    int mayor(int x1,int x2);
    int mayor(int x1,int x2,int x3);
    float mayor(float x1,float x2);
    float mayor(float x1,float x2,float x3);
};

int Matematica::mayor(int x1,int x2)
{
    if (x1>x2)
        return x1;
    else
        return x2;
}

int Matematica::mayor(int x1,int x2,int x3)
{
    if (x1>x2 && x1>x3)
        return x1;
    else
        if (x2>x3)
            return x2;
        else
            return x3;
```

```
}

float Matematica::mayor(float x1,float x2)
{
    if (x1>x2)
        return x1;
    else
        return x2;
}
float Matematica::mayor(float x1,float x2,float x3)
{
    if (x1>x2 && x1>x3)
        return x1;
    else
        if (x2>x3)
            return x2;
        else
            return x3;
}
int main()
{
    Matematica m1;
    cout<<"Mayor entre 6 y 8 : ";
    cout<<m1.mayor(6,8);
    cout <<"\n";
    cout<<"Mayor entre 10, 40 y 5 : ";
    cout<<m1.mayor(10,40,5);
    cout <<"\n";
    cout<<"Mayor entre 6.2  y  9.3 : ";
    cout<<m1.mayor(6.2f,9.3f);
    cout <<"\n";
    cout<<"Mayor entre 7 , 12.5  y  4.2 : ";
    cout<<m1.mayor(7.0f,12.5f,4.2f);
    cout <<"\n";
    return 0;
}
```

Podemos observar que hemos definido cuatro métodos con el mismo nombre llamado **mayor**:

```
class Matematica {
public:
    int mayor(int x1,int x2);
    int mayor(int x1,int x2,int x3);
    float mayor(float x1,float x2);
    float mayor(float x1,float x2,float x3);
};
```

Dos de los mismos ***difieren en cantidad de parámetros*** (2 y 3) y cuando tienen la misma cantidad de parámetros ***difieren en el tipo de parámetros***: int y float.

La **sobrecarga de métodos** nos facilita reducir la cantidad de nombres de métodos cuando realizan la misma actividad, si no existiera la sobrecarga estaríamos obligados a definir cuatro nombres de métodos distintos, por ejemplo:

```
class Matematica {
public:
    int mayorDosEnteros(int x1,int x2);
    int mayorTresEnteros(int x1,int x2,int x3);
    float mayorDosReales(float x1,float x2);
    float mayorTresReales(float x1,float x2,float x3);
};
```

Cuando se llama a un método sobrecargado el compilador sabe a cuál método llamar según la cantidad de parámetros que le pasamos:

```
cout<<m1.mayor(6,8);
cout<<m1.mayor(10,40,5);
```

y si hay dos métodos con la misma cantidad de parámetros analiza el tipo de datos que le pasamos:

```
cout<<m1.mayor(6,8);
cout<<m1.mayor(6.2f,9.3f);
```

Debemos agregar la letra **f** al final del valor real para ***indicar que se trata de un valor de tipo float***.

Sobrecarga del constructor

Como sabemos el **constructor** es un método de la clase y como tal ***podemos sobrecargarlo***, es decir ***definir más de un constructor***.
Cuando definimos un **objeto** de la **clase** es cuando indicamos a qué **constructor** llamaremos según nuestras necesidades.

EJEMPLO 2:

Crear una clase llamada **EstructuraVector**. Definir un atributo de tipo vector de 5 elementos enteros. Declarar **dos constructores**, uno sin parámetros que cargue el vector con valores cero y otro constructor que reciba un entero indicando el valor entero con el que deben inicializarse las componentes.

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

class EstructuraVector {
    int vec[5];
public:
    EstructuraVector();
    EstructuraVector(int valor);
    void imprimir();
};

EstructuraVector::EstructuraVector()
{
    for(int f=0;f<5;f++)
        vec[f]=0;
}

EstructuraVector::EstructuraVector(int valor)
{
        for(int f=0;f<5;f++)
        vec[f]=valor;
}
void EstructuraVector::imprimir()
{
    for(int f=0;f<5;f++)
    {
        cout <<vec[f];
        cout <<"-";
    }
    cout<<"\n\n";
}

int main()
{
    EstructuraVector v1;
```

```
    v1.imprimir();
    EstructuraVector v2(12);
    v2.imprimir();
    return 0;
}
```

Podemos ver que hemos definido **dos constructores** que **difieren** en este caso en la **cantidad de parámetros** que tienen:

```
class EstructuraVector {
    int vec[5];
public:
    EstructuraVector();
    EstructuraVector(int valor);
    void imprimir();
};
```

Ahora cuando en el main definimos objetos de la clase **EstructuraVector** debemos elegir que **constructor** llamaremos:

```
    EstructuraVector v1;
    v1.imprimir();
    EstructuraVector v2(12);
    v2.imprimir();
```

Estamos llamando al **constructor que no tiene parámetros** cuando definimos el **objeto v1** y estamos llamando al **constructor que tiene un parámetro** cuando definimos el **objeto v2**.

Colaboración de clases en C++

Normalmente en un problema resuelto con la metodología de programación orientada a objetos no interviene una sola clase, sino que hay *muchas clases que interactúan y se comunican*.

EJEMPLO 3:

Un banco tiene **3 clientes** que pueden hacer **depósitos** y **extracciones**. También el banco requiere que al final del día calcule la **cantidad de dinero que hay depositada**.
Lo primero que hacemos es identificar las clases:
Podemos identificar la clase **Cliente** y la clase **Banco**.
Luego debemos definir los atributos y los métodos de cada clase:

```
Cliente
    atributos
        nombre
        monto
    métodos
        constructor
        depositar
        extraer
        retornarMonto

Banco
    atributos
        3 Cliente (3 objetos de la clase Cliente)
    métodos
        constructor
        operar
        depositosTotales
```

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

```
class Cliente {
     char nombre[40];
     float monto;
public:
     Cliente(const char nom[40]);
     void depositar(int m);
     void extraer(int m);
     float retornarMonto();
     void imprimir();
};

class Banco {
     Cliente cliente1, cliente2, cliente3;
public:
     Banco();
     void operar();
     void depositiosTotales();
};

Cliente::Cliente(const char nom[40])
{
     strcpy_s(nombre, nom);
     monto = 0;
}

void Cliente::depositar(int m)
{
     monto = monto + m;
}

void Cliente::extraer(int m)
{
     monto = monto - m;
}

float Cliente::retornarMonto()
{
     return monto;
}

void Cliente::imprimir()
{
     cout << "Nombre:" << nombre << "  Monto:" << monto << "\n\n";
}

Banco::Banco() :cliente1("juan"), cliente2("pedro"), cliente3("luis")
{
}

void Banco::operar()
{
     cliente1.depositar(100);
     cliente2.depositar(150);
     cliente3.depositar(200);
     cliente3.extraer(150);
}

void Banco::depositiosTotales()
{
```

```
     float   t   =   cliente1.retornarMonto()   +   cliente2.retornarMonto()   +
cliente3.retornarMonto();
     cout << "El total de dinero en el banco es:" << t << "\n\n";
     cliente1.imprimir();
     cliente2.imprimir();
     cliente3.imprimir();
}

int main()
{
     Banco banco1;
     banco1.operar();
     banco1.depositiosTotales();
     return 0;
}
```

Los **atributos** de una clase normalmente son privados para que no se tenga acceso directamente desde otra clase, **los atributos son modificados por los métodos de la misma clase**:

```
class Cliente {
     char nombre[40];
     float monto;
public:
     Cliente(const char nom[40]);
     void depositar(int m);
     void extraer(int m);
     float retornarMonto();
     void imprimir();
};
```

El **constructor** recibe como parámetro el nombre del cliente (debemos agregar el modificador const ya que le pasaremos una cadena fija) y lo almacena en el atributo respectivo e inicializa el atributo monto en cero:

```
Cliente::Cliente(const char nom[40])
{
     strcpy_s(nombre, nom);
     monto = 0;
}
```

Los métodos **depositar** y **extraer** actualizan el **atributo monto** con el dinero que llega como parámetro (para simplificar el problema no hemos validado que cuando se extrae dinero el atributo monto quede con un valor negativo):

```
void Cliente::depositar(int m)
{
    monto = monto + m;
}

void Cliente::extraer(int m)
{
    monto = monto - m;
}
```

El método **retornarMonto** tiene por objetivo comunicar al **Banco** la cantidad de dinero que tiene el cliente (tengamos en cuenta que como el atributo monto es privado de la clase, debemos tener un método que lo retorne):

```
float Cliente::retornarMonto()
{
    return monto;
}
Por último el método imprimir muestra nombre y el monto de dinero del cliente:
void Cliente::imprimir()
{
    cout << "Nombre:" << nombre << "  Monto:" << monto<<"\n\n";
```

```
}
```

Como podemos observar en el **main** no definimos objetos de la **clase Cliente**. *¿Entonces donde definimos objetos de la **clase Cliente?***
La respuesta es que en la **clase Banco** definimos **tres objetos** de la **clase Cliente**.
Veamos ahora la clase **Banco** que requiere la **colaboración** de la clase **Cliente**. Primero definimos **tres atributos** de tipo **Cliente**:

```
class Banco {
    Cliente cliente1, cliente2, cliente3;
public:
    Banco();
    void operar();
    void depositiosTotales();
};
```

Para poder inicializar los **tres objetos** de la **clase Cliente** en la **clase Banco** lo debemos hacer en el **constructor**:

```
Banco::Banco() :cliente1("juan") , cliente2("pedro") , cliente3("luis")
{
}
```

Podemos observar que realmente el **código del constructor** de la **clase Banco** está vacío, pero en la parte de la **declaración de la cabecera del método** disponemos **dos puntos** y seguidamente llamamos al **constructor** de cada uno de los objetos de la **clase Cliente** pasando el nombre del cliente respectivo (sin no hacemos esto nos genera un error sintáctico ya que el **constructor** de la **clase Cliente** tiene como parámetro una cadena de caracteres)
El **método operar del banco** (llamamos a los métodos depositar y extraer de los clientes):

```
void Banco::depositiosTotales()
{
    float    t    =    cliente1.retornarMonto()    +    cliente2.retornarMonto()    +
cliente3.retornarMonto();
    cout <<"El total de dinero en el banco es:"<<t<<"\n\n";
    cliente1.imprimir();
    cliente2.imprimir();
    cliente3.imprimir();
}
```

El método **depositosTotales** obtiene el monto depositado de cada uno de los tres clientes, procede a mostrarlos y llama al **método imprimir** de cada cliente para poder mostrar el nombre y depósito:

```
void Banco::depositiosTotales()
{
    float    t    =    cliente1.retornarMonto()    +    cliente2.retornarMonto()    +
cliente3.retornarMonto();
    cout <<"El total de dinero en el banco es:"<<t<<"\n\n";
    cliente1.imprimir();
    cliente2.imprimir();
    cliente3.imprimir();
}
```

Por último en el main definimos un objeto de la **clase Banco** (la clase **Banco** es la **clase principal** en nuestro problema):

```
int main()
{
    Banco banco1;
    banco1.operar();
    banco1.depositiosTotales();
    return 0;
}
```

EJEMPLO 4:

Crear un programa que permita **jugar a los dados**. Las reglas de juego son:
se tiran tres dados si los tres salen con el mismo valor mostrar un mensaje que "gano", sino "perdió".
Lo primero que debemos hacer es identificar las clases: la clase **Dado** y la clase **JuegoDeDados**. Luego los **atributos** y los **métodos** de cada clase:

```
Dado
    atributos
        valor
    métodos
        tirar
        imprimir
        retornarValor

JuegoDeDados
    atributos
        3 Dado (3 objetos de la clase Dado)
    métodos
        jugar
```

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

class Dado {
    int valor;
public:
    void tirar();
    void imprimir();
    int retornarValor();
};

class JuegoDeDados {
    Dado dado1, dado2, dado3;
public:
    void jugar();
};

void Dado::tirar()
{
    valor = rand() % 6 + 1;
}

void Dado::imprimir()
{
    cout << "Valor del Dado:" << valor << "\n";
}

int Dado::retornarValor()
{
    return valor;
}

void JuegoDeDados::jugar()
{
    dado1.tirar();
    dado1.imprimir();
    dado2.tirar();
    dado2.imprimir();
```

```
    dado3.tirar();
    dado3.imprimir();
    if (dado1.retornarValor() == dado2.retornarValor() &&
        dado1.retornarValor() == dado3.retornarValor())
    {
        cout<<"Gano";
    }
    else
    {
        cout<<"Perdio";
    }
}

int main()
{
    srand(time(NULL));
    JuegoDeDados juego1;
    juego1.jugar();
    return 0;
}
```

La **clase Dado** define el **atributo "valor"** donde almacenamos un **valor aleatorio** que representa el número que sale al tirarlo.

```
class Dado {
    int valor;
public:
    void tirar();
    void imprimir();
    int retornarValor();
};
```

El **método tirar** almacena el valor aleatorio:

```
void Dado::tirar()
{
    valor = rand() % 6 + 1;
}
```

El **método imprimir** de la **clase Dado** muestra por pantalla el **valor del dado**:

```
void Dado::imprimir()
{
    cout << "Valor del Dado:" << valor << "\n";
}
```

Por último, el **método** que **retorna el valor del dado** (se utiliza en la otra clase para ver si los tres dados generaron el mismo valor):

```
int Dado::retornarValor()
{
    return valor;
}
```

La **clase JuegoDeDatos** define **tres atributos** de la **clase Dado** (con esto decimos que la **clase Dado colabora** con la clase **JuegoDeDados**):

```
class JuegoDeDados {
    Dado dado1, dado2, dado3;
public:
    void jugar();
};
```

No hace falta definir constructor en la **clase JuegoDeDados** ya que *no tenemos que inicializar los tres objetos* de la **clase Dado**. En el **método jugar** llamamos al **método tirar** de cada dado, pedimos que se **imprima** el valor generado y finalmente **verificamos** si se ganó o no:

```
void JuegoDeDados::jugar()
```

```
{
    dado1.tirar();
    dado1.imprimir();
    dado2.tirar();
    dado2.imprimir();
    dado3.tirar();
    dado3.imprimir();
    if (dado1.retornarValor() == dado2.retornarValor() &&
        dado1.retornarValor() == dado3.retornarValor())
    {
        cout<<"Gano";
    }
    else
    {
        cout<<"Perdio";
    }
}
```

En el **main** creamos solo un objeto de la **clase principal** (**JuegoDeDados**) y también iniciamos la *semilla de valores aleatorios* llamando a la **función srand**:

```
int main()
{
    srand(time(NULL));
    JuegoDeDados juego1;
    juego1.jugar();
    return 0;
}
```

Herencia en C++

La **herencia** significa que se pueden **crear nuevas clases** partiendo de **clases existentes**, que tendrá todas los **atributos** y los **métodos** de su **'superclase'** o **'clase padre'** y además se le podrán añadir otros **atributos** y **métodos propios**.

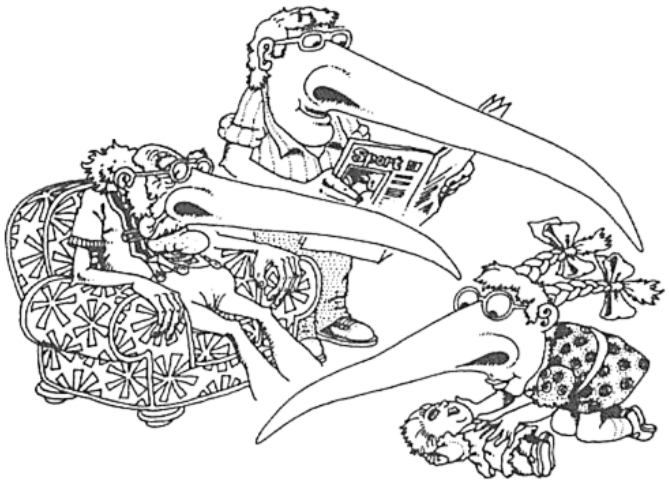
+ Clase padre o base

Clase de la que ***desciende*** o ***deriva*** una clase. Las **clases hijas** (descendientes) **heredan** (incorporan) automáticamente los **atributos** y **métodos** de la clase padre.

+ Subclase o clase derivada

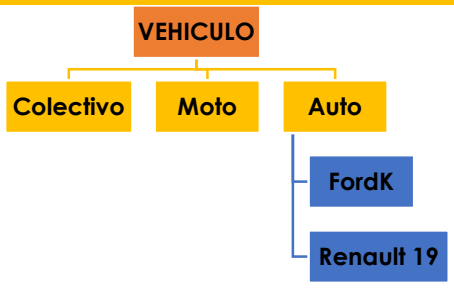
Clase que **desciende de otra**.

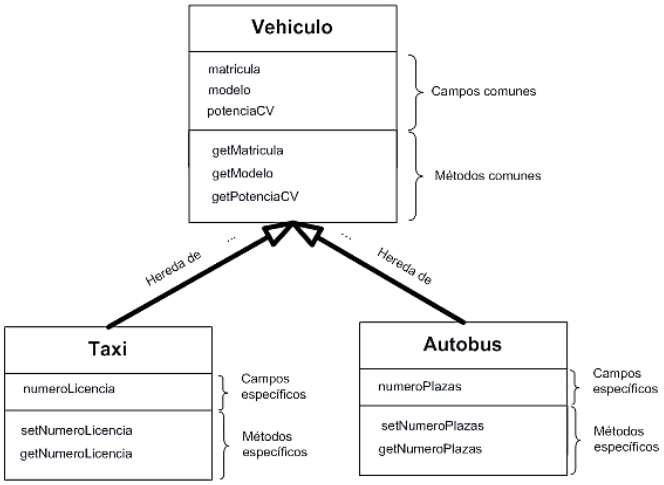
Hereda automáticamente los **atributos** y **métodos** de su **superclase**. Es una **especialización** de otra clase. Admiten la definición de **nuevos atributos** y **métodos** para aumentar la **especialización de la clase**.



Una subclase puede heredar la estructura y comportamiento de su superclase.

EJEMPLOS DE HERENCIA:





Siempre hacia abajo en la jerarquía hay una **especialización** (las **subclases** añaden nuevos **atributos** y **métodos**)

La **colaboración** es cuando una clase contiene un objeto de otra clase como atributo.

Cuando la **relación entre dos clases** es del tipo **"...*tiene un*..."** o **"...*es parte de*..."**, estamos frente a una relación de **colaboración** de clases **no** de **herencia**.
Si tenemos una **ClaseA** y otra **ClaseB** y entre ellas existe una relación de tipo **"...*tiene un*..."**, se debe declarar en la **ClaseA** un atributo de la **ClaseB**.

EJEMPLO 5:

Dada una **clase Auto**, una **clase Rueda** y una **clase Volante**. Vemos que la relación entre ellas es: **Auto** "...*tiene 4*..." **Rueda, Volante** "...*es parte de*..." **Auto**; pero la **clase Auto** no debe derivar de **Rueda** ni **Volante** de **Auto** porque la relación no es de tipo-subtipo sino de **colaboración**. Debemos declarar en la **clase Auto** 4 **atributos** de tipo **Rueda** y 1 de tipo **Volante**.
Luego si vemos que dos clases responden a la pregunta **ClaseA** "..*es un*.." **ClaseB** es posible que haya una relación de **herencia**. Por ejemplo:
Autobus "*es un*" **Vehiculo**

EJEMPLO 6:

Supongamos que necesitamos implementar **dos clases** que llamaremos **Suma** y **Resta**. Cada clase tiene como **atributo valor1, valor2** y **resultado**. Los **métodos** a definir son **cargar1** (que inicializa el atributo valor1), **carga2** (que inicializa el atributo valor2), **operar** (que en el caso de la **clase "Suma"** suma los dos atributos y en el caso de la **clase "Resta"** hace la diferencia entre **valor1** y **valor2**, y otro método **mostrarResultado**)
Si analizamos ambas clases encontramos que muchos atributos y métodos son idénticos. En estos casos es bueno definir una **clase padre** que agrupe dichos **atributos** y **responsabilidades comunes**.
La relación de **herencia** es:

Solamente el **método operar** es distinto para las clases **Suma** y **Resta** (esto hace que no lo podamos disponer en la **clase Operacion**), luego los **métodos cargar1, cargar2** y **mostrarResultado** son idénticos a las dos clases, esto hace que podamos disponerlos en la **clase Operacion**. Lo mismo los **atributos valor1, valor2** y **resultado** se definirán en la **clase padre Operacion**.

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

class Operacion {
```

```
protected:
    int valor1;
    int valor2;
    int resultado;
public:
    void cargar1();
    void cargar2();
    void mostrarResultado();
};

class Suma: public Operacion{
public:
    void operar();
};

class Resta : public Operacion {
public:
    void operar();
};

void Operacion::cargar1()
{
    cout << "Ingrese primer valor:";
    cin >> valor1;
}

void Operacion::cargar2()
{
    cout << "Ingrese segundo valor:";
    cin >> valor2;
}

void  Operacion::mostrarResultado()
{
    cout <<resultado<<"\n";
}

void Suma::operar()
{
    resultado = valor1 + valor2;
}

void Resta::operar()
{
    resultado = valor1 - valor2;
}

int main()
{
    Suma suma1;
    suma1.cargar1();
    suma1.cargar2();
    suma1.operar();
    cout << "La suma de los dos valores es:";
    suma1.mostrarResultado();

    Resta resta1;
    resta1.cargar1();
    resta1.cargar2();
    resta1.operar();
    cout << "La diferencia de los dos valores es:";
```

```
    resta1.mostrarResultado();

    return 0;
}
```

La **clase Operación** define los tres atributos:

```
class Operacion {
protected:
    int valor1;
    int valor2;
    int resultado;
public:
    void cargar1();
    void cargar2();
    void mostrarResultado();
};
```

Ya veremos que definimos los atributos con este nuevo modificador de acceso (**protected**) para que la **subclase** tenga acceso a dichos **atributos**. Si los definimos **private** las **subclases** no pueden acceder a dichos atributos.

Los **métodos** de la **clase Operacion** son:

```
void Operacion::cargar1()
{
    cout << "Ingrese primer valor:";
    cin >> valor1;
}

void Operacion::cargar2()
{
    cout << "Ingrese segundo valor:";
    cin >> valor2;
}

void  Operacion::mostrarResultado()
{
    cout <<resultado<<"\n";
}
```

Ahora veamos cómo es la **sintaxis** para indicar que **una clase hereda de otra**:

```
class Suma: public Operacion{
public:
    void operar();
};
```

Utilizamos el caracter **dos puntos** y seguidamente el **nombre de la clase padre** precedido por el **modificador public** (con esto estamos indicando que todos los **métodos** y **atributos** de la **clase Operación** son también métodos de la **clase Suma**)

Luego la característica que añade la **clase Suma** es el siguiente **método**:

```
void Suma::operar()
{
    resultado = valor1 + valor2;
}
```

El **método operar** puede acceder a los **atributos heredados** (siempre y cuando los mismos se declaren **protected**, en caso que sean **private** si bien lo **hereda** de la **clase padre** solo los pueden **modificar métodos** de dicha **clase padre**)

Ahora podemos decir que la **clase Suma** tiene **cuatro métodos** (tres heredados y uno propio) y **3 atributos** (todos heredados)

Luego en el **main** creamos un objeto de la **clase Suma** y otro de la **clase Resta**:

```
int main()
{
    Suma suma1;
    suma1.cargar1();
    suma1.cargar2();
```

```
    suma1.operar();
    cout << "La suma de los dos valores
es:";
    suma1.mostrarResultado();

    Resta resta1;
    resta1.cargar1();
    resta1.cargar2();
    resta1.operar();
    cout << "La diferencia de los dos
valores es:";
    resta1.mostrarResultado();

    return 0;
}
```

Podemos llamar tanto al **método propio** de la **clase Suma "operar()"** como a los **métodos heredados**. Quien utilice la **clase Suma** solo debe conocer qué **métodos públicos** tiene (independientemente que pertenezcan a la **clase Suma** o a una **clase superior**), no podemos acceder a los **métodos privados** y **protegidos**.
La lógica es similar para declarar la **clase Resta**.
La **clase Operación** agrupa en este caso un conjunto de **atributos** y **métodos comunes** a un **conjunto de subclases** (**Suma, Resta**). No tiene sentido definir objetos de la **clase Operacion**.
El **nivel de acceso protegido**, se comporta de manera similar al **nivel de acceso privado** con la diferencia de que permite que los miembros de clase con este nivel de acceso puedan ser heredados a clases derivadas.

Es importante aclarar que los miembros de una clase con **acceso privado NO son heredables**, como tampoco lo son los **constructores, destructores, clases** y **funciones amigas**, y **operadores sobrecargados**.

El planteo de **jerarquías de clases** es una tarea compleja que requiere un perfecto entendimiento de **todas las clases que intervienen en un problema**, cuáles son sus **atributos** y **responsabilidades**.
Existen 3 tipos de herencia en C++:

- **Herencia pública:** todos los **miembros públicos** y **protegidos** de la **clase base** conservan esos **mismos niveles de acceso** respectivamente en las **clases derivadas**.

```
class ClaseDerivada : public ClaseBase
```

- **Herencia protegida:** todos los **miembros públicos** de la **clase base** adquieren el **nivel de acceso protegido** en las **clases derivadas**, mientras que los miembros protegidos conservan su nivel de acceso. Lo anterior indica que una **clase derivada** puede luego **heredar** a **otra clase** los **miembros protegidos** que heredó de su **clase base**.

```
class ClaseDerivada : protected ClaseBase
```

- **Herencia privada:** todos los **miembros públicos** y **protegidos** de la **clase base** adquieren el **nivel de acceso privado** en las **clases derivadas**. De ahí se desprende que una **clase derivada** que haya heredado mediante **herencia privada** no puede heredar a otras clases los miembros que ha heredado de otras clases.

```
class ClaseDerivada : private ClaseBase
```

Tipos de herencia en C++ de acuerdo con la estructura de jerarquía

- **Herencia simple:** se da cuando una **clase derivada hereda** solo de una **clase base** y de igual modo la **clase base** no hereda a ninguna otra clase.

```
class BaseA
{
};
class DerivadaBdeA: public BaseA
{
};
```

- **Herencia múltiple:** Ocurre cuando una **clase derivada** hereda de más de una **clase base** al mismo tiempo.

```
class BaseA
{
};
class BaseB
{
};
class DerivadaCdeAyB: public BaseA, public BaseB
{
};
```

- **Herencia multinivel:** se da cuando una **clase derivada** 'X' hereda de una **clase base** a través de otra **clase intermediaria** que actúa como **clase derivada** para la **clase base original** y como **clase base** para la **clase derivada** 'X'.

```
class BaseA
{
};
class DerivadaBdeA: public BaseA
{
};
class DerivadaCdeB: public DerivadaBdeA
{
};
```

- **Herencia jerárquica:** Ocurre cuando varias clases derivadas heredan de una clase base en común.

```
class BaseA
{
};
class DerivadaBdeA: public BaseA
{
};
class DerivadaCdeA: public BaseA
{
};
```

- **Herencia híbrida:** Este tipo de herencia ocurre cuando se combinan dos o más tipos de las herencias mencionadas anteriormente.

```
class BaseA
{
};
class BaseB
{
};
class DerivadaBdeA: public BaseA
{
};
class    DerivadaCdeAyB:    public
DerivadaBdeA, public BaseB
{
};
```

ACTIVIDADES

1. Dado el siguiente código:

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

class ClaseBase
{
    protected:
    int unaVar = 0;
    public:
    ClaseBase(int x):unaVar(x){}
    void unMetodo(void)
    {
        cout<<"unaVar = "<<unaVar<<endl;
    }
};

class ClaseDerivada : public ClaseBase
{
    public:
    ClaseDerivada(int x):ClaseBase(x){} /* Ejecución del constructor de la clase base para
inicializar a unaVar */
};

int main()
{
    ClaseDerivada obj1(50); /* Aquí el constructor de la clase derivada invoca el
constructor de la clase base */
    obj1.unMetodo();
    return 0;
}
```

I. ¿Compila exitosamente el siguiente programa? ¿Porque?
II. ¿Qué resultados se obtiene luego su ejecución?

2. Realizar en el constructor la carga de datos. Definir otros dos métodos para imprimir los datos ingresados y un mensaje si es mayor o no de edad (edad >=18).
Implementar una clase que permita cargar un vector de 5 elementos. Definir tres métodos sobrecargados que :

```
void imprimir(); //imprime todo el vector
void imprimir(int hasta);//imprime desde el principio del vector hasta el valor que le
pasamos
void imprimir(int desde,int hasta);//imprime un rango de valores del vector.
```

3. Plantear una clase llamada Punto con dos atributos llamados x e y. Definir dos constructores uno sin parámetros donde cargue en los atributos x e y el valor cero y otro con dos parámetros que cargue los atributos x e y con los valores que llegan al constructor. Imprimir los valores de los atributos.

4. Plantear una clase Club y otra clase Socio. La clase Socio debe tener los siguientes atributos privados: nombre y la antigüedad en el club (en años). En el constructor pedir la carga del nombre y su antigüedad. La clase Club debe tener como atributos 3 objetos de la clase Socio. Definir una responsabilidad para imprimir el nombre del socio con mayor antigüedad en el club.

5. Confeccionar una clase Persona que tenga como atributos el nombre y la edad. Definir como responsabilidades un método que cargue los datos personales y otro que los imprima. Plantear una segunda clase Empleado que herede de la clase Persona. Añadir un atributo sueldo y los métodos de cargar el sueldo e imprimir su sueldo. Definir un objeto de la clase Persona y llamar a sus métodos. También crear un objeto de la clase Empleado y llamar a sus métodos.