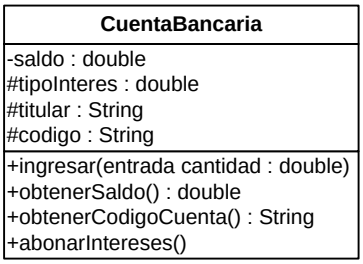
Tema 4.

Anexo I. Estudio de la implementación de la herencia, clases abstractas y polimorfismo en C++.

Para ver como se implementa la herencia, clases abstractas y el polimorfismo vamos a utilizar el dominio de información de cuentas bancarias empleado en el Tema 2. En nuestra práctica del Tema 2, nuestro programa manipulaba una clase denominada ***CuentaBancaria.*** La clase ***CuentaBancaria*** se muestra en el siguiente diagrama UML.

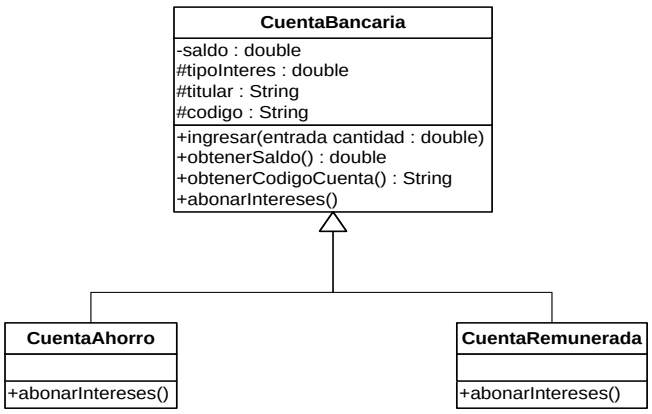


Supongamos que necesitamos manejar dos tipos de cuentas bancarias: ***CuentaAhorro y CuentaRemunerada***. Las características de cada cuenta son las siguientes:

- **Cuenta Remunerada**
  - En este tipo de cuenta **se abonan intereses siempre** independientemente del saldo
- **CuentaAhorro**
  - En este tipo de cuenta **se abonan intereses sólo sí el saldo es superior a 3000** euros.

Como se puede observar el método ***abonarIntereses()*** tiene un comportamiento que está en función del tipo de cuenta. Es decir es diferente para cada subclase. Este método debe ser **sobrescrito** en cada subclase a fin de dar la implementación adecuada.

Analizando estas clases obtenemos el siguiente diagrama de clases UML



1. Implementación de la Generalización/especialización en C++.

La definición de la clase base **CuentaBancaria.h** es la siguiente:

```
#include <stdlib.h>
#include <string>
using namespace std;
class CuentaBancaria
{
    private:
              double saldo;
    protected:
         string titular;
         double tipoInteres;
         string codigo;
     //Constructor
      CuentaBancaria(double saldo, string titular, string codigo, double tipoInteres) {
        this->saldo = saldo;
        this->titular = titular;
        this->codigo = codigo;
        this->tipoInteres = tipoInteres;
    }
    public:
    //Método para ingresar dinero en la cuenta
            void ingresar(double cantidad){saldo+=cantidad;}
    //Método para obtener el saldo
            double obtenerSaldo(){return saldo;}
    //Método para obtener el código de identificación
              string obtenerCodigoCuenta(){return codigo;}
    //Método abstracto que necesita ser sobrescrito por cada subclase
            virtual   void abonarIntereses()=0;
               ~CuentaBancaria();
};
```

Observe con atención el siguiente método de la clase **CuentaBancaria**

```
virtual   void abonarIntereses()=0;
```

El método se define como abstracto (palabra reservada **virtual**) y no se le proporciona cuerpo, esto indica que no tiene ninguna implementación (su cuerpo esta vacio) y se indica con ”=0”, la responsabilidad de la implementación de este método corresponde a cada subclase.

Importante: Debido a que **CuentaBancaria** define un método abstracto la clase en sí se convierte en una clase abstracta: Una clase abstracta en C++ es aquella que al menos define una **función virtual pura.**

1.1. Intento de instancia de una clase abstracta

Si intentamos realizar una instancia de la clase ***CuentaBancaria*** el compilador nos avisara que no se puede instanciar una clase abstracta.

```
int main(int argc, char *argv[])

{

    CuentaBancaria *c001=new CuentaBancaria(100,"Gustavo","C00023",3.2);

    system("PAUSE");

    return EXIT_SUCCESS;

}
```

1.2. Definición de una subclase en C++

Sintaxis general:

```
 class <NombreSubclase> : <NombreClaseBase>{

//Cuerpo de la clase

}
```

El carácter **“:”** indica que la clase *<NombreSubclase>* hereda de la clase denominada *<NombreClaseBase>*.

Subclase ***CuentaAhorro.h***

```
#include <string>

#include "cuentabancaria.h"

using namespace std;

class CuentaAhorro : CuentaBancaria {

       public:

          CuentaAhorro(double saldo, string titular, string codigo, double tipoInteres);

               void abonarIntereses();

        ~CuentaAhorro();

};
```

CuentaAhorro.cpp

```
#include "cuentaahorro.h"

CuentaAhorro::CuentaAhorro(double saldo, string titular, string codigo, double
tipoInteres):CuentaBancaria(saldo,titular,codigo,tipoInteres)

{

}

void CuentaAhorro::abonarIntereses(){

     //Abonar intereses sólo si el saldo es superior a 3000 euros

        if (this->obtenerSaldo()>=3000.0){

           double cant=this->obtenerSaldo()*(tipoInteres/100);

           this->ingresar(cant);

        }

}

CuentaAhorro::~CuentaAhorro()

{

}
```

Definición Subclase **CuentaRemunerada.h**

```
#include "cuentabancaria.h"

class CuentaRemunerada:  CuentaBancaria

{

public:

CuentaRemunerada(double saldo, string titular, string codigo, double tipoInteres);

void abonarIntereses();

 ~CuentaRemunerada();

};
```

CuentaRemunerada.cpp

```
#include "cuentaremunerada.h"

CuentaRemunerada::CuentaRemunerada(double saldo, string titular, string codigo, double
tipoInteres):CuentaBancaria(saldo,titular,codigo,tipoInteres)

{

}

void CuentaRemunerada::abonarIntereses()

{

  //Abonar intereses siempre
```

```
    double cant=this->obtenerSaldo()*(tipoInteres/100);

    this->ingresar(cant);

}

CuentaRemunerada::~CuentaRemunerada()

{

}
```

Prueba de la subclase ***CuentaAhorro y CuentaRemunerada***

```
#include <cstdlib>

#include <iostream>

#include "cuentabancaria.h"

#include "cuentaahorro.h"

#include "cuentaremunerada.h"

using namespace std;

int main(int argc, char *argv[])

{

    CuentaAhorro *c001=new CuentaAhorro(1000,"Luis Eduardo Abad Diaz","C00023",3.2);

    CuentaRemunerada *c002=new CuentaRemunerada(3000,"Alfonso Garcia Ruiz","C0043",5.0);

    //ingresar dinero en cuentas

    c001->ingresar(1000);

    c002->ingresar(3000);

    //abonar intereses en cuentas

    c001->abonarIntereses();

    c002->abonarIntereses();

    cout<<"saldo cuenta:"<<c001->obtenerCodigoCuenta()<<": "<<c001-
>obtenerSaldo()<<endl;

    cout<<"saldo cuenta:"<<c002->obtenerCodigoCuenta()<<": "<<c002-
>obtenerSaldo()<<endl;

    system("PAUSE");

    return EXIT_SUCCESS;

}
```

Salida:

```
saldo cuenta:C00023: 2000.0

saldio cuenta:C0043: 6300.0
```

Como se puede comprobar la subclase ha heredado los atributos y métodos de la clase base ***CuentaBancaria***

1.3. Análisis de miembros heredados por una subclase

Las subclases ***CuentaAhorro y CuentaRemunerada*** han heredado atributos y comportamiento de la clase base **CuentaBancaria**. Además cada subclase **deber proporcionar una implementación al método abonarIntereses().Pero** ¿Qué hereda exactamente la subclase ***CuentaAhorro*** y ***CuentaRemunerada***? La siguiente tabla muestra un resumen de todo lo heredado por cada subclase.

Atributos	**Métodos**
private double saldo; protected String titular; protected String codigo; protected double tipoInteres;	void ingresar(double cantidad) double obtenerSaldo() String obtenerCodigoCuenta()
Obligación de sobrescribir el método abonarIntereses()	

Tabla herencia CuentaBancaria y subclases

IMPORTANTE: Los **constructores de una clase base** no son heredados por las subclases. Como se ve en el constructor de la clase base la inicialización correcta de una subclase implica inicializar la clase base primero. Esto se realiza con la siguiente llamada a *super(lista parametros)*.

```
CuentaRemunerada::CuentaRemunerada(double saldo, string titular, string codigo, double
tipoInteres):CuentaBancaria(saldo,titular,codigo,tipoInteres)

{

}
```

Para **instanciar una clase base desde una subclase** en general utiliza la siguiente sintaxis:

```
public <NombreSubclase>(lista parámetros):ContructorClaseBase

{

//Cuerpo del constructor de la subclase

}
```

1.4. Acceso a los miembros de una clase base

Los **métodos de la subclase** no tienen acceso a los **miembros privados** de la clase base, pero sí a los miembros protegidos y públicos. Por ejemplo en el método ***abonarIntereses()*** de la subclase ***CuentaRemunerada*** no tiene acceso directo al atributo **saldo** ya que este es declarado privado (**prívate)** en la clase base ***CuentaBancaria***. Para acceder al atributo saldo la subclase debe usar el método **obtenerSaldo() como se observa en el siguiente fragmento.**

```
void CuentaRemunerada::abonarIntereses()

{

  //Abonar intereses siempre

    double cant=this->obtenerSaldo()*(tipoInteres/100);

    this->ingresar(cant);

}
```

Sin embargo la subclase si tiene acceso al atributo **tipoInteres** de la clase base ya que este es declarado como protegido (**protected).**

Importante: La restricción prívate en los atributos de una clase base puede sorprender pero es necesaria para mantener la propiedad de encapsulación de la información. Si se pudiese tener acceso a los miembros privados de una clase, simplemente se conseguiría heredando. De esta forma se cumple que el interfaz de una clase es el único medio de manipulación de la clase. Luego como regla general es un mejor establecer el modificador **private** para los atributos de una clase base y que las subclases accedan a ellos a través de una interfaz

Tabla general de visibilidad de miembros entre clases base y subclases:

	Un miembro declarado en una clase como			
Puede ser accedido desde	**privado**	**predeterminado**	**protegido**	**público**
Su misma clase	Sí	Sí	Sí	Sí
Cualquier clase o subclase de su mismo paquete	No	Sí	Sí	Sí
Cualquier clase de otro paquete	No	No	No	Sí
Cualquier subclase de otro paquete	No	No	Sí	Sí

Tabla. Javier Ceballos (Java 2)

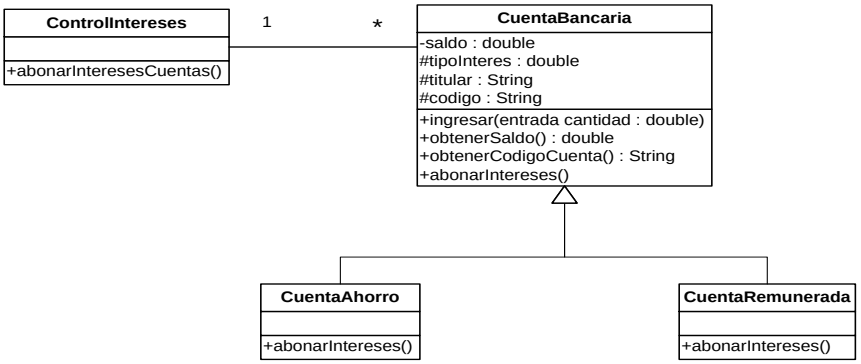
1.5. Estudio del uso del polimorfismo en C++

En ejemplo anterior declarábamos una clase base denominada ***CuentaBancaria***. Esta clase era abstracta por que definía un método virtual puro (=0) que no tenía implementación **virtual void abonarIntereses()**. La implementación adecuada de **abonarIntereses()** se proporciona por cada una de las subclases que heredan de esta clase base.

Vamos a suponer que tenemos una clase denominada ***ControlIntereses*** que expone un único método denominado **abonarInteresesCuentas()** con una lógica de negocio como lo siguiente:

Todos los días uno de cada mes se procede a abonar los interés correspondientes a cada una de las cuentas existente en el banco.

El diseño de clases en UML podría ser el siguiente:



El método **abonarInteresesCuentas()** tiene algunas consideraciones de diseño importantes:

1. La lógica de control de este método considera todas las cuentas existentes de manera homogénea y a para todas aplica el mismo tratamiento **abonarIntereses()**.

2. La lógica de **abonoIntereses() es propia de cada subclase e independiente** de este método.
3. Si en el futuro se añade una nueva cuenta este modulo no debería sufrir cambios.

Definición de la clase ***ControladorIntereses.h***

```
#include <list.h>

#include "cuentaBancaria.h"

class ControladorIntereses

{

       private:

            list<CuentaBancaria *> cuentas;

    public:


               ControladorIntereses();

               void registrarCuenta(CuentaBancaria *cb);

               void abonarInteresesCuentas();

               ~ControladorIntereses();

};
```

ControladorIntereses.cpp

```


#include "controladorintereses.h"

#include "cuentabancaria.h"

ControladorIntereses::ControladorIntereses()

{

}

void ControladorIntereses::registrarCuenta(CuentaBancaria *cb){



     cuentas.push_front(cb);

}

void ControladorIntereses::abonarInteresesCuentas()

{

     //itereador_lista se posiciona al principio de la lista

     list<CuentaBancaria *>::iterator iterador_lista=cuentas.begin();

     CuentaBancaria *cb;

     //Recorrer cada elemento de la lista

     for (;iterador_lista!=cuentas.end();iterador_lista++){
```

```
          cb=*iterador_lista; //Asignar el contenido del iterador

           cb->abonarIntereses();

      }

}

ControladorIntereses::~ControladorIntereses()

{

}
```

Ahora probamos el polimorfismo

```
#include <cstdlib>

#include <iostream>

#include "cuentabancaria.h"

#include "cuentaahorro.h"

#include "cuentaremunerada.h"

#include "controladorintereses.h"

using namespace std;



int main(int argc, char *argv[])

{

    //crear cuentas bancarias

        CuentaAhorro *ca001=new CuentaAhorro(2000,"Luis Eduardo Abad Diaz","CA001",3.2);

        CuentaAhorro *ca002=new CuentaAhorro(4000,"Manuel García Solis","CA002",3.2);

        CuentaRemunerada *cr001=new CuentaRemunerada(3000,"Alfonso Garcia
Ruiz","CR003",5.0);



        //crear el controlador

        ControladorIntereses *ctrl= new ControladorIntereses();

        //Registrar cada una de las cuentas en el controlador

        ctrl->registrarCuenta(ca001);

        ctrl->registrarCuenta(ca002);

        ctrl->registrarCuenta(cr001);

        //abonar intereses en todas las cuentas

        ctrl->abonarInteresesCuentas();

        //Comprobar el pago de intereses

        cout<<"saldo cuenta:"<<ca001->obtenerCodigoCuenta()<<": "<<ca001-
>obtenerSaldo()<<endl;
```

```
        cout<<"saldo cuenta:"<<ca002->obtenerCodigoCuenta()<<": "<<ca002-
>obtenerSaldo()<<endl;

        cout<<"saldo cuenta:"<<cr001->obtenerCodigoCuenta()<<": "<<cr001-
>obtenerSaldo()<<endl;

     system("PAUSE");

    return EXIT_SUCCESS;

}
```