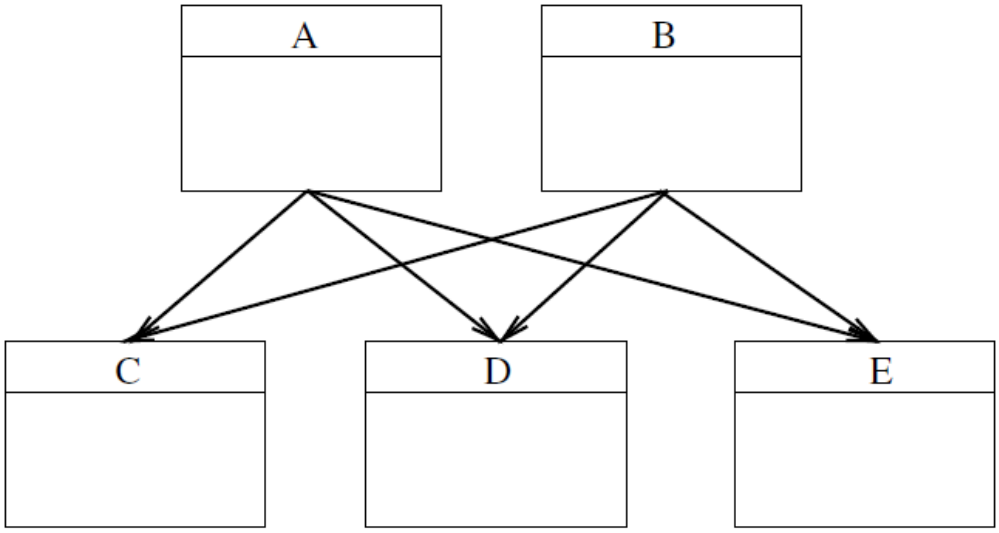
9. Herencia múltiple

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;
}
```

9.1 Ambigüedad

El problema se da cuando hay ambigüedad en las variables o métodos de las clases Base.

Ejemplo:

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

class Papa1{
public:
    int var;
    void metodo(){ cout<<"metodo papa1"<<endl; }
};
class Papa2{
public:
    int var;
    void metodo(){ cout<<"metodo papa2"<<endl; }
};
class Hijo: public Papa1, public Papa2{
public:
    int varHijo;
    void metodoHijo(){ cout<<"metodo hijo"<<endl; }
};

int main(){
    Hijo h;
    h.var = 10;
    h.varHijo = 10;
    h.metodo();
    h.metodoHijo();
}
```

```
Error de compilación por ambiguedad en la variable var y en el metodo metodo()
```

Pero se puede resolver especificando cual variable y cuál método son los que se van a utilizar.

Ejemplo:

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

class Papa1{
public:
    int var;
    void metodo(){ cout<<"metodo papa1"<<endl; }
};
class Papa2{
public:
    int var;
    void metodo(){ cout<<"metodo papa2"<<endl; }
};
class Hijo: public Papa1, public Papa2{
public:
    using Papa1::var;
    using Papa1::metodo;
    int varHijo;
    void metodoHijo(){ cout<<"metodo hijo"<<endl; }
};
int main(){
    Hijo h;
    h.var = 10;
    h.varHijo = 10;
    h.metodo();
    h.metodoHijo();
}
```

9.2 Incialización

La inicialización de los objetos se hace de la forma tradicional:

- Primero se llaman los constructores de las clases Base, en el orden como se declaró la herencia
- Se puede mandar llamar a los constructores después de la firma del constructor de la clase hija

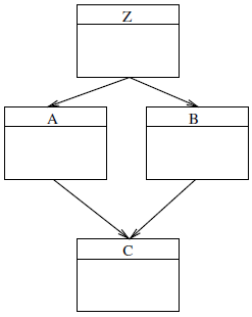
La destrucción se hace en el orden inverso de la construcción

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

class Papa1{
public:
    Papa1(){ cout<<"Papa1()"<<endl; }
    ~Papa1(){ cout<<"~Papa1()"<<endl; }
};
class Papa2{
public:
    Papa2(){ cout<<"Papa2()"<<endl; }
    ~Papa2(){ cout<<"~Papa2()"<<endl; }
};
class Hijo: public Papa1, public Papa2{
public:
    Hijo() : Papa2(), Papa1(){ cout<<"Hijo()"<<endl; }
    ~Hijo(){ cout<<"~Hijo()"<<endl; }
};

int main(){
    Hijo h;
}
```

```
Papa1()
Papa2()
Hijo()
~Hijo()
~Papa2()
~Papa1()
```

9.2 Base común

Otro “problema” es cuando se hereda de dos clases con base común



Ejemplo:

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

class Abuelo{
   public: int abuelo;
};
class Papa1: public Abuelo{
   public: int papa1;
};
class Papa2: public Abuelo{
   public: int papa2;
};
class Hijo: public Papa1, public Papa2{
};

int main(){
    Hijo h;
    h.abuelo = 10;
}
```

```
Error de compilación por ambiguedad en la variable abuelo
```

¿Qué pasa en la memoria?

Abuelo
int: abuelo

Papa1
int: abuelo
int: papa1

Papa2
int: abuelo
int: papa2

Hijo
int: abuelo
int: papa1
int: abuelo
int: papa2

Pero se resuelve haciendo que la herencia sea de forma virtual, esto lo que hace es buscar si en el árbol de herencia existe una base en común, y de ser así, sólo incluye una vez los métodos o variables de la clase en común.

Ejemplo:

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

class Abuelo{
   public: int abuelo;
};
class Papa1: virtual public Abuelo{
   public: int papa1;
};
class Papa2: virtual public Abuelo{
   public: int papa2;
};
class Hijo: public Papa1, public Papa2{
};

int main(){
    Hijo h;
}
```

¿Qué pasa en la memoria?

Ejemplo:

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

class Abuelo{
   public:  Abuelo(){ cout<<"Abuelo()"<<endl; }
};
class Papa1: public Abuelo{
   public:  Papa1(){ cout<<"Papa1()"<<endl; }
};
class Papa2: public Abuelo{
   public:  Papa2(){ cout<<"Papa2()"<<endl; }
};
class Hijo: public Papa1, public Papa2{
   public:  Hijo(){ cout<<"Hijo()"<<endl; }
};
int main(){
    Hijo h;
}
```

```
Abuelo()
Papa1()
Abuelo()
Papa2()
Hijo()
```

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

class Abuelo{
   public:  Abuelo(){ cout<<"Abuelo()"<<endl; }
};
class Papa1: virtual public Abuelo{
   public:  Papa1(){ cout<<"Papa1()"<<endl; }
};
class Papa2: virtual public Abuelo{
   public:  Papa2(){ cout<<"Papa2()"<<endl; }
};
class Hijo: public Papa1, public Papa2{
   public:  Hijo(){ cout<<"Hijo()"<<endl; }
};
int main(){
    Hijo h;
      }
```

```
Abuelo()
Papa1()
Papa2()
Hijo()
```

9.3 Herencia múltiple con clases abstractas puras (interfaces)

Una forma de utilizar herencia múltiple sin tener “problemas” es utilizando clases abstractas puras, lo que en otros lenguajes de programación orientada a objetos les llamamos interfaces.

Ejemplo:

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

class Dibujable{
   public: virtual void dibujar() = 0;
};
class Figura{
   public:
     void imprimir(){
         cout<<nombre();
         cout<<" area: "<<calcularArea();
         cout<<" perimetro:"<<calcularPerimetro();
         cout<<endl; dibujar();
     }
     virtual string nombre() = 0;
     virtual void dibujar() = 0;
     virtual float calcularArea() = 0;
     virtual float calcularPerimetro() = 0;
};
class Cuadrado: public Figura, public Dibujable{
     float lado;
   public:
     Cuadrado(float l){ lado = l; }
     string nombre(){ return "Cuadrado"; }
     float calcularArea(){ return lado*lado; }
     float calcularPerimetro(){ return 4*lado; }
     void dibujar(){
         for(int i=1;i<=lado;i++){
             for(int j=1;j<=lado;j++)
                 cout<<"*";
             cout<<endl;
         }
     }
};


int main(){
    Cuadrado c(2);
    c.imprimir();
}
```