Universidad de Murcia

Material didáctico para la asignatura Sistemas Inteligentes de 3º de Grado en Informática

AUTORES:

- José Manuel Cadenas Figueredo
- María del Carmen Garrido Carrera
- Raquel Martínez España
- Santiago Paredes Moreno

Capítulo 3

Herencia en C++

3.1 Introducción

La herencia en un lenguaje de programación orientado a objetos consiste en crear una nueva clase, llamada clase derivada, a partir de otra clase. La clase derivada "hereda", es decir, obtiene todas las propiedades de la clase original y además puede añadir propiedades nuevas. La herencia es una herramienta bastante potente en muchos aspectos en el desarrollo de aplicaciones y podríamos afirmar que es el principio de la programación orientada a objetos. Entre las ventajas de la herencia nos encontramos con que una mejor organización del diseño en las aplicaciones, permite la reutilización de código y en ciertas situaciones mejora el mantenimiento de la aplicación.

Tras la definición debemos de aclarar algunos conceptos, por ejemplo, la clase derivada es la nueva clase que se crea y que hereda de una clase original, a la que llamaremos clase base. Una clase derivada puede convertirse en una clase base ya que a partir de una clase derivada se puede seguir obteniendo clases nuevas. Además, una clase derivada puede ser derivada de más de una clase base. Esto es lo que se denomina herencia múltiple o derivación múltiple. Por lo tanto, la herencia nos permite encapsular diferentes funcionalidades de un objeto bien real o imaginario, y poder vincular tales funcionalidades a otros objetos más complejos, que heredarán las características del objeto más básico y además añadirán nuevas funcionalidades propias del objeto más complejo.

Antes de introducirnos en los conceptos del lenguaje C++, vamos a mostrar un diagrama de clases de cómo se modela la herencia en tales diagramas. Supongamos un

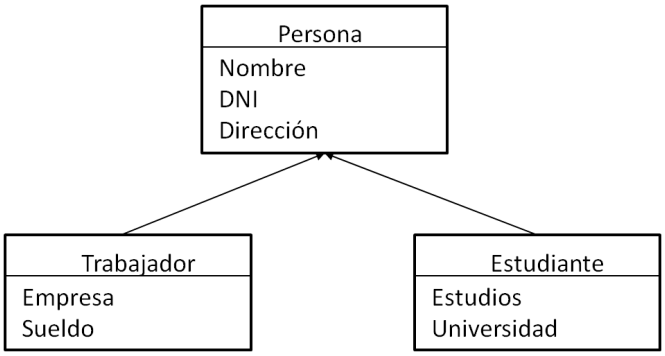

Figura 3.1: Diagrama de clases

ejemplo de una aplicación en la que debemos de registrar personas y las personas serán trabajadores o estudiantes.

El diagrama de clases de este ejemplo sería el mostrado en la Figura 3.1.

Así la clase `Persona` recoge los atributos comunes a todas las personas, pero la clase `Estudiante` además de heredar los atributos de la clase `Persona` define atributos suyos propios, igual que ocurre con la clase `Trabajador`.

3.2 Herencia simple y visibilidad en la herencia

El lenguaje C++ permite herencia simple y herencia múltiple, es decir, permite que una clase herede de una única clase o que una clase herede de más de una clase base. Sin tener en cuenta el tipo de herencia, cuando se crea una nueva clase derivada a partir de una o varias clases bases, C++ permite definir el tipo de visibilidad a aplicar en la herencia. La visibilidad en la herencia significa en qué nivel de visibilidad se "heredan" los atributos y funciones que son públicas o protegidas en la clase base. Antes de entrar en los niveles de visibilidad debemos de aclarar que toda función o atributo que es privado en la clase base, no es visible en la clase derivada. Por lo tanto, si queremos crear una jerarquía de clases y queremos que las clases derivadas hereden ciertos atributos de las clases bases, lo recomendable sería poner esos atributos y/o funciones como protegidas (`protected`).

Una vez aclarado el concepto de los atributos y funciones privados en la clase base, pasamos a ver cómo se crea una clase derivada en el lenguaje C++. La estructura para crear una clase heredada de otra es la siguiente:

```
class <claseDerivada>: [public|private] <claseBase> {...};
```

donde la `claseDerivada` es el nombre de la clase que va a heredar de la `claseBase`. Un aspecto a destacar es si la clase derivada hereda de la clase base de forma pública (`public`) o de forma privada (`private`). Siguiendo el ejemplo, si quisiéramos crear la clase `Trabajador` como clase derivada de `Persona`, tendríamos que declararla de la siguiente forma:

```
class Trabajador: public Persona {
    ...
    ...
};
```

Pero, ¿qué implica heredar de una clase base de forma pública o de forma privada? La clave está en la privacidad que queramos que la clase derivada tenga con respecto a las posibles clases derivadas que pueden surgir de ella. Cuando se realiza una herencia pública, como la que hemos declarado en la clase derivada `Trabajador`, lo que estamos indicando es que todas las funciones y atributos que son públicos en la clase `Persona` pasan a ser también públicos en la clase `Trabajador`. Por lo tanto, desde donde se cree un objeto de tipo `Trabajador` se podrán utilizar las funciones públicas de la clase `Persona`. Por el contrario, si una clase derivada se crea con visibilidad privada de la clase base, esto significa que las funciones y atributos públicos de la clase base pasan a ser privados en la clase derivada, por lo cual estas funciones y atributos no podrán ser utilizados desde fuera de la clase y tampoco serán visibles a una nueva clase derivada que tome como base la clase derivada que hereda de forma privada. En otras palabras, si la clase `Trabajador` hereda de forma privada de la clase `Persona` y creamos una nueva clase derivada llamada `TrabajadorIndefinido` tomando como clase base la clase `Trabajador`, la nueva clase derivada `TrabajadorIndefinido` no tiene acceso a los atributos y funciones públicas de la clase `Persona`. Por lo tanto, antes de decidir si la herencia se realiza con visibilidad pública o privada, habrá que tener en cuenta las cuestiones que se acaban de detallar.

Tras explicar los conceptos de visibilidad, vamos a presentar el ejemplo completo de las clase `Persona`, `Trabajador` y `Estudiante` utilizando la visibilidad pública.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Persona{
6
7  protected:
8    char * direccion;
9    char * nombre;
10   char * dni;
11
12 public:
13   Persona(char * nombre, char * direccion, char * dni);
14   char * getNombre();
15   void getInfo();
16 };
17
18 class Trabajador: public Persona{
19
20 protected:
21   int sueldo;
22   char * empresa;
23
24 public:
25   Trabajador(char * nombre, char * direccion, char * dni, int sueldo, char
         * empresa);
26   int getSueldo();
27   char * getEmpresa();
28 };
29
30 class Estudiante: public Persona{
31
32 protected:
33    char * universidad;
34    char * estudios;
35
36 public:
37    Estudiante(char * nombre, char * direccion, char * dni, char * estudios,
          char * universidad);
38    char * getEstudios();
39 };
40
41 //implementación de las clases
```

```
42
43 Persona::Persona(char * _nombre, char * _direccion, char * _dni){
44     nombre = new char[strlen(_nombre)+1];
45     strcpy(nombre,_nombre);
46     direccion = new char[strlen(_direccion)+1];
47     strcpy(direccion,_direccion);
48     dni = new char[strlen(_dni)+1];
49     strcpy(dni,_dni);
50 }
51
52 char * Persona::getNombre(){return nombre;}
53
54 void Persona::getInfo(){cout << "Nombre: " << nombre << " Direccion: " <<
       direccion << endl;}
55
56 Trabajador::Trabajador(char * _nombre, char * _direccion, char * _dni, int
       _sueldo, char * _empresa): Persona(_nombre,_direccion,_dni){
57     sueldo = _sueldo;
58     empresa = new char[strlen(_empresa)+1];
59     strcpy(empresa,_empresa);
60 }
61
62 int Trabajador::getSueldo(){return sueldo;}
63
64 char * Trabajador::getEmpresa(){return empresa;}
65
66 Estudiante::Estudiante(char * _nombre, char * _direccion, char * _dni, char
       * _estudios, char * _universidad): Persona(_nombre,_direccion,_dni) {
67     estudios = new char[strlen(_estudios)+1];
68     strcpy(estudios,_estudios);
69     universidad = new char[strlen(_universidad)+1];
70     strcpy(universidad,_universidad);
71 }
72
73 char * Estudiante::getEstudios(){return estudios;}
74
75 //Vamos a crear objetos de las diferentes clases y a invocar los métodos.
76
77 int main(){
78
79     Persona maria("Maria","Gran via","85296385G");
80     cout << "----------------" << endl;
81     maria.getInfo();
82     cout << "----------------" << endl;
```

```
83
84      Estudiante pepe("Pepe","Paseo de la catedra","85697412J","Informatica","
            Universidad de Murcia");
85      cout << "---------------" << endl;
86      cout << "Estudios " << pepe.getEstudios() << endl;
87      cout << "---------------" << endl;
88
89      Trabajador josefa("Josefa","Avenida del rio","65473251U",2500,"Oracle
            Solutions");
90      cout << "---------------" << endl;
91      josefa.getInfo();
92      cout << "Sueldo " << josefa.getSueldo() << endl;
93      cout << "---------------" << endl;
94
95      system("pause");
96
97      return 0;
98  }
```

Como vemos en el ejemplo, las clases derivadas de la clase `Persona` pueden utilizar los atributos definidos como protegidos y las funciones definidas como públicas. Si los atributos hubieran sido definidos como privados no podrían ser utilizados. Por otro lado, los constructores de las clases derivadas deben de llamar al constructor de la clase base cuando sea necesario para iniciar los valores correspondientes. Esto en el ejemplo se puede apreciar en las clases `Estudiante` y `Trabajador`:

```
1 Estudiante(char * _nombre, char * _direccion, char * _dni, char * _estudios
      , char * universidad) : Persona(_nombre,_direccion,_dni){...}
```

Sin embargo, en el ejemplo anterior, las clases derivadas no han redefinido ningún método de la clase base. Por lo tanto, cuando un objeto de la clase derivada invoca a un método de la clase base, la funcionalidad que aplica es la de la clase base, pero ¿qué ocurre si queremos redefinir un método de la clase base en una clase derivada? Para poder redefinir métodos en las clases derivadas, la clase base debe indicar qué métodos permite redefinir en las clases derivadas declarándolos como métodos virtuales. Así, un método en la clase derivada que tenga la misma signatura que un método virtual de la clase base significa que ese método está redefinido. Es importante tener en cuenta que en el fichero de cabecera (.h) de la clase derivada se deben de incluir los métodos que

se redefinen de la clase base. Hay que tener en cuenta que una vez que un método es declarado como virtual, permanece como virtual en las clases derivadas. Para invocar las diferentes versiones de los métodos se debe de utilizar la clasificación de rutinas, es decir, especificar a qué clase se está haciendo referencia de la siguiente forma:

```
1 nombreClase::nombreMetodo
```

Así, si quisiéramos redefinir el método `getInfo()` de la clase `Persona` en sus clases derivadas, deberíamos de declarar el método `getInfo()` como `virtual` y luego redefinirlo. También hay que comentar que los métodos virtuales nos permiten utilizar el polimorfismo en el lenguaje C++, puesto que mediante estos métodos podremos aplicar la ligadura dinámica. Vamos a ver un ejemplo de cómo funciona el polimorfismo y las funciones virtuales.

```
1 #include <iostream>
2
3 using namespace std;
4
5 class Persona{
6
7 protected:
8   char * direccion;
9   char * nombre;
10  char * dni;
11
12 public:
13  Persona(char * nombre, char * direccion, char * dni);
14  char * getNombre();
15  virtual void getInfo();
16 };
17
18 class Trabajador: public Persona{
19
20 protected:
21  int sueldo;
22  char * empresa;
23
24 public:
25  Trabajador(char * nombre, char * direccion, char * dni, int sueldo, char
      * empresa);
```

```
26    int getSueldo();
27    char * getEmpresa();
28    //sobrescribiendo el método en la cabecera
29    void getInfo();
30    char * getNombre();
31  };
32
33  class Estudiante: private Persona{
34
35  protected:
36     char * universidad;
37     char * estudios;
38
39  public:
40     Estudiante(char * nombre, char * direccion, char * dni, char * estudios,
           char * universidad);
41     char * getEstudios();
42     //sobrescribiendo el método en la cabecera
43     void getInfo();
44     char * getNombre();
45  };
46
47  //implementación de las clases
48
49  Persona::Persona(char * _nombre, char * _direccion, char * _dni){
50       nombre = new char[strlen(_nombre)+1];
51       strcpy(nombre,_nombre);
52       direccion = new char[strlen(_direccion)+1];
53       strcpy(direccion,_direccion);
54       dni = new char[strlen(_dni)+1];
55       strcpy(dni,_dni);
56  }
57
58  char * Persona::getNombre(){return nombre;}
59
60  void Persona::getInfo(){cout << "Nombre: " << nombre << " Direccion: " <<
       direccion << endl;}
61
62  Trabajador::Trabajador(char * _nombre, char * _direccion, char * _dni, int
       _sueldo, char * _empresa): Persona(_nombre,_direccion,_dni){
63    sueldo = _sueldo;
64    empresa = new char[strlen(_empresa)+1];
65    strcpy(empresa,_empresa);
66  }
```

```
67
68 int Trabajador::getSueldo(){return sueldo;}
69
70 char * Trabajador::getEmpresa(){return empresa;}
71
72 void Trabajador::getInfo(){
73   //llamando al método de la clase base
74   Persona::getInfo();
75   cout << "Sueldo " << sueldo << " Empresa "<< empresa << endl;
76 }
77
78 char * Trabajador::getNombre(){
79   cout << "Soy un trabajador y mi nombre es: " <<  endl;
80   return Persona::nombre;
81 }
82
83 Estudiante::Estudiante(char * _nombre, char * _direccion, char * _dni, char
       * _estudios, char * _universidad): Persona(_nombre,_direccion,_dni) {
84     estudios = new char[strlen(_estudios)+1];
85     strcpy(estudios,_estudios);
86     universidad = new char[strlen(_universidad)+1];
87     strcpy(universidad,_universidad);
88 }
89
90 char * Estudiante::getEstudios(){return estudios;}
91
92 void Estudiante::getInfo(){
93   //llamando al método de la clase base
94   Persona::getInfo();
95   cout << "Estudios " << estudios << " Universidad " << universidad <<endl;
96 }
97
98 char * Estudiante::getNombre(){
99   cout << "Soy un estudiante y mi nombre es: " <<  endl;
100  return Persona::nombre;
101 }
102
103 //Vamos a crear objetos de las diferentes clases y a invocar los métodos.
104
105 int main(){
106
107   Persona * maria = new Estudiante("Maria","Gran via","85296385G","
        Educacion","Universidad de Valencia");
108   cout << "----------------" << endl;
```

```
109   maria->getInfo();
110   cout << "Nombre " << maria->getNombre() << endl;
111   cout << "---------------" << endl;
112
113   Persona * josefa = new Trabajador("Josefa","Avenida del rio","65473251U"
          ,2500,"Oracle Solutions");
114   josefa->getInfo();
115   cout << "Nombre " << josefa->getNombre() << endl;
116   cout << "---------------" << endl;
117
118   system("pause");
119
120   return 0;
121 }
```

Si nos fijamos en el ejemplo, tenemos que la clase `Estudiante` y `Trabajador` han redefinido el método `getInfo()` y dicho método está declarado como `virtual` en la clase base `Persona`. Sin embargo, la clase `Trabajador` y `Estudiante` han declarado un método llamado `getNombre()`. Este método también está definido en la clase base, pero no está definido como `virtual`. Vamos a analizar la salida del programa y ver cuándo se ha aplicado la ligadura dinámica y cuándo no. La salida del programa es la siguiente:

```
---------
Nombre: Maria Direccion: Gran via
Estudios: Educacion Universidad: Universidad de Valencia
Maria
---------
Nombre: Josefa Direccion: Avenida del rio
Sueldo: 2500 Empresa: Oracle Solutions
Josefa
---------
```

Al analizar la salida podemos ver que cuando se ha llamado al método `getInfo()` sí que se han ejecutado los métodos redefinidos en las clases derivadas, sin embargo, se ha ejecutado el método `getNombre()` de la clase base `Persona`, no de las clases derivadas. Esto es así porque el método `getNombre()` no está definido como `virtual` de ahí que no se haya aplicado la ligadura dinámica. Un último aspecto a comentar es el uso de la

función `dynamic_cast` que se utiliza para comparar el tipo de un objeto (igual que la función `instanceof` del lenguaje Java). Esta función sólo es aplicable a tipos puntero. Veamos un ejemplo sencillo:

```
1  int main(){
2
3    int nInvitados=3;
4    Persona * misInvitados[nInvitados];
5
6    misInvitados[0] = new Estudiante("Maria","Gran via","85296385G","
         Educacion","Universidad de Valencia");
7
8    misInvitados[1] = new Trabajador("Josefa","Avenida del rio","65473251U"
         ,2500,"Oracle Solutions");
9
10   misInvitados[2] = new Trabajador("Gregoria","Avenida las Americas","
         32589854U",1500, "IMB Solutions");
11
12   int nEstudiante = 0;
13   for(int i = 0; i < nInvitados; i++){
14     // Si es estudiante, añado un estudiante invitado.
15     if(dynamic_cast<Estudiante *>(misInvitados[i])){
16       nEstudiante++;
17     }
18   }
19
20   cout << "Numero de estudiantes = " << nEstudiante << endl;
21   cout << "Numero de trabajadores = " << nInvitados-nEstudiante << endl;
22
23   system("pause");
24
25     return 0;
26 }
```

3.3 Herencia múltiple

Una de las características que ofrece el lenguaje C++ y que no ofrecen de forma directa otros lenguajes como Java, es que permite la herencia múltiple. El concepto de herencia múltiple consiste en crear una clase derivada heredando de una o más clases base. Un problema bastante común cuando se hereda de dos clases bases es que exista cierta

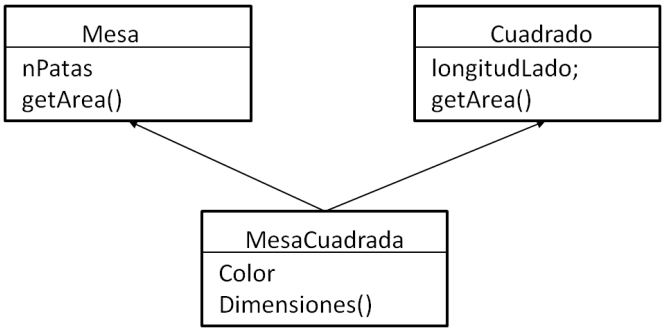

Figura 3.2: Diagrama de clases de herencia múltiple

ambigüedad, por ejemplo en dos atributos que se llamen igual o dos métodos con el mismo nombre. Para solucionar tal ambigüedad se aplica el operador de ámbito `::` indicando de qué clase base se desea ejecutar el método. Un ejemplo de herencia múltiple podría darse en el ejemplo de la Figura 3.2.

En este ejemplo tenemos una clase `MesaCuadrada` que hereda a la vez de la clase `Mesa` y de la clase `Cuadrado`. El problema comentado ocurre cuando la clase `MesaCuadrada` necesita llamar al método `getArea()`. En este caso y como hemos comentado se debe de especificar la clase de la que se quiere ejecutar el método. El ejemplo presentado en el diagrama quedaría implementado de la siguiente forma:

```
1  #include<iostream>
2
3  using namespace std;
4
5  class Mesa{
6
7  protected:
8    int nPatas;
9    int area;
10
11 public:
12   Mesa(int _nPatas, int _area){nPatas = _nPatas; area = _area;};
13   int getArea(){return area;};
14 };
15
16 class Cuadrado{
17
```

```
18 protected:
19   int longitudLado;
20
21 public:
22   Cuadrado(int _longitud){longitudLado = _longitud;};
23   int getArea(){return longitudLado*longitudLado;};
24 };
25
26 class MesaCuadrada: public Mesa, public Cuadrado{
27
28 protected:
29    char color[20];
30
31 public:
32    MesaCuadrada(char _color[20], int _longitud, int _nPatas);
33 };
34
35 //implementación del constructor de MesaCuadrada
36
37 MesaCuadrada::MesaCuadrada(char _color[20], int _longitud, int _nPatas):
       Mesa(_nPatas,_longitud*_longitud), Cuadrado(_longitud){
38    strcpy(color,_color);
39 }
40
41 int main(){
42
43    MesaCuadrada mimesa("Rojo",5,4);
44
45    cout << "El area del tablero de la mesa es " << mimesa.Cuadrado::getArea
          () << endl;
46
47    system("pause");
48
49    return 0;
50 }
```

Como hemos podido ver en el ejemplo, para definir herencia múltiple lo único que hay que indicar al declarar la nueva clase derivada son las clases de las que hereda.

```
1 class MesaCuadrada: public Mesa, public Cuadrado{...}
```

Además en el constructor de `MesaCuadrada` vemos cómo se llama a los constructores

de las clases de las que se hereda:

```
MesaCuadrada(char _color[20], int _longitud, int _nPatas): Mesa(_nPatas,
    _longitud*_longitud), Cuadrado(_longitud){strcpy(color,_color);}
```

Por último, para evitar la ambigüedad del método `getArea()`, se utiliza el operador de ámbito para aclarar que queremos utilizar el método de la clase `Cuadrado`. Otra posible solución podría haber sido redefinir en la clase `MesaCuadrada` el método `getArea()` especificando el método que se desea utilizar. El método `getArea()` redefinido quedaría de la siguiente forma:

```
// Redefinición del método getArea()

int MesaCuadrada::getArea(){
    return Cuadrado::getArea();
}

int main(){

    MesaCuadrada mimesa("Rojo",5,4);

    cout << "El area del tablero de la mesa es " << mimesa.getArea() <<endl;

    system("pause");

    return 0;
}
```

Al redefinir el método `getArea()` en la clase `MesaCuadrada`, cuando llamamos al método siempre se llama al de la clase `Cuadrado`.

Un último aspecto a comentar es la ambigüedad y duplicidad que se puede producir en la herencia múltiple si dos clases heredan de una clase común y después otra clase hereda de esas dos clases. Un ejemplo sería el diagrama de la Figura 3.3.

Como vemos en el ejemplo, la clase `Becario` hereda de la clase `Trabajador` y `Estudiante`, pero a su vez estas clases heredan de la clase `Persona` por lo que se podría interpretar que `Becario` hereda dos veces de la clase `Persona`. Para evitar estas ambigüedades, existe la herencia mediante clases virtuales. La herencia mediante clases virtuales tiene como función eliminar esa posible duplicidad de heredar dos veces de una

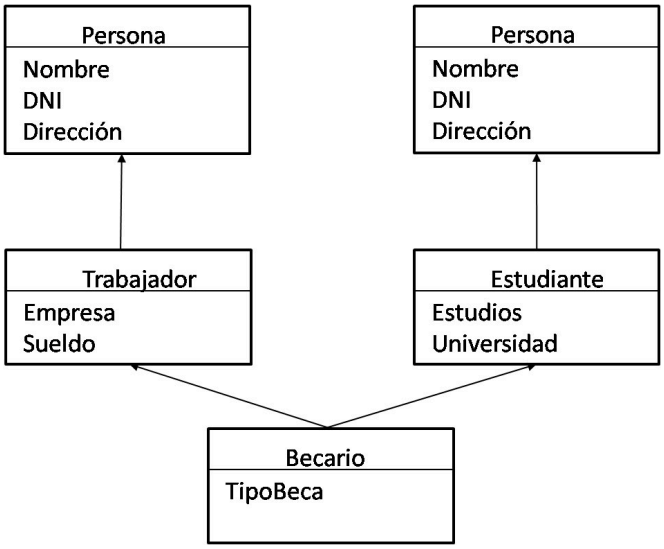

Figura 3.3: Diagrama de clases

misma clase. La única modificación en el código es al declarar la herencia en las clases `Trabajador` y `Estudiante` que heredan de la clase `virtual Persona`. La declaración de la herencia en este caso sería como sigue:

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Persona{
6
7  protected:
8    char * direccion;
9    char * nombre;
10   char * dni;
11
12 public:
13   Persona(char * nombre, char * direccion, char * dni);
14   char * getNombre();
15   void getInfo();
16 };
17
18 class Trabajador: public virtual Persona{
19
```

```
20 protected:
21   int sueldo;
22   char * empresa;
23
24 public:
25   Trabajador(char * nombre, char * direccion, char * dni, int sueldo, char
         * empresa);
26   int getSueldo();
27   char * getEmpresa();
28 };
29
30 class Estudiante : public virtual Persona{
31
32 protected:
33    char * universidad;
34    char * estudios;
35
36 public:
37    Estudiante(char * nombre, char * direccion, char * dni, char * estudios,
          char * universidad);
38    char * getEstudios();
39 };
40
41 //implementación de las clases
42
43 Persona::Persona(char * _nombre, char * _direccion, char * _dni){
44     nombre = new char[strlen(_nombre)+1];
45     strcpy(nombre,_nombre);
46     direccion = new char[strlen(_direccion)+1];
47     strcpy(direccion,_direccion);
48     dni = new char[strlen(_dni)+1];
49     strcpy(dni,_dni);
50 }
51
52 char * Persona::getNombre(){return nombre;}
53
54 void Persona::getInfo(){cout << "Nombre: " << nombre << " Direccion: " <<
     direccion << endl;}
55
56 Trabajador::Trabajador(char * _nombre, char * _direccion, char * _dni, int
     _sueldo, char * _empresa): Persona(_nombre,_direccion,_dni){
57   sueldo = _sueldo;
58   empresa = new char[strlen(_empresa)+1];
59   strcpy(empresa,_empresa);
```

```
60 }
61
62 int Trabajador::getSueldo(){return sueldo;}
63
64 char * Trabajador::getEmpresa(){return empresa;}
65
66 Estudiante::Estudiante(char * _nombre, char * _direccion, char * _dni, char
       * _estudios, char * _universidad): Persona(_nombre,_direccion,_dni) {
67     estudios = new char[strlen(_estudios)+1];
68     strcpy(estudios,_estudios);
69     universidad = new char[strlen(_universidad)+1];
70     strcpy(universidad,_universidad);
71 }
72
73 char * Estudiante::getEstudios(){return estudios;}
74
75 class Becario: public Estudiante, public Trabajador{
76
77 protected:
78    char tipoBeca[50];
79
80 public:
81    Becario(char * _nombre, char * _direccion, char * _dni, char * _estudios
          , char * _universidad, int _sueldo, char * _empresa, char _tipoBeca
          [50]);
82    char getEstudios();
83 };
84
85 //Implementación de Becario
86
87 Becario::Becario(char * _nombre, char * _direccion, char * _dni, char *
      _estudios, char * _universidad, int _sueldo, char * _empresa, char
      _tipoBeca[50]) :
88 Estudiante(_nombre,_direccion,_dni,_estudios,_universidad), Trabajador(
      _nombre,_direccion,_dni,_sueldo,_empresa),Persona(_nombre,_direccion,
      _dni){
89   strcpy(tipoBeca,_tipoBeca);
90 }
91
92 int main(){
93
94   Becario pepe("Maria","Gran via","85296385G","Educacion","Universidad de
        Valencia",2500,"Oracle Solutions","Tipo 1");
95   pepe.getInfo();
```

```
96    cout << "Nombre " << pepe.getNombre() << endl;
97    cout << "---------------" << endl;
98
99    system("pause");
100
101   return 0;
102 }
```

Como podemos ver en el constructor de `Becario` hemos utilizado los constructores de `Persona`, `Trabajador` y `Estudiante` y además hemos modificado la forma de heredar de la clase `Trabajador` y de la clase `Estudiante`, puesto que para evitar ambigüedades se ha realizado la herencia colocando delante de la clase `Persona` la palabra `virtual`. Con esto lo que se consigue es que la clase `Becario` herede una sola vez de la clase `Persona`.