Lenguaje de Programación C++

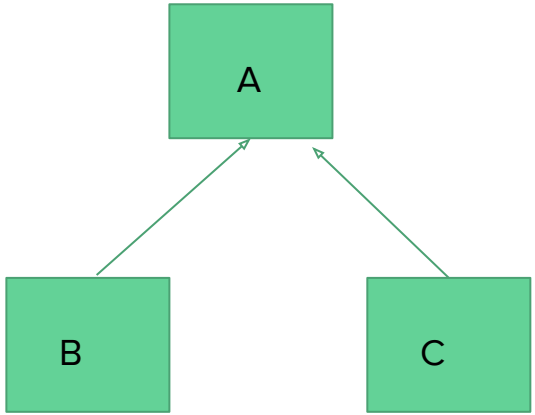
Herencia y polimorfismo

Herencia

Dentro del marco teórico, verán el concepto de lo que es la generalización y la especificación de los objetos. En una visión de conjuntos, podemos decir que un conjunto contiene a otros o al revés, un conjunto está contenido. En nuestro ámbito, podemos decir que somos un conjunto de personas, y de ellas distinguimos dos tipos, alumnos y docentes. En la vista general, todos somos personas (generalización) si especificamos, vemos características distintivas, algunos son alumnos, otros son docentes. Todos tienen una base común que es la de Persona, luego, se extiende esta clase a las particulares.

A esto le llamamos herencia. La clase Docente y Alumno heredan las caracteristicas de la Persona.

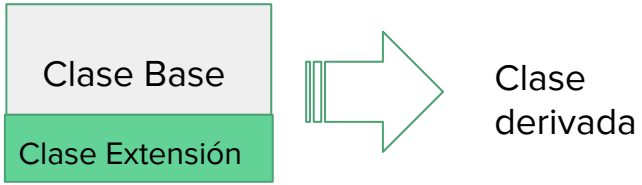
Herencia



Es importante que la herencia indica un tipo de relación especial entre clases/objetos.

Los Objetos B y C heredan de A, por lo tanto B y C ***son*** de tipo A.

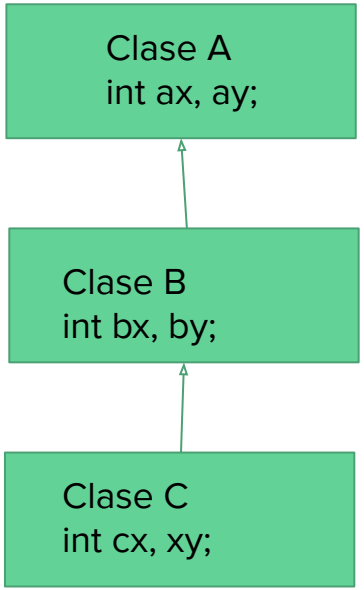
Si tenemos una visión “lógica” sería esto:



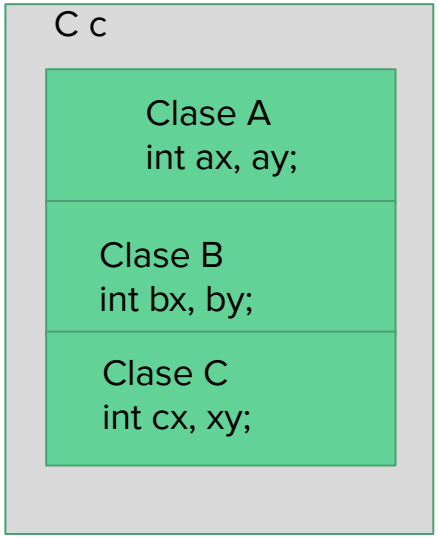
Son varias las formas de llamar a las clases: Padre - Hija / Clase Base - Extendida / Superior - Inferior

Herencia

Siguiendo con otro ejemplo:



Si Instanciamos a C tendrimos:



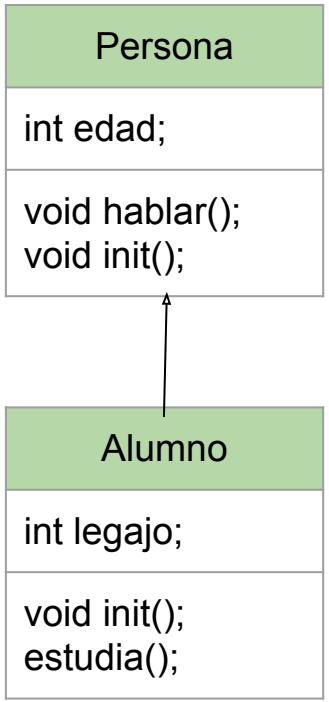
El objeto obtenido posee todos los atributos y métodos propios y de sus clases padres

Ejemplo

```
class Persona{
protected:
    int edad;
public:
    Persona(){ … }
    void habla();
    void init();
};

class Alumno{
    int legajo;
public:
    Alumno(){ … }
    void init(); // sobrecargado
    void estudia();
};
```

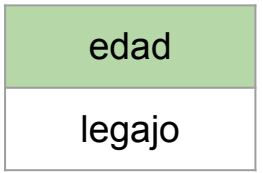
Representación de clases



Representación de objetos

Persona P

Alumno A



A.init/(;
A.habla();
A.estudia();

Acceso a los datos dentro de la Herencia

- Todos los atributos de la clase base se encuentran en la clase derivada aunque estos no estén accesibles (y por lo tanto se tienen en cuenta en el tamaño)
- Si el nombre de un atributo de la clase derivada es igual al de la clase base, a este último se lo debe invocar a través del nombre completo: clase_base::dato
- Todo miembro privado de la clase base no podrá ser accedido en forma directa por su derivada, se deberá tener que usar métodos públicos o protegidos.
    - Un buen diseño debe tener en cuenta esto y solo debe haber atributos comunes a las clases derivadas y estos estar protegidos y no privados

Acceso a los datos...

```
class A{
  int x;
public:
  int y;
  void setX(int i){x = i;}

};
class B:public A{
    int y;
 public:
    void f(){
        y = 5; // atributo local
        A::y = 4; // atributo de clase base
        x = 9; // ERROR!!!
        setX(9); // Bien!
    }
};
```

Si compilamos....

```
$ g++ -Wall -oherencia1 herencia.cpp
herencia.cpp: En la función miembro ‘void B::f()’:
herencia.cpp:20:9: error: ‘int A::x’ is private within this context
   20 |        x = 9; // ERROR!!!
      |        ^
herencia.cpp:5:7: nota: declared private here
    5 |   int x;
      |       ^
$
```

Constructores y destructores

- El constructor y el destructor no se hereda. Cada clase lo mantiene en forma independiente.
- Cuando hay una cadena de herencias se ejecutan en orden:
  - El constructor se llama en forma descendente desde la clase base a sus derivadas
  - El destructor de forma ascendente desde la derivada inferior hasta la clase base superior
- La inicialización se realiza en la clase derivada de la siguiente forma:

```
class A{
  int x;
public:
  A(int ax){x=ax;}
  void setX(int i){x = i;}
};
```

```
class B:public A{
    int y;
 public:
    void B(int bx, int by): A(bx){
        y = by;
    }
};
```

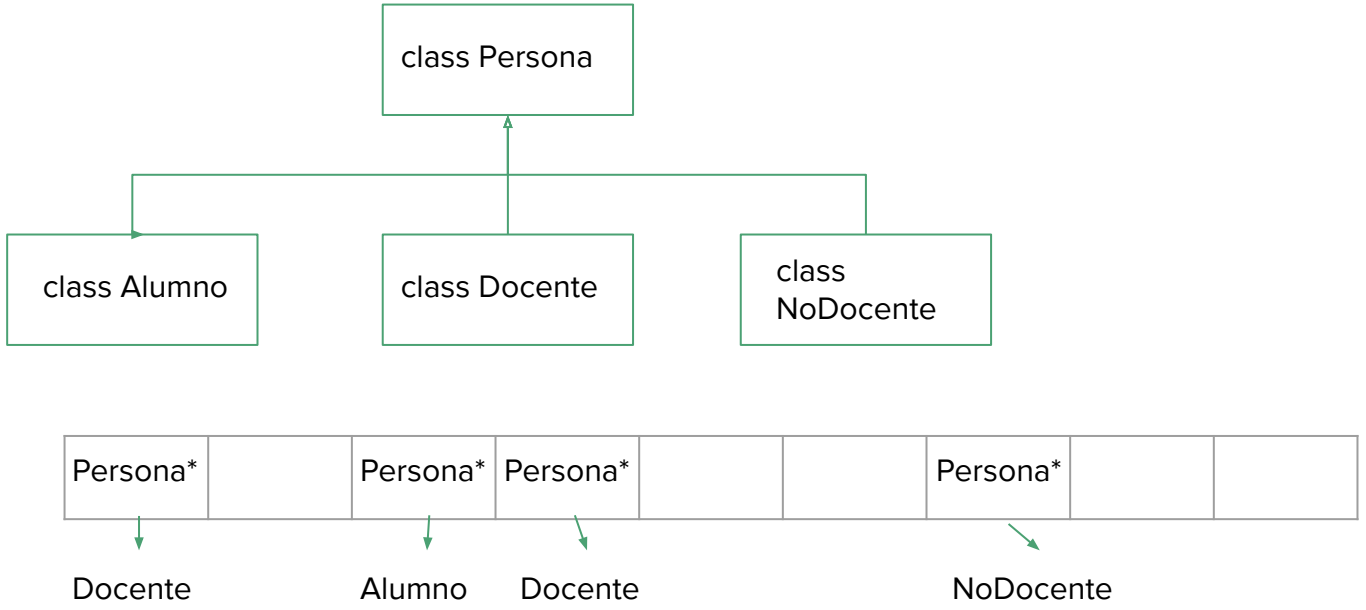
invoca al constructor superior

Lenguaje de Programación C++

Polimorfismo

Introducción

- Expresamos que una clase B que hereda de A, es de tipo A. Por lo tanto podemos considerar un contenedor del siguiente tipo:



Introducción...

Un puntero de la clase Base puede señalar a un objeto de la clase derivada. Esto no implica una pérdida de información.

```
class Persona{
  int edad;
public:
  void hablar();
  void caminar();
};
class Estudiante:public Persona{
  int legajo;
public:
  void hablar();
  void estudiar();
}
```

```
int main(){
  Estudiante e;
  e.hablar(); // desde Estudiante
  e.estudiar(); // desde Estudiante
  e.caminar(); // desde Persona

  Persona* p = new Estudiante();
  p->hablar(); // desde Persona
  p->caminar(); // desde Persona
  p->estudiar(); // ERROR
}
```

Typing y Binding estático y dinámico

- ***Typing***: Identifica el tipo de objeto que estamos usando.
- ***Binding***: la asociación de direcciones de memoria de funciones/variables.
  - Typing estático: en tiempo de compilación (como en C)
  - Typing dinámico: en tiempo de ejecución (como en Java)
- En C++ el ***binding*** es estático excepto que se indique lo contrario.
  - Usando la palabra clave ***virtual***

Veamos con el ejemplo anterior como funcionaria:

Typing y Binding

```
class Persona{
  int edad;
public:
  virtual void hablar();
  void caminar();
};

class Estudiante:public Persona{
  int legajo;
public:
  void hablar();
  void estudiar();
}
```

```
int main(){
  Estudiante e;
  e.hablar(); // desde Estudiante
  e.estudiar(); // desde Estudiante
  e.caminar(); // desde Persona

  Persona* p = new Estudiante();
  p->hablar(); // desde Estudiante
  p->caminar(); // desde Persona
  p->estudiar(); // ERROR
}
```

C++ enlaza automáticamente con el método *hablar* del estudiante por estar implementado (considere que hablar en Persona como un método abstracto)

Contenedores

Teniendo en cuenta el gráfico de la transparencia 10,más lo visto del polimorfismo de clases. Veamos un breve ejemplo:

```
class Persona{
  int edad;
public:
  virtual void hablar();
  void caminar();
};
class Estudiante:public Persona{
  int legajo;
public:
  virtual void hablar();
  void estudiar();
};
```

```
class Docente:public Persona{
  int cuil;
public:
  virtual void hablar();
  void catedra();
};

int main(){
  Persona** p= new Persona*[2];
  p[0]= new Estudiante();
  p[1]= new Docente();

  for(int i=0; i<2; i++)
    p[i]->hablar();

  return 0;
}
```

Clases Abstractas

No siempre en nuestro diseño es conveniente que se pueda instanciar un tipo de objeto, principalmente si está diseñado para que herede: Ej: Persona.

Del mismo modo ocurre con los métodos, en general, si el método es virtual no necesito implementarlo en la clase base, sino en las que lo heredan por sus particularidades. Para solucionar esto, definimos al método como "***puro***" y por lo tanto la clase se convierte en ***abstracta*** y no es posible instanciarla ni pasarla por por valor.

Para esto, definimos a un método de la clase de la siguiente manera:

virtual retorno nombre_función (param1, ...) = 0;

Reconociendo tipos

Como hemos visto, cuando un contenedor es de una clase base, al guardar los objetos de ese tipo, estos hacen un ***up cast***... El objeto no pierde sus características particulares pero no podrá acceder a ellas hasta que no se haga un ***down casting***.

¿Cómo identifico que tipo de objeto era? Podemos utilizar

typeid();

Veamos un ejemplo:

typeid()

```
class Persona{
  int edad;
public:
  virtual void hablar();
  void caminar();
};
class Estudiante:public Persona{
  int legajo;
public:
  virtual void hablar();
  void estudiar();
};
class Docente:public Persona{
  int cuil;
public:
  virtual void hablar();
  void catedra();
};
```

```
int main(){
  Persona* p= new Docente();

  cout<< typeid(p).name()<<endl;// Persona
  cout<< typeid(*p).name()<<endl;// Docente
  cout<< (typeid(p) == typeid(Docente*))
    << endl; // Falso
  cout<< (typeid(p) == typeid(Persona*))
    << endl; // Verdadero

  if (typeid(*p) == typeid(Docente)){
    Docente* d = (Docente*)p;
    d->catedra();
  }
  return 0;
}
```