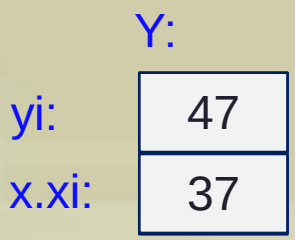
Que vimos? Reutilización de código

- Mejorar el copy/paste/change.
- Creación de nuevas clases en base a las ya existentes, sin necesidad de modificar su código.
  - Composición: crear nuevas clases compuestas por objetos de clases ya existente.
  - Herencia: crear nuevas clases del tipo de una clase ya existente, sin modificarla.

- La herencia es uno de los aspectos más importantes de la programación orientada a objetos.

Composición

```
class X {
      int xi;
   public:
      X() { xi = 0; }
      void set(int ii) { xi = ii; }
      int read() const { return xi; }
};
class Y {
      int yi;
  public:
      X x;
      Y() { yi = 0; }
      void f(int ii) { yi = ii; }
      int g() const { return yi; }

};
```




```
int main() {
     Y y;
     y.f(47);
     y.x.set(37);
     y.x.read();
}
```

Orden de construcción y destrucción

- Orden de construcción
  - Invocación de constructores de clases base
  - Invocación de constructores de miembros
  - Ejecución del cuerpo del constructor

- Orden de destrucción:
  - Ejecución del cuerpo del destructor
  - Invocación de destructores de miembros
  - Invocación de destructores de clases base

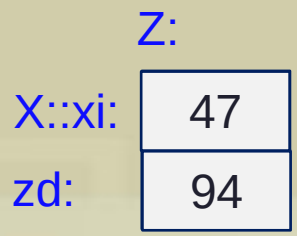
Construcción de objetos miembro

```
class Club {
    string name;
    vector<string> members;
    vector<string> officers;
    Date founded;
    // ...
    Club(const string& n, Date fd);
};

Club::Club(const string &n, Date fd)
    : name{n}, member{}, officers{}, founded{fd}
{
    // ...
}
```

- Mecanismo de member initialization list. Mismo mecanismo que para constantes o referencias.
- Se realiza en el orden de declaración de los miembros. Ojo!

Herencia

```
class X {
        int xi;
    public:
        X() : xi{0} { }
        void set(int ii) { xi = ii; }
        int read() const { return xi; }
};
class Z : public X {
        double zd;
    public:
        Z() : zd{0} { }
        void set(int ii) {
            zd = 2.0 * ii;
            X::set(ii);
        }
};
```




```
int main() {
    Z z;
    z.set(47);
    z.read();
}
```

Herencia

- **`Z`** hereda de **`X`**. **`X`** es una clase base de **`Z`**. **`Z`** es una clase derivada de **`X`**
  - **`Z`** contiene todos los atributos de **`X`**.
  - **`Z`** contiene todos los métodos de **`X`**.
  - Todos los elementos privados de **`X`** son privados de **`Z`**.
  - La herencia se especificó como **`public`**, por lo que todos los miembros públicos de **`X`** son accesibles como públicos a través de **`Z`**.
  - El método **`set`** está redefinido en **`Z`**.
  - El método **`read`** no está redefinido en **`Z`**, se activa el de **`X`** cuando se llama a través de un objeto de clase la clase **`Z`**.

Orden de construcción y destrucción

- Orden de construcción
  - Invocación de constructores de clases base
  - Invocación de constructores de miembros
  - Ejecución del cuerpo del constructor

- Orden de destrucción:
  - Ejecución del cuerpo del destructor
  - Invocación de destructores de miembros
  - Invocación de destructores de clases base

Lista de inicialización en el constructor

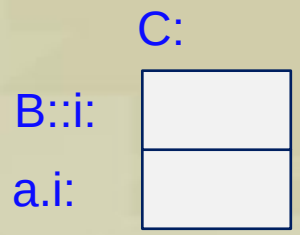
- El compilador garantiza la ejecución de los constructores.
- El constructor de la clase base se ejecuta antes que el de la clase derivada.
- Para mandarle argumentos al constructor de una clase base se hace en la lista de inicialización.

```
class X {
        int i;
    public:
        X(int ii) { i = ii; }
};

class Z : public X {
    public:
        Z(int ii) : X(ii) { }
};
```

```
class Y {
        X myx;
    public:
        Y(int ii) : myx(ii) { }
};
```

Herencia y Composición

```
class A {
        int i;
    public:
        A(int ii) : i(ii) { }
        void f() const { }
};


class B {
        int i;
    public:
        B(int ii) : i(ii) { }
        void f() const { }
};
```




```
class C : public B {
        A a;
    public:
        C(int ii) : B(ii), a(ii) { }
        void f() {
           a.f();
           B::f();
        }
};
```

Métodos que no se heredan

- Constructores.
- Destructores.
- `operator=()`.

Composición vs. herencia

- Composición se utiliza cuando un objeto está compuesto o contiene otros, pero no se quiere sus interfaces. Típicamente los objetos embebidos son privados.
  - Ejemplo: un auto tiene motor, ruedas, puertas, etc. en este caso se puede pensar que sean objetos privados o no.

- Se utiliza herencia cuando el objeto derivado cumple la relación de *“es un”* con el objeto padre. El objeto derivado hereda la interface del padre.
  - Ejemplo: un auto es un vehículo.

Upcasting

- Cuando una clase hereda de otra, cumple una relación de *“es un”*.
- La interface de la clase base está presente en la clase derivada.
- El compilador acepta un puntero o una referencia a una clase derivada cuando necesita una referencia o un puntero a la clase base.
- Toda la funcionalidad que alguien puede esperar a través de un puntero o referencia de un objeto de una clase base está presente en un puntero o referencia a un objeto de la clase derivada.

Upcasting

```
enum note { Do, Re, Mi };

class Instrument {
  public:
    void play(note n) const { }
};

class Wind : public Instrument {
    /* ... */
};

void tune(Instrument &i) {
    // ...
    i.play(Do);
}
```

```
int main() {
    Wind flute;
    tune(flute);
    return 0;
}
```

Upcasting

- Objetos pueden ser tratados como de su tipo o de un tipo base.

```
enum note { Do, Re, Mi}; // Etc.

class Instrument {
    public:
        void play(note n) const { cout << "Instrument::play " << n << endl;}
};
// Wind objects are Instruments
class Wind : public Instrument {
    public:
        // Redefine interface function:
        void play(note n) const { cout << "Wind::play " << n << endl;}
};

void tune(Instrument& i) { i.play(Re); }

int main() {
  Wind flute;
  tune(flute); // Upcasting
}
```

Ligado (binding)

- La conexión entre una llamada a una función y el cuerpo de la función se denomina ligado. Cuando esta conexión se realiza antes de que el programa sea corrido (realizado por el compilador/linker) se denomina “early binding”.

- La solución al problema anterior es el “late binding”, que significa que el ligado ocurre en tiempo de ejecución. Cuando un lenguaje implementa esto, debe tener algún mecanismo para determinar el tipo del objeto a tiempo de ejecución y llamar a la función miembro adecuada.

Métodos Virtuales

- Para causar que ocurra “late binding” se requiere el uso de keyword **`virtual`**.

```
class Instrument {
    public:
        virtual void play(note n) const {
            cout << "Instrument::play " << n << endl;
        }
};
```

Métodos Virtuales

Para causar que ocurra “late binding” se requiere el uso de keyword **`virtual`**.

```
class Instrument {
public:
        virtual void play(note n) const {
            cout << "Instrument::play " << n << endl;
        }

        virtual const char* what() const {
            return "Instrument";
        }

        // Assume this will modify the object:
        virtual void adjust(int v) { }
};
```

Ejemplo

```
void tune(Instrument& i) {
    // ...
    i.play(Re);
}

void f(Instrument& i) { i.adjust(1); }

// Pointer Upcasting during array initialization:
Instrument* A[] = {
    new Wind,           // Wind*
    new Percussion,     // Percussion*
    new Woodwind,       // Woodwind*
    new Brass           // Brass*
};
```

Métodos Virtuales Puros

- Declarar métodos virtuales en una clase base es una manera de asegurar que todas las clases derivadas en una jerarquía tengan una interface común.
- Cuando en la clase base no tengo información para realmente implementar el método, puedo declararlo y dejarlo sin definir.

```
class Instrument {
    public:
        virtual void play(note n) const = 0;
};
```

- Se llaman métodos virtuales puros.
- Para poder instanciar un objeto de una clase, esta no puede tener métodos virtuales puros. Es decir, las clases derivadas están obligadas en algún lugar de la jerarquía a definir el método virtual puro.

Ejemplo

```
class Container {
    public:
        virtual double& operator[](int) = 0;
        virtual int size() const = 0;
        virtual ~Container() { }
};

void use(Container &c) {
    const int sz = c.size();
    for( int i = 0; i != sz; ++i )
        cout << c[i] << endl;
}
```

- La definición de la interface **`Container`** ya me habilita a utilizarla.
- Sin embargo no puedo instanciar objetos de tipo **`Container`**.
- Alguna clase derivada ***debe*** definir estos métodos virtuales puros.

Ejemplo

```
Container cont;   // ERROR de compilación
```

- Error de compilación debido a que **`Container`** tiene métodos virtuales puros. Las clases con algún método virtual puro se llaman clases abstractas. No son instanciables.

```
class Vector : public Container {
    public:
        // ...
        double& operator[](int)  { /*...*/ }
        int size() const         { return sz; }
};

Vector vect(5);
use(vect);
```

- Ahora sí.

Herencia – Nuevo nivel de protección

- A los niveles de protección **`public`** y **`private`** para los miembros de una clase se le agrega uno nuevo: **`protected`**.
- Los miembros de una clase declarados en una sección **`protected`**, se comportan como **`public`** para clases derivadas y como **`private`** para cualquier otro uso.

```
class Base {
    protected:
        int i;
};
class Der : public Base {
    public:
        void g(int ii) {
            i = ii;
        }
};
```

```
void f() {
    Base b;
    b.i = 0;  // Error

    Der d;
    d.g(3);
}
```