# Polimorfismo

Instituto Balseiro

# Polimorfismo y funciones virtuales

- Polimorfismo (implementado en C++ con funciones virtuales) es la tercera característica importante de un lenguaje orientado a objetos (después de abstracción y herencia).
- Provee otra separación entre interface e implementación (desacople del “que” del “como”).
- Encapsulación crea nuevos tipos combinando características.
- Control de acceso separa la interface de la implementación haciendo los detalles privados.
- Las funciones virtuales manejan el desacople en términos de tipos.

# Evolución de un programador en C++

- Programadores C adquieren C++ en cuatro etapas:
  - Un “mejor C”: obligación de declarar todas las funciones, manejo de tipos, etc.
  - “object based C++”: Beneficios en la organización de código, agrupamiento de estructuras de datos, namespaces, constructores, destructores y algo de herencia simple. Muchos quedan trabados en esta etapa ya que se obtienen grandes beneficios sin mucho esfuerzo mental.
  - “true object oriented programming”. Las funciones virtuales mejoran el concepto de tipo en lugar de solo encapsular código dentro de estructuras.

# Upcasting

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

```
enum note { middleC, Csharp, Eflat }; // Etc.

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

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

void f() {
    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.

# Funciones virtuales

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

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

# Extensibilidad

```
class Instrument {
    public:
        virtual void play(note) const {
            cout << "Instrument::play" << endl;
        }
        virtual const char* what() const {
            return "Instrument";
        }
        // Assume this will modify the object:
        virtual void adjust(int) { }
};
```

# Extensibilidad

```
class Wind : public Instrument {
    public:
        void play(note) const {
            cout << "Wind::play" << endl;
        }
        const char* what() const { return "Wind"; }
        void adjust(int) { }
};

class Percussion : public Instrument {
    public:
        void play(note) const {
            cout << "Percussion::play" << endl;
        }
        const char* what() const { return "Percussion"; }
        void adjust(int) { }
};
```

# Extensibilidad

```
class Brass : public Wind {
    public:
        void play(note) const {
            cout << "Brass::play" << endl;
        }
        const char* what() const { return "Brass"; }
};

class Woodwind : public Wind {
    public:
        void play(note) const {
            cout << "Woodwind::play" << endl;
        }
        const char* what() const { return "Woodwind"; }
};
```

# Extensibilidad

```
// Identical function from before:
void tune(Instrument& i) {
    // ...
    i.play(middleC);
}

// New function:
void f(Instrument& i) { i.adjust(1); }

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

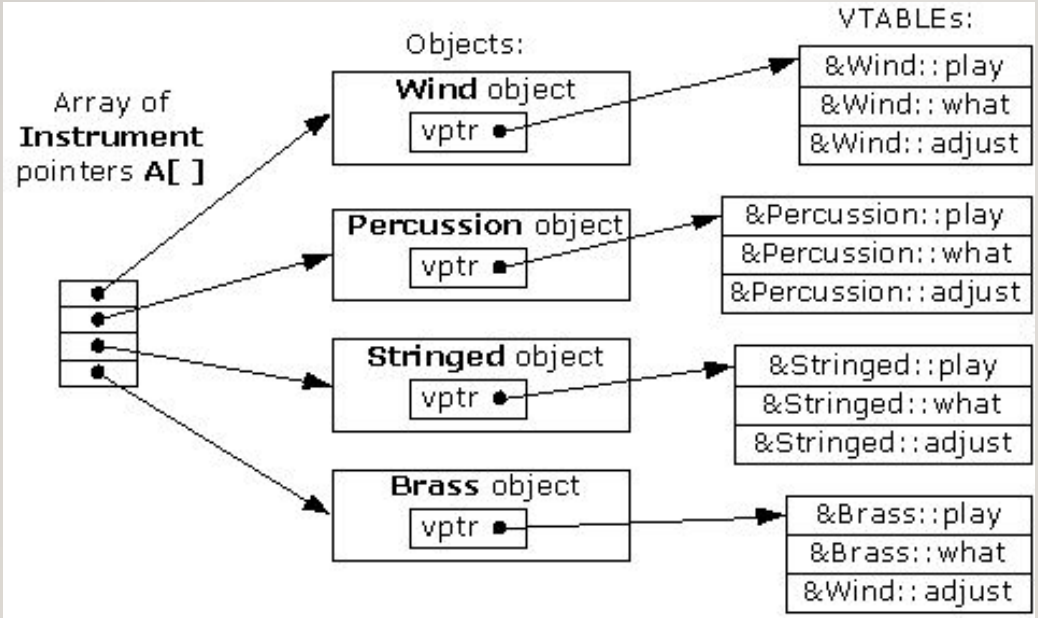
# Extensibilidad

```
int main() {
    Wind flute;
    Percussion drum;
    Stringed violin;
    Brass flugelhorn;
    Woodwind recorder;

    tune(flute);
    tune(drum);
    tune(violin);
    tune(flugelhorn);
    tune(recorder);

    f(flugelhorn);

    return 0;
}
```

# VTABLE

- La palabra virtual obliga al compilador a instalar los mecanismos necesarios para realizar “late binding”.
- Para ello típicamente se crea una tabla simple (llamada VTABLE) para cada clase que contiene funciones virtuales.
- Coloca las direcciones de las funciones virtuales en la tabla y un puntero a la tabla (vpointer o vtable_ptr) en cada objeto de la clase.

# VTABLE


VTABLEs:
Objects:
Array of
Instrument
pointers A[ ]
Wind object
vptr
&Wind::play
&Wind::what
&Wind::adjust
Percussion object
vptr
&Percussion::play
&Percussion::what
&Percussion::adjust
Stringed object
vptr
&Stringed::play
&Stringed::what
&Stringed::adjust
Brass object
vptr
&Brass::play
&Brass::what
&Wind::adjust

# Clases bases abstractas y funciones virtuales puras

- Frecuentemente en un diseño se desea que una clase base presente solo una interface a sus clases derivadas.
- Clases abstractas, se logra declarando al menos una función virtual pura.
- Cuando una clase abstracta es derivada, todas las funciones virtuales puras deben ser implementadas o la clase derivada también será abstracta.

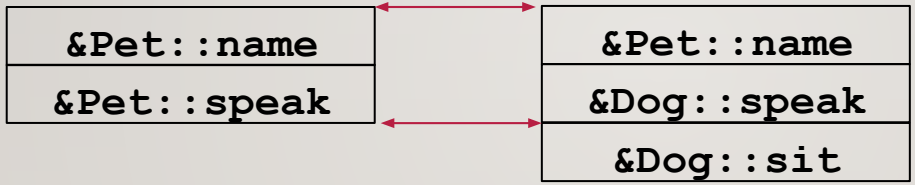
# Funciones virtuales

```
class Instrument {
    public:
        // Pure virtual functions:
        virtual void play(note) const = 0;
        virtual const char* what() const = 0;
        virtual void adjust(int) = 0;
};
```

# VTABLE en la herencia

```
class Pet {
        string pname;
    public:
        Pet(const string& petName) : pname(petName) { }
        virtual string name() const  { return pname; }
        virtual string speak() const { return ""; }
};
class Dog : public Pet {
        string name;
    public:
        Dog(const string& petName) : Pet(petName) { }
        virtual string sit() const {
            return Pet::name() + " sits";
        }
        string speak() const override {
            return Pet::name() + " says 'Bark!'";
        }
};
```

# VTABLE en la herencia

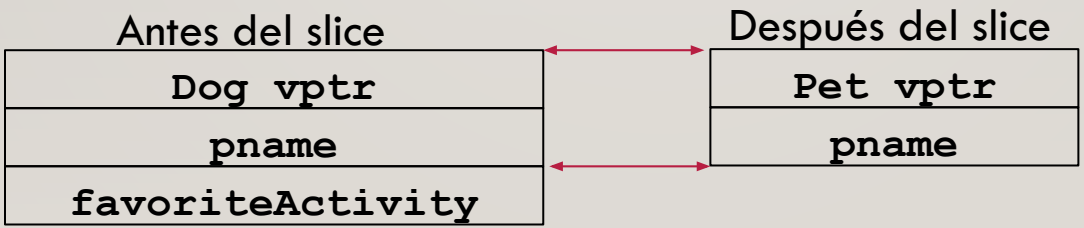


# Object slicing

```
class Pet {
        string pname;
    public:
        Pet(const string& name) : pname(name) {}
        virtual string name() const { return pname; }
        virtual string description() const {
            return "This is " + pname;
        }
};

class Dog : public Pet {
        string favoriteActivity;
    public:
        Dog(const string& name, const string& activity)
            : Pet(name), favoriteActivity(activity) { }
        string description() const {
            return Pet::name() + " likes to " + favoriteActivity;
        }
};
```

# Object slicing

```
void describe(Pet p) { // Slices the object
     cout << p.description() << endl;
}

void f() {
     Pet p("Alfred");
     Dog d("Fluffy", "sleep");
     describe(p);
     describe(d);
}
```



# Overloading y overriding

```
class Base {
    public:
        virtual int f() const {
            cout << "Base::f()\n";
            return 1;
        }
        virtual void f(string) const {}
        virtual void g() const {}
};

class Derived1 : public Base {
    public:
        void g() const {}
};

class Derived2 : public Base {
    public:
        // Overriding a virtual function:
        int f() const {
            cout << "Derived2::f()\n";
            return 2;
        }
};
```

```
class Derived3 : public Base {
    public:
        // Cannot change return type:
        //! void f() const {
        //!   cout << "Derived3::f()\n";
        //! }
};

class Derived4 : public Base {
    public:
        // Change argument list:
        int f(int) const {
            cout << "Derived4::f()\n";
            return 4;
        }
};
```

# Overloading y overriding

```
void f() {
    string s("hello");
    Derived1 d1;
    int x = d1.f();
    d1.f(s);

    Derived2 d2;
    x = d2.f();
    //!  d2.f(s);     // string version hidden

    Derived4 d4;
    x = d4.f(1);
    //!  x = d4.f(); // f() version hidden
    //!  d4.f(s);     // string version hidden

    Base& br = d4;    // Upcast
    //!  br.f(1);     // Derived version unavailable
    br.f();           // Base version available
    br.f(s);          // Base version available
}
```

# Overload control

```
struct B0 {
    void f(int) const;
    virtual void g(double);
};

struct B1 : B0 { /* ... */ };
struct B2 : B1 { /* ... */ };
struct B3 : B2 { /* ... */ };
struct B4 : B3 { /* ... */ };
struct B5 : B4 { /* ... */ };

struct D : B5 {
    void f(int) const;          // override f() in base class
    void g(int);                // override g() in base class
    virtual int h();            // override h() in base class
};
```

# Overload control

```
struct B0 {
    void f(int) const;
    virtual void g(double);
};

struct B1 : B0 { /* ... */ };
struct B2 : B1 { /* ... */ };
struct B3 : B2 { /* ... */ };
struct B4 : B3 { /* ... */ };
struct B5 : B4 { /* ... */ };

struct D : B5 {
    void f(int) const override; // override f() in base class
    void g(int) override;       // override g() in base class
    virtual int h() override;   // override h() in base class
};
```

# Tipo variante de retorno

```
class PetFood {
public:
  virtual string foodType() const = 0;
};

class Pet {
public:
  virtual string type() const = 0;
  virtual PetFood* eats() = 0;
};

class Bird : public Pet {
public:
  string type() const { return "Bird"; }
  class BirdFood : public PetFood {
  public:
    string foodType() const {
      return "Bird food";
    }
  };
  // Upcast to base type:
  PetFood* eats() { return &bf; }
private:
  BirdFood bf;
};
```

```
class Cat : public Pet {
public:
  string type() const { return "Cat"; }
  class CatFood : public PetFood {
  public:
    string foodType() const { return "Birds"; }
  };
  // Return exact type instead:
  CatFood* eats() { return &cf; }
private:
  CatFood cf;
};
void f() {
  Bird b;
  Cat c;
  Pet* p[] = { &b, &c, };
  for( int i=0; i<sizeof(p)/sizeof(*p); ++i )
    cout << p[i]->type() << " eats "
         << p[i]->eats()->foodType() << endl;
  // Can return the exact type:
  Cat::CatFood* cf = c.eats();
  Bird::BirdFood* bf;
  // Cannot return the exact type:
  //!  bf = b.eats();
  // Must downcast:
  bf = dynamic_cast<Bird::BirdFood*>(b.eats());
}
```

# Funciones virtuales en constructores y destructores

- Si se llaman a funciones virtuales dentro de un constructor o en un destructor solo la versión local de la función es utilizada, es decir, el mecanismo virtual no trabaja dentro del constructor ni destructor.
- Dentro del constructor, el objeto puede estar parcialmente formado, solo sus clases bases están construidas, pero los heredadas están pendientes.
- Dentro del destructor, el objeto puede estar parcialmente formado, solo sus clases bases están construidas, pero las heredadas ya están destruidas.

# Destructores virtuales

- No se puede utilizar `virtual` en constructores.
- Se puede y frecuentemente se usan destructores virtuales.

```
class Base1 {
public:
  ~Base1() { cout << "~Base1()\n"; }
};

class Derived1 : public Base1 {
public:
  ~Derived1() { cout << "~Derived1()\n"; }
};

class Base2 {
public:
  virtual ~Base2() { cout << "~Base2()\n"; }
};
```

```
class Derived2 : public Base2 {
public:
  ~Derived2() { cout << "~Derived2()\n"; }
};

void f() {
  Base1* bp = new Derived1;  // Upcast
  delete bp;

  Base2* b2p = new Derived2; // Upcast
  delete b2p;
}
```

# Destructores virtuales puros

```
class AbstractBase {
    public:
        virtual ~AbstractBase() = 0;
};

AbstractBase::~AbstractBase() { }


class Derived : public AbstractBase {
    // No overriding of destructor necessary?
};

void f() {
    Derived d;
}
```

# Jerarquía de clases: Herencia múltiple

- En general, una clase es construida desde un entramado de clases bases denominado jerarquía de clases.
- Una clase puede tener mas de una clase base directa.

```
class Satellite : public Task, public Displayed {
    // ...
};
```

- La clase **Satellite** hereda la unión de las operaciones soportadas por **Task** y **Displayed**.

```
void f(Satellite &s) {
    s.draw();      // Displayed::draw()
    s.delay(10) ; // Task::delay()
    s.transmit(); // Satellite::transmit()
}
```

# Jerarquía de clases: Herencia múltiple

- Un **`Satellite`** puede ser pasado a funciones que esperan un **`Task`** o un **`Displayed`**.

```
void highlight(Displayed *);
void suspend(Task *);

void g(Satellite *p)
{
     highlight(p);
     suspend(p);
}
```

# Herencia múltiple: resolución de ambigüedades

- Dos clases bases pueden tener funciones miembro con el mismo nombre.

```
class Task {
// ...
    virtual debug_info *get_debug();
};
class Displayed {
// ...
    virtual debug_info *get_debug();
};

void f(Satellite *sp) {
    debug_info *dip = sp->get_debug(); // error: ambiguous
    dip = sp->Task::get_debug();       // ok
    dip = sp->Displayed::get_debug();  // ok
}
```

# Herencia múltiple: resolución de ambigüedades

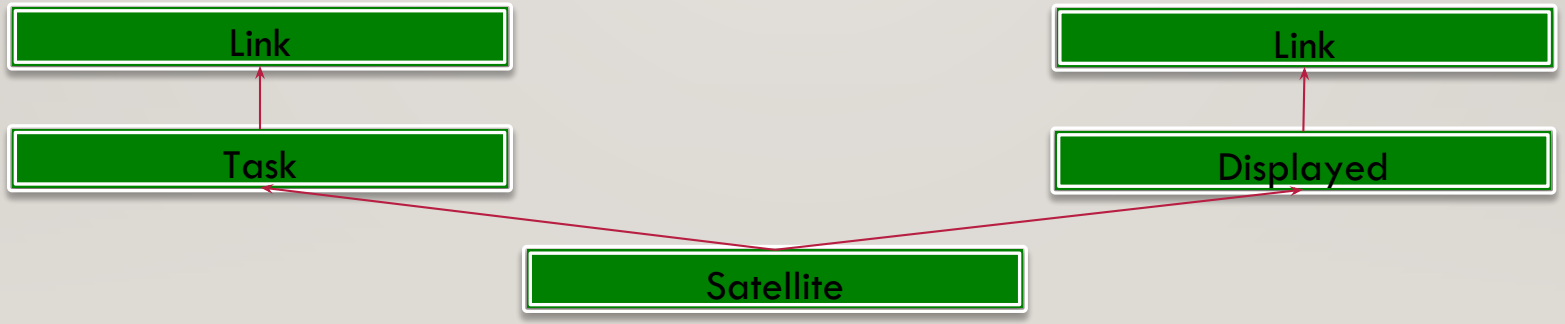
- Una mejor solución al problema anterior.

```
class Satellite : public Task, public Displayed {
    // ...
    debug_info *get_debug() {
        debug_info *dip1 = Task::get_debug();
        debug_info *dip2 = Displayed::get_debug();
        return dip1->merge(dip2);
    }
};
```

- Esta solución localiza la información de las clases base de `Satellite`. `Satellite::get_debug()`, sobrescribe la función `get_debug` de las dos clases bases y por lo tanto esta función es llamada cuando se active `get_debug` sobre un objeto `Satellite`.

# Herencia múltiple: Clases base duplicadas

```
struct Link { Link *next; };
class Task : public Link {
  // the Link is used to maintain a list of all Tasks
  // (the scheduler list)
  // ...
};
class Displayed : public Link {
  // the Link is used to maintain a list of all Displayed
  // objects (the display list)
  // ...
};
```



# Herencia múltiple: Clases base duplicadas

```
void mess_with_links(Satellite *p)
{
    p->next = 0;          // error: ambiguous (which Link?)
    p->Link::next = 0;    // error: ambiguous (which Link?);

    p->Task::Link::next = 0;       // ok
    p->Displayed::Link::next = 0; // ok
    // ...
}
```

# Herencia múltiple: Clase base virtual

```
class Storable {
    public:
        virtual const char *get_file() = 0 ;
        virtual void read() = 0 ;
        virtual void write() = 0 ;
        virtual ~Storable() { write(); }
};
class Transmitter : public Storable {
    public:
        void write();
        //...
};
class Receiver : public Storable {
    public:
        void write();
        //...
};
```

# Herencia múltiple: Clase base virtual

```
class Radio : public Transmitter, public Receiver {
    public:
        const char *get_file();
        void read();
        void write();
        //...
};

void Radio::write() {
    Transmitter::write();
    Receiver::write();
    // write radiospecific information
}
```

# Herencia múltiple: Clase base virtual

```
class Storable {
    public:
        Storable(const char *fn);
        virtual void read() = 0 ;
        virtual void write() = 0 ;
        virtual ~Storable();
    private:
        const char *store;

        Storable (const Storable &);
        Storable &operator=(const Storable &);
};
```

- Este cambio obliga a cambiar el diseño de la clase `Radio`.

# Herencia múltiple: Clase base virtual

```
class Transmitter: public virtual Storable {
    public:
        void write();
        //...
};
class Receiver : public virtual Storable {
    public:
        void write();
        //...
};
class Radio : public Transmitter, public Receiver {
    public:
        void write();
        //...
};
```