

# C++

Juan A. Romero - [aromero@uco.es](mailto:aromero@uco.es)  
v2.0 - 2018

# Breve Historia

- *Bjarne Stroustrup* (Dinamarca) lo comienza en 1979 en *Bell Labs* (EEUU)
- Un C con clases. Considerado LPOO “no *puro*”
- Desde ISO 1998 hasta ISO C++17
- Estándar actual: C++17
- Más en <http://isocpp.org>

# Compilando

- g++ (es GCC, *the GNU compiler collection*)  
<http://gcc.gnu.org>
- Extensiones recomendadas de los ficheros fuente:  
.h y .cc
- Guardas de inclusión en .h y cada fichero .cc  
tendrá sus propios *includes* independientes
- Redireccionar la salida de errores de gcc o g++ a  
un fichero:

```
g++ prueba.c &> salida.txt
```

( &> redirecciona salida estándar al fichero )

# Hello C++

- Comentarios de una línea //
- Objetos *cin* y *cout*, #include <iostream>
- Espacio de nombres  
`namespace nombre { ... }`
- `using namespace std;`  
`using namespace std::cin;`  
`using namespace std::cout;`
- El tipo *bool*
  - Valores: `true` y `false`

# string

- La clase string

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

- Métodos:

- Asignar con =, concatenar con +
- Métodos length(), find()
- Obtener el char \* con el método c\_str()
- <, >, <=, >=, ==, !=
- Etc.

- Más en:

- <http://en.cppreference.com/w/>
- <http://www.cplusplus.com/>

# typedef, const, parámetros defecto

- No es necesario `typedef`
- Declaración de variables en cualquier lugar (siempre antes de su uso)
- Constantes con `const`: datos y parámetros
- Parámetros por defecto: al final de la lista de parámetros y solo en la declaración de la función (solo en el fichero .h)

```
int activa(float a, int b=0, int c=1)
```

# Funciones *inline*

- Funciones *inline* (en línea) para optimizar la ejecución de funciones cortas
- Se escribe “inline” delante de la declaración de la función (en el .h), en el cuerpo (en el .cc), o en ambos (mejor)
- Si simplemente se escribe el código en el .h, se considera inline.
- Si la función no es corta, C++ no la considerará inline

# Referencias

- Referencias (alias):

- int i;
- int &p=i;
- p= 77;

- Referencias como parámetros:

```
void intercambia(int &a, int &b) {  
    int aux=a;  
    a=b;  
    b=aux;  
}
```

- Paso de param. usando referencias es eficiente y rápido. Es la forma adecuada para el paso de objetos en C++
- Si el parámetro no se modifica: referencias constantes



# Sobrecarga de funciones

- Muy útil y cómodo
- También dentro de una misma clase

```
void intercambia(int &a, int &b);
```

```
void intercambia(float &a, float &b);
```

# class

```
class Point{  
    private:  
        int x_, y_;  
    public:  
        void set(int x, int y){x_=x;y_=y;}  
        int getx(){return x_;};  
        int gety(){return y_;};  
};  
Point p;  
p.set(2,-1);  
cout << p.getx() << "\n";  
cout << p.gety() << endl;
```

# Constructores y destructores

```
// inicializa el objeto
```

```
Point::Point() {...}
```

```
// tareas de finalización del objeto
```

```
Point::~~Point() {...}
```

# Iniciadores (de miembros)

```
class A{  
private:  
    int x_, y_;  
    ...  
public:  
    A() : x_(1), y_(1) {};  
    ...  
};
```

(si la declaración de la función está en el .h y el cuerpo en el .cc, debe ir en .cc)

# Métodos constantes

```
int Date::getDay() const {return day_};
```

```
int Date::setDay(int d);
```

si se usa .h y .cc hay que ponerlo en los dos

```
const Date birthDate(1,1,1970);
```

```
birthDate.getDay();
```

```
birthDate.setDay(5); //setDay() is not const
```

**IMPORTANTE:** si una función recibe un objeto `const`, solo podrá invocar métodos `const` de ese objeto

# Constructor de copia

obj2
a=
b=

obj1
a=7
b=3

Copia por defecto de un objeto:

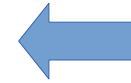
```
MiClase obj1;
```

```
MiClase obj2(obj1);
```

```
MiClase obj2=obj1;
```

*iguales*

obj2
a=7
b=3



obj1
a=7
b=3

constructor de copia por defecto: *solo asignaciones*

Constructor de copia propio:

```
MiClase(const MiClase &e) { ... }
```

(necesario cuando solo las asignaciones no es suficiente)

# Herencia

```
class A{
private:
    int n_;
public:
    void setN(int x);
    int getN();
};

class B: public A{
private:
    int m_;
public:
    void setM(int x);
    int getM();
};
```

```
int main(void){
    B obj;
    . . .
    obj.setM(3);
    cout << obj.getM()

    // heredada de la clase A
    obj.setN(4);

    // heredada de la clase A
    cout << obj.getN();
    . . .
}
```

# Herencia, iniciadores base

```
class A{
private:
    int n_;
public:
    A(int x){n_=x;}
}
class B: public A{
...};
```

```
// Si x tiene un valor por defecto:
// A(int x=0){n_=x;}
```

```
A obj1;
A obj2(55);
```

```
// Y no sería necesario iniciador:
B::B(...) {...}
```

**Iniciadores base:** si el constructor de la clase A tiene un parámetro obligatorio, se debe enviar desde la clase derivada y usar un iniciador de la clase base:

```
B::B(int y, . . .): A(y){...};
```

O bien

```
B::B(): A(10){...}; // en este caso se envía una cte.
```



## Iniciadores base: ¿dónde?

El iniciador base se considera ya código que ejecuta una llamada a la clase base y solo debe ponerse en el cuerpo del constructor de la clase derivada.

# Variables miembro static

**Static members:** only one copy of the static member is shared by all objects of a class in a program

```
class A{  
...  
static int i; // solo si es const se inicializa aquí  
...  
};  
  
int A::i=4;    // se usa como A::i
```

# Funciones miembro static

**Static Function Members.** By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the static functions are accessed using only the class name and the scope resolution operator `::` before the name of the function.

Solo en el `.h` se pone `static` delante de su prototipo.

Después, para invocarla se escribe:

```
nombre_clase::nombre_funcion()
```

Uso:

- Agrupar funciones independientes en un mismo módulo
- Para poder usarlas sin tener que declarar objetos de su clase

# Inicialización de miembros (y de miembros constantes)

```
Class A{
```

```
private:
```

```
    const int i_; // necesita inicializarse
```

```
    B obj1; //necesita param.
```

```
    C obj2; // necesita param.
```

```
public:
```

```
    A(int edad, float peso):i_(7),  
        obj1(peso), obj2(edad){...}
```

```
    ...
```

```
};
```

# STL: Standard Template Library

Contiene: funciones, algoritmos, iteradores y contenedores:

- Vector
  - <http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html#VECTOR>
- List (enlaces en la web de la asignatura):
  - <http://www.cplusplus.com/reference/list/list/>
  - <http://www.yolinux.com/TUTORIALS/LinuxTutorialC++STL.html#LIST>
- stack
- queue
- set, map, hash, etc
- Más en:
  - <http://en.cppreference.com/w/>
  - <http://www.cplusplus.com/>

# Excepciones

```
try{...} // código a monitorizar  
catch (int &i){...}  
catch (float &i) {...}  
catch (...) {...}
```

- El código que se *monitoriza* (try) puede lanzar excepciones de varios tipos que se capturan en cada catch
- **Ejemplos en moodle:** ejemplo1Ex.cc, ejemplo2Ex.cc, ejemplo3Ex.cc y ejemploEx4.cc

# this

```
class A{  
private:  
    int i_;  
...  
};
```

Accediendo al dato desde un método de la clase:

```
i_ = x;  
es igual que  
this->i_ = x; //puntero al objeto actual (this)
```

# Sobrecarga de operadores

```
class Punto{
public:
    Punto(int x, int y){x_=x; y_=y;};
    int getX(){return x_};
    int getY(){return y_};
    Punto operator=(const Punto &p);
    Punto operator+(const Punto &p);
    Punto operator++(void); // para ++p
    Punto operator++(int); // para p++
private:
    int x_, y_;
};

Punto Punto::operator=(const Punto &p)
{
    x_ = p.x_;
    y_ = p.y_;
    return *this;
}

Punto Punto::operator+(const Punto &p)
{
    Punto aux;
    aux.x_ = x_ + p.x_;
    aux.y_ = y_ + p.y_;
    return aux;
}
```

```
Punto Punto::operator++(void)// ++b;
{
    x_++;
    y_++;
    return *this;
}

Punto Punto::operator++(int)// b++;
{
    Punto aux=*this; // OJO!!
    x_++;
    y_++;
    return aux;
}

.....
int main(void)
{
    Punto a(1,2),b(10,10),c(0,0);
    c=a+b;
    c=a+b+c;
    b++;
    c.out << "b.x= " << b.getX() <<
    "b.y= " << b.getY() << endl;
    c.out << "c.x= " << c.getX() <<
    "c.y= " << c.getY() << endl;
}
```



# Funciones friend

```
class Circulo;

class Linea{
private:
    int color_;
public:
    friend mismoColor(Linea l, Circulo c);
...
};

class Circulo{
private:
    int color_;
public:
    friend mismoColor(Linea l, Circulo c);
...
};
```

```
int mismoColor(Linea l, Circulo c)
{
    if (l.color_==c.color_)
        return 1;
    else
        return 0;
}
```

Las funciones friend no reciben  
el puntero `this`

# Sobrecarga de operadores con funciones friend

```
class Contador{  
private:  
    int valor_;  
public:  
    Contador() {valor_=0;}  
    int Contador::operator+(int x)  
        {return valor_+x;}  
...  
};  
  
Contador c;  
int i;  
...  
...  
i = c + 10;  
i = 10 + c // No funciona
```

SOLUCIÓN:

```
class Contador{  
public:  
    friend int operator+(Contador ob, int i);  
    friend int operator+(int i, Contador ob);  
.....  
private:  
    int valor_;  
};  
  
// esta podría no ser friend  
int operator+(Contador ob, int i) {  
    return ob.get() + i;  
}  
  
int operator+(int i, Contador ob) {  
    return i+ob.get();  
}
```

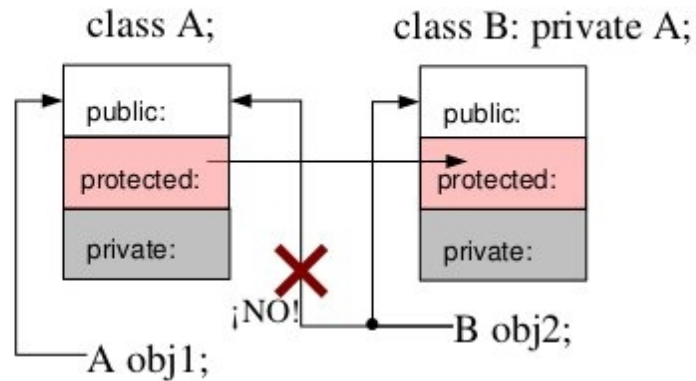
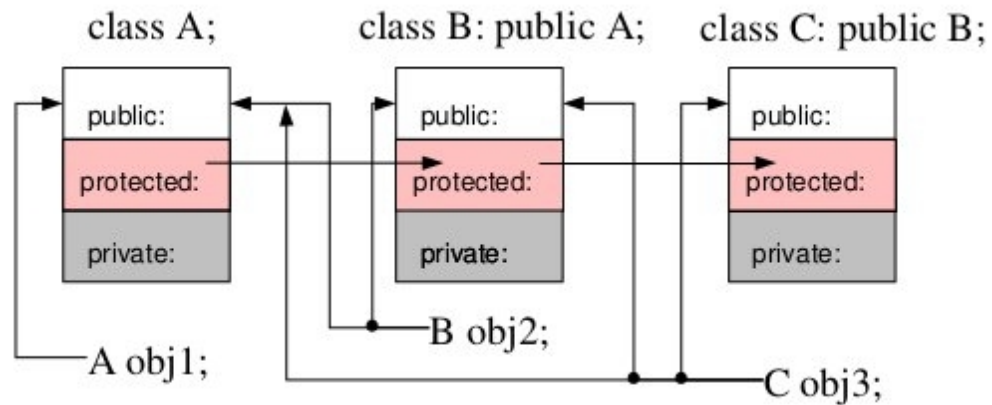
# Herencia private

```
class Pila: private Lista
```

Ejemplo:

- Una pila “**NO ES**” una lista
- Una pila se crea “**POR MEDIO DE**” una lista

# Tipos de herencia y accesos



Tipos de accesos:

public, private, protected

# Herencia múltiple

```
class C: public A, public B{  
    ...  
};
```

# Objetos pasados por valor (como cualquier variable)

```
class X
{
private:
    int a_;
public:
    X(){a_=5;}; // tres funciones en linea
    int set(int i){a_=i;};
    int get(){return a_;};
};

void f(X obj){obj.set(8);}

int main(void)
{
    X x;
    cout << x.get();
    f(x);
    cout << x.get();
}
```

```
$ g++ prueba.cc -o prueba
$ ./prueba
5
5
```

# Punteros a objetos

```
Fecha f;  
Fecha *p;  
p=&f;  
cout << p->getDay(); // f.getDay()
```

O bien:

```
Fecha *p;  
P= new Fecha(1,1,1970);  
cout << p->getDay();
```

# Funciones virtuales

```
class Figura{
protected:
double x_, y_;
public:
void setDim(double i, double j){
x_=i;y_=j;
};
virtual double area(){
cout << "no definida aqui";return 0.0;
};

class Triangulo : public Figura{
public:
virtual double area(){
// x_= base, y_= altura
return(x_*y_/2);
}
};

class Cuadrado : public Figura{
public:
virtual double area(){
// x_, y_ lados
return(x_*y_);
}
};
```

Figura es una  
clase abstracta

```
int main(void)
{
int opcion;
Figura *p;
cout << "elige figura \n"
<< "1.- triangulo \n"
<< "2.- cuadrado \n";
cin >> opcion;
if(opcion==1)
{
Triangulo t;
p=&t;
}
else
{
Cuadrado c;
p=&c;
}
p->setDim(3.0,4.0);
cout << "\n Area = " << p->area() <<
endl;
}
```

puntero a  
clase base

vinculación  
dinámica



# Funciones virtuales (clases abstractas)

```
class Figura{
protected:
    double x_, y_;
public:
    virtual double area()=0;
    virtual double pinta()=0;
    virtual double borra()=0;
    virtual double color()=0;
    virtual double mueve()=0;
    virtual double escala()=0;
};
```

```
class Triangulo : public Figura{
public:
    virtual double area(){...};
    virtual double pinta(){...};
    virtual double borra(){...};
    virtual double color(){...};
    virtual double mueve(){...};
    virtual double escala(){...};
    ...
};
```

```
class Cuadrado : public Figura{
public:
    virtual double area(){...};
    virtual double pinta(){...};
    virtual double borra(){...};
    virtual double color(){...};
    virtual double mueve(){...};
    virtual double escala(){...};
};
...
```

Figura es una  
clase abstracta

función  
polimórfica

```
int cambiaFigura(Figura *p)
{
    p->borra();
    p->mueve(2);
    p->escala(100);
    p->color(77);
    p->pinta();
}
```

vinculación  
dinámica

# Funciones virtuales puras

```
virtual tipo nombre_funcion(params...) = 0;
```

Funciones virtuales y virtuales puras se usan en clases abstractas

(en C++ se considera que es clase abstracta solo si tiene funciones virtuales puras):

- Definen la interfaz genérica
- No se pueden definir objetos de estas clases
- Son usadas para derivar de ellas y para el polimorfismo

# Funciones virtuales puras (clases abstractas)

```
#include <iostream>
#include <string>
using namespace std;
class Animal
{
private:
    string name_;
public:
    Animal(string name):name_(name){}
    string getName(){return name_;}
    virtual string talk() = 0;
};

class Gato: public Animal
{
public:
    Gato(string name):Animal(name){}
    virtual string talk(){return "Miau!";}
};

class Perro: public Animal
{
public:
    Perro(string name):Animal(name){}
    virtual string talk(){return "Guau!";}
};
```

```
int main()
{
    Gato g("milu");
    Perro p("boby");
    Apuntes de C++
    cout << g.talk() << endl;
    cout << p.talk() << endl;
}
```

Si no se declara talk() dentro de las clases derivadas:

animal.cc: En la función 'int main()':

animal.cc:44:7: error: no se puede declarar que la variable 'p' sea del tipo abstracto 'Perro'

animal.cc:29:7: nota:

porque las siguientes funciones virtual son puras dentro de 'Perro':

animal.cc:15:25: nota: virtual const char\* Animal::talk()

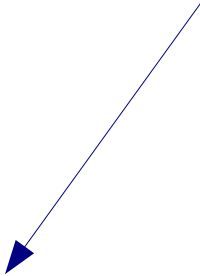
# Funciones virtuales puras (clases abstractas)

```
#include <iostream>
#include <string>
using namespace std;
class Animal
{
private:
    string name_;
public:
    Animal(string name):name_(name){}
    string getName(){return name_;}
    virtual string talk() = 0;
};

class Gato: public Animal
{
public:
    Gato(string name):Animal(name){}
    virtual string talk(){return "Miau!";}
};

class Perro: public Animal
{
public:
    Perro(string name):Animal(name){}
    virtual string talk(){return "Guau!";}
};
```

función  
polimórfica



```
void do_miau(Animal *p)
{
    p->talk();
}
```

# Plantillas de función

template  
function

```
template <class T>
```

```
void print_vector(T *v, const int n)
{
    for(int i=0;i<n;i++)
        cout << v[i] << " , ";
}
```

```
int main(void)
{
    int a[5]={1,3,5,7,9};
    float b[4]={5.6, 7.8, 3.9, 1.2};
    char c[5]="hola";
    cout << "vector de enteros";
    print_vector(a, 5);
    cout << "vector de floats";
    print_vector(b, 4);
    cout << "vector de char";
    print_vector(c, 4);
}
```

- “T” parámetro formal (tipo genérico)
- Se usa class o typename
- Puede haber varios tipos genéricos:  
    <class T1, class T2>
- Pueden tener cualquier nombre

# Plantillas de clase

(clases genéricas, *class template*)

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

template <class T> class MiClase{
private:
    T x_, y_;
public:
    MiClase (T a, T b){ x_=a; y_=b;};
    T div(){return x_/y_};
};

int main()
{
    MiClase <int> iobj(10,3);
    MiClase <double> dobj(3.3, 5.5);
    cout << "división entera = " << iobj.div() << endl;

    cout << "división real = " << dobj.div() << endl;
}

salida:
$ ./a.out
division entera = 3
division real = 0.6
```

# Reserva de memoria dinámica (new y delete)

```
int *v;  
float *f;  
v = new int [10] // vector de 10 enteros  
f = new float [5]; // vector de 5 reales  
delete [] v;  
delete [] f;
```

## Objetos:

```
A *obj;  
obj = new A(parametros del constructor);  
delete obj;
```

# E/S en C++. Streams

(ver apuntes)



# E/S en C++. Ficheros

(ver apuntes)

# Insertador (<<) y extractor (>>)

```
#include <iostream>
using namespace std;
class Punto{
private:
    int coordx_;
    int coordy_;
public:
    Punto() {coordx_=coordy_=1;};
    friend ostream &operator<<(ostream
&stream, const Punto &p);
    friend istream &operator>>(istream
&stream, Punto &p);
    ...
};
ostream &operator<<(ostream &stream,
const Punto &p)
{
    stream << "(";
    stream << p.coordx_;
    stream << ", ";
    stream << p.coordy_;
    stream << ")";
    return stream;
}
```

extrator  
propio

insertador  
propio

```
istream &operator>>(istream &stream,
Punto &p)
{
    cout << "Introduce x ";
    stream >> p.coordx_;
    cout << "Introduce y ";
    stream >> p.coordy_;
    return stream;
}
int main(void)
{
    Punto a,b;
    cin >> a;
    cin >> b;
    cout << a << b << endl;
}
```

```
Introduce x 4
Introduce y 1
Introduce x 2
Introduce y 3
(4, 1) (2, 3)
```

# Más . . .

Referencias:

- <http://en.cppreference.com/w/>
- <http://www.cplusplus.com/>
- Bibliografía de la asignatura

**FIN**