

# Introducción a la programación funcional en C++



`using std::cpp` 2014

Joaquín M<sup>a</sup> López Muñoz <joaquin@tid.es>

Madrid, octubre 2014

*Telefonica*



Todo empezó aquí





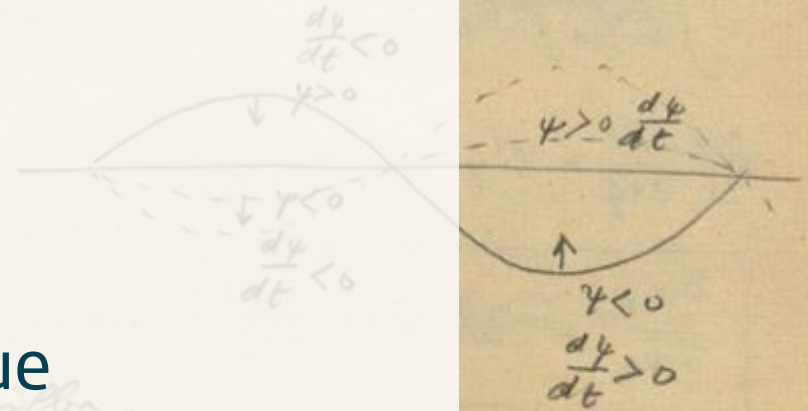
O aquí, según se mire





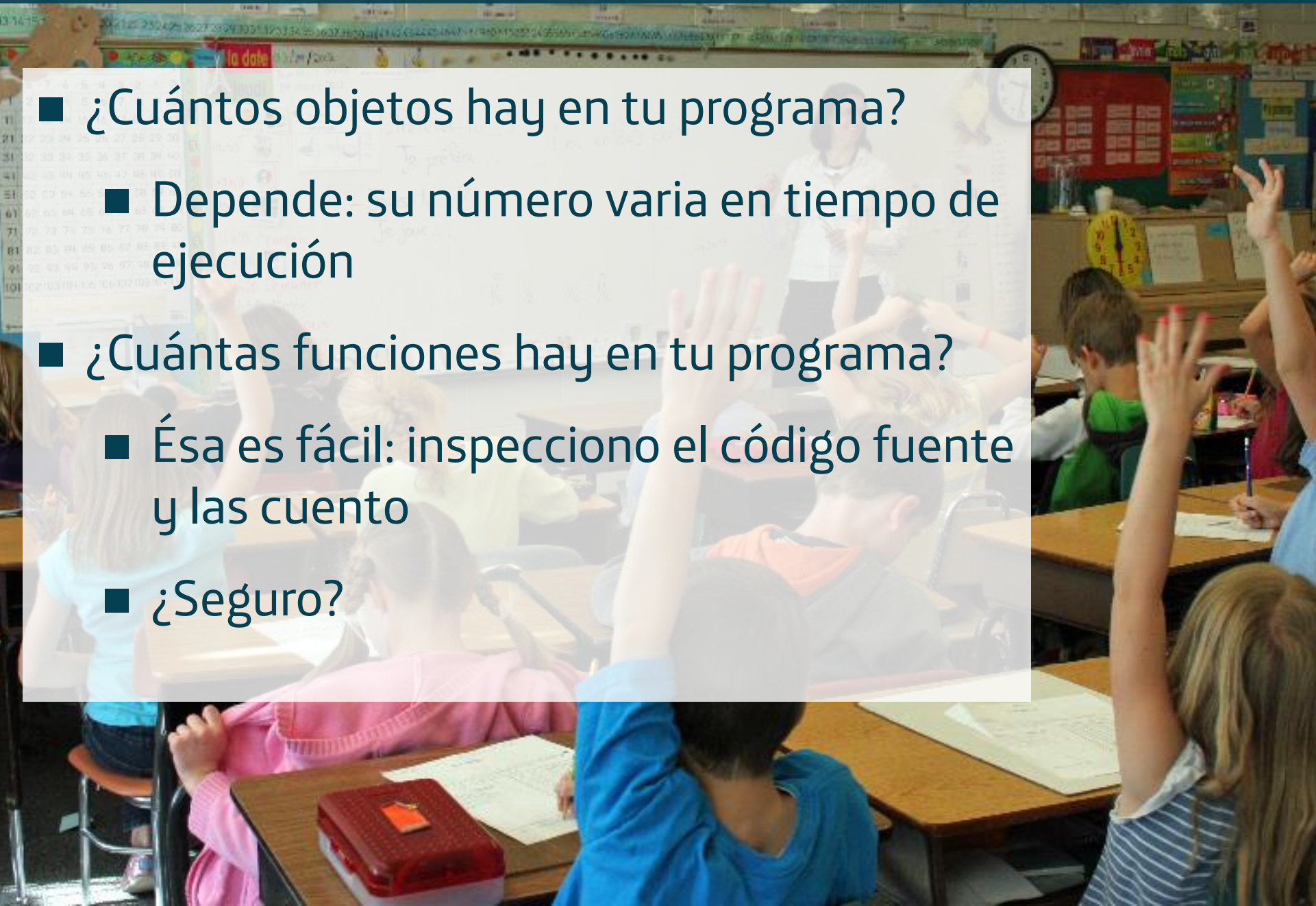
# ¿Qué es una función?

- Una curva
- Una expresión analítica
- Una variable física
- Una relación entre números
- Un subconjunto  $f \subseteq X \times Y$  tal que  $(x, y) \in f, (x, y') \in f \rightarrow y = y'$
- Un procedimiento para obtener un resultado unívocamente a partir de unos argumentos



# Dos preguntas simples

- ¿Cuántos objetos hay en tu programa?
  - Depende: su número varia en tiempo de ejecución
- ¿Cuántas funciones hay en tu programa?
  - Ésa es fácil: inspecciono el código fuente y las cuento
  - ¿Seguro?





# El paradigma de la programación funcional





# El paradigma de la programación funcional

- Funciones ~ ciudadanos de primera
  - Pueden pasarse, devolverse, crearse, combinarse y, claro está, invocarse
- Funciones que aceptan y devuelven funciones
- Función como computación, no como grafo
  - Máquinas de Turing  $\leftrightarrow$  cálculo  $\lambda$
- *Pureza*
  - Inherentemente paralelizables
  - Recursividad como recurso indispensable

# Funciones de orden superior

- Funciones que aceptan y/o devuelven funciones
- Ejemplos en la vida “real”

- Integral definida:

$$I(f, a, b) = \int_a^b f(x)dx$$

- Operadores  $\frac{d}{dx}$  y  $\int dx$

- Composición de funciones:

$$(g \circ f)(x) = g(f(x))$$



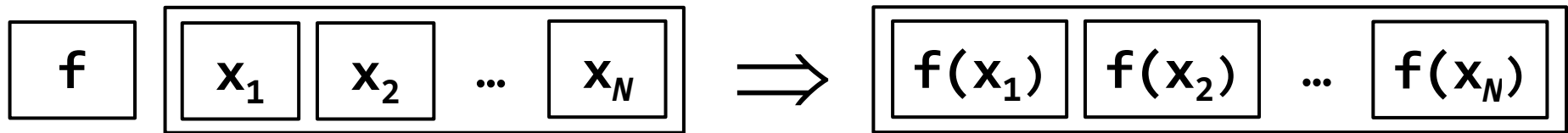
# Funciones que aceptan funciones





# map

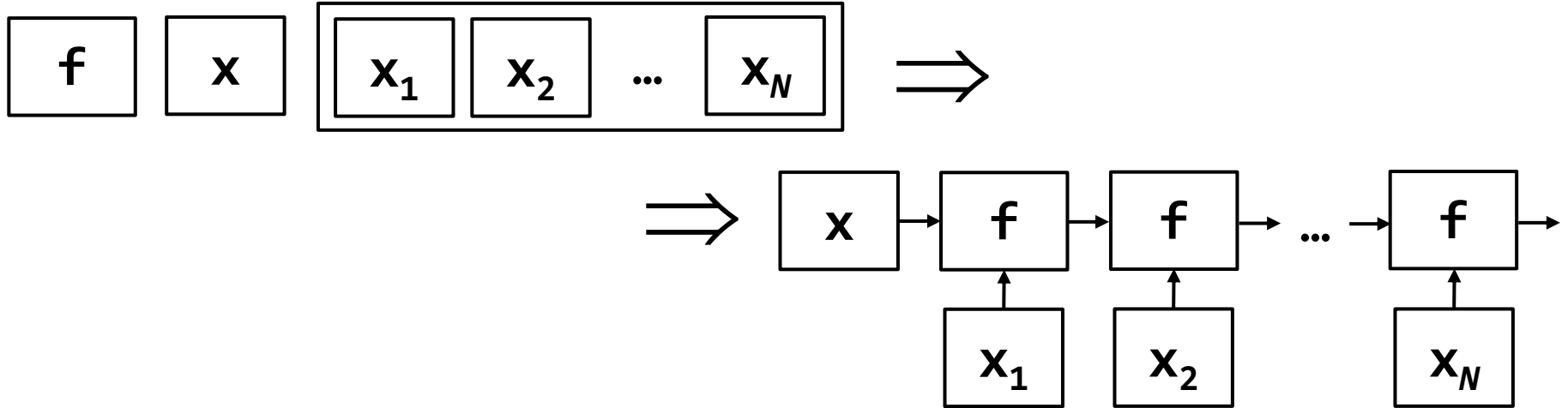
## ■ map f list



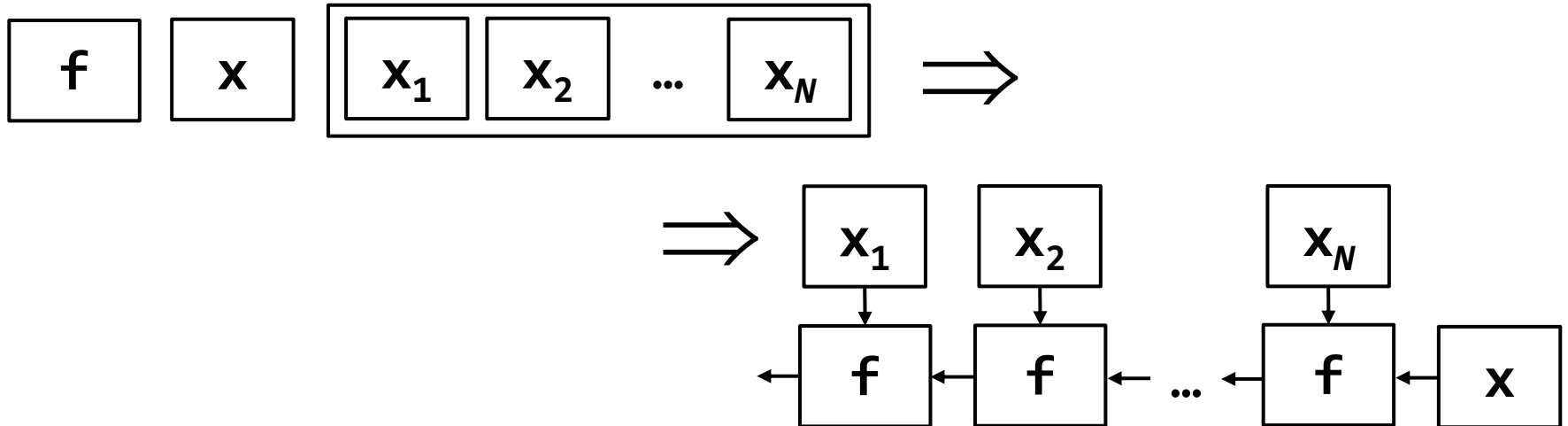


# fold

## ■ foldl f x list



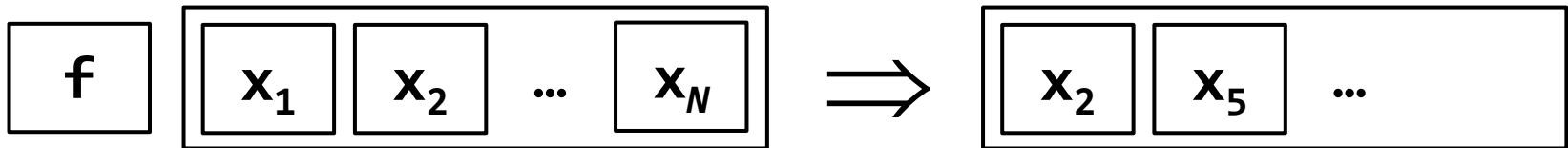
## ■ foldr f x list





# filter

## ■ filter f list

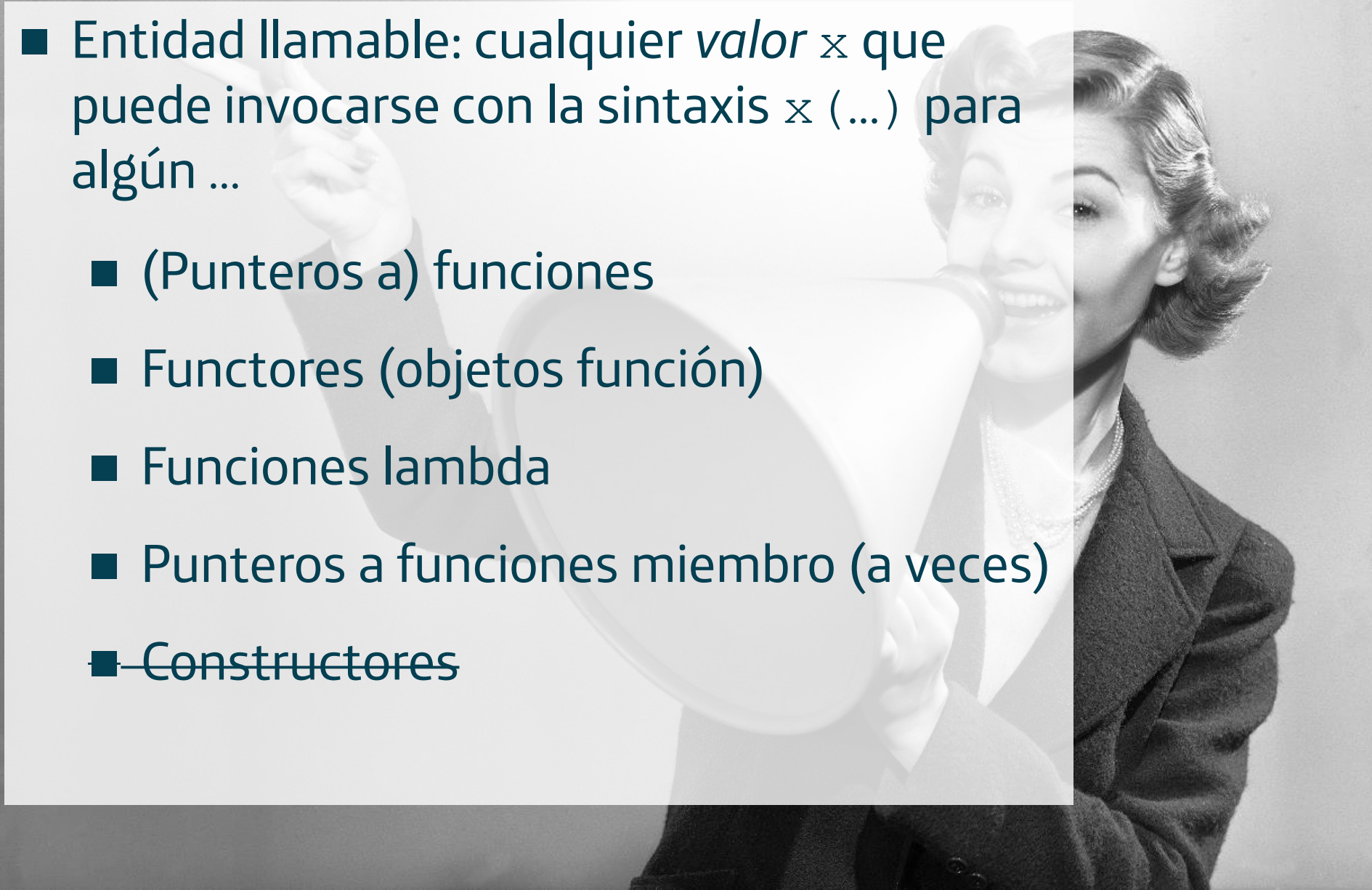


## ■ ¿Equivalentes en C++?

### ■ ¿Qué tipo de funciones aceptan estos algoritmos?

# Una noción amplia de “función” en C++

- Entidad llamable: cualquier *valor*  $x$  que puede invocarse con la sintaxis  $x(\dots)$  para algún ...
  - (Punteros a) funciones
  - Functores (objetos función)
  - Funciones lambda
  - Punteros a funciones miembro (a veces)
  - ~~Constructores~~





# Mecanismos de generación de funciones en C++





# Mecanismos de generación de funciones en C++

- `std::bind`
- Funciones lambda
- `std::function`







- `bind` transforma una entidad llamable de  $n$  argumentos en una entidad llamable de  $m$  argumentos con  $n-m$  parámetros fijos y el resto potencialmente reordenados
- Matemáticas: aplicación parcial de funciones

```
void f(const std::string& s0, const std::string& s1, const std::string& s2){  
    std::cout<<"From "<<s0<<" to "<<s1<<" through "<<s2<<"\n";  
}
```

```
using namespace std::placeholders;
```

```
int main(){  
    auto f1=std::bind(f, "New York", _1, _2); // what's the return type?  
    auto f2=std::bind(f, "Moscow", "Beijing", _1);  
    auto f3=std::bind(f, _2, _1, _3);  
  
    f1("Los Angeles", "Denver");  
    f2("Ulan Bator");  
    f3("Madrid", "Paris", "Barcelona");  
}
```

- Argumentos por valor: `std::cref`, `std::ref` para aceptar referencias

# Un consejo sobre `std::bind`



- No lo uses 😊 excepto en casos muy simples
- Las funciones lambda son más legibles y tienen menos problemas de resolución de nombres

```
auto f=std::bind(std::sin,0); // error: no matching function
```

- Los `binds` anidados tienen un comportamiento peculiar:

```
struct eval{
    template<typename F>
    void operator()(F f,int x)const{std::cout<<"f("<<x<<"")="<<f(x)<<"\n";}
};

int foo(int x,int y){return x+y;}

int main()
{
    auto f=std::bind(foo,5,_1);
    auto g=std::bind(eval(),f,_1); // bind(eval(),bind(foo,5,_1),_1)

    eval()(f,2); // "f(2)=7"
    g(2);       // ??
}
```



# Funciones lambda

- ¿De donde viene este nombre tan peculiar?
- El cálculo  $\lambda$  lo inventó Alonzo Church durante los años 20-30 (en Princeton) para modelar matemáticamente el concepto de computación
  - ¡La piedra angular de la informática teórica!
  - Lisp no es más que un intérprete de cálculo  $\lambda$  con algunos añadidos prácticos
- ¿Por qué usó Church la letra “ $\lambda$ ”?



# Funciones lambda

- ¿Cómo se llama la función que acepta  $x$  y devuelve  $\sin x$ ?
  - Obvio: se llama `sin`
- ¿Y la que acepta  $x$  y devuelve  $(\sin x)^2$ ?
  - ¡ $(\sin x)^2$  no es un nombre sino una expresión!
- Podemos inventárnoslo (tedioso): `squaredsin`
- O convertir la expresión en un nombre:  $\lambda x. (\sin x)^2$
- $\lambda x$  es un constructo sintáctico para indicar que  $x$  es un parámetro formal de la expresión siguiente, *no* el valor  $x$
- En C++: las funciones lambda son funciones sin nombre creadas en el punto de definición



# Funciones lambda en C++

- Ordena un vector de `ints`:

```
std::sort(v.begin(),v.end());
```

- Ordénalo según el *valor absoluto* de sus elementos:

```
bool abscompare(int x,int y) // defined at global scope
{
    return abs(x)<abs(y);
}
```

...

```
std::sort(v.begin(),v.end(),abscompare);
```

- Esto es un rollo: tenemos que definir y *nombrar* una función *lejos de donde la usamos* para usarla *sólo una vez* en el programa
  - Vamos a hacerlo mejor

# Funciones lambda en C++

```
std::sort(  
    v.begin(), v.end(),  
    [](int a, int b){return abs(a)<abs(b);}  
);
```

- La especificación de una función lambda se compone de:

**[closures<sub>opt</sub>]** (**params**)<sub>opt</sub> **->** **ret<sub>opt</sub>** **{body}**

- Capturas (closures) entre corchetes **[]**
- Parámetros (si los hay) entre paréntesis **()**
- El tipo de retorno precedido por **->** (se puede omitir casi siempre)
- El cuerpo de la función entre corchetes **{}**
- Es más simple de lo que parece a primera vista



# Funciones lambda en C++

- Las capturas de una función lambda son las variables del entorno externo usadas en el cuerpo

```
int off;  
[off](int x,int y){return x+y+off;} // off captured by value
```

```
int acc;  
[&acc](int x){acc+=x*x;} // acc captured by reference
```

```
[&acc,off](int x){acc+=(x+off)*(x+off);} // combined
```

- `[]` No se captura nada
- `[&]` Todas las variables mencionadas se capturan por referencia
- `[=]` Todas las variables mencionadas se capturan por valor (copiadas)
- `[this]` Acceso a los miembros de un objeto
- `[x=y]` Captura con renombrado (C++14)

# Funciones lambda en C++

- El tipo de retorno, si es necesario especificarlo, se indica con `->` justo antes del cuerpo de la función

```
[](int x, double y) -> double  
{  
    if(x<y) return x;  
    return y;  
}
```

- En general, el compilador debe poder deducir el tipo de retorno sin nuestra ayuda



# Funciones lambda en C++

```
find_if(v.begin(),v.end(),[](int i){return i>x && i<y;});
```

```
for_each(v.begin(),v.end(),[](int n){acc+=n*n;});
```

```
remove_if(v.begin(),v.end(),[](int n){return x<=n && n<y;});
```

```
for_each(v.begin(),v.end(),[](int& i){i*=2;});
```

```
class scale
{
    int _scale;
    ...
    void apply_scale(const vector<int>& v)const
    {
        for_each(v.begin(),v.end(),
            [this](int n){cout<<n*_scale<<endl;});
    }
};
```

```
remove_if(v.begin(),v.end(),[min=x,max=2*x](int n){return min<=n && n<max;});
```



- A partir de C++14:

```
template<typename Container>
void abs_sort(Container& c){
    using value_type=Container::value_type;
    std::sort(
        c.begin(),c.end(),
        [](const value_type& x,const value_type& y)
        {return abs(x)<abs(y);});
}
```

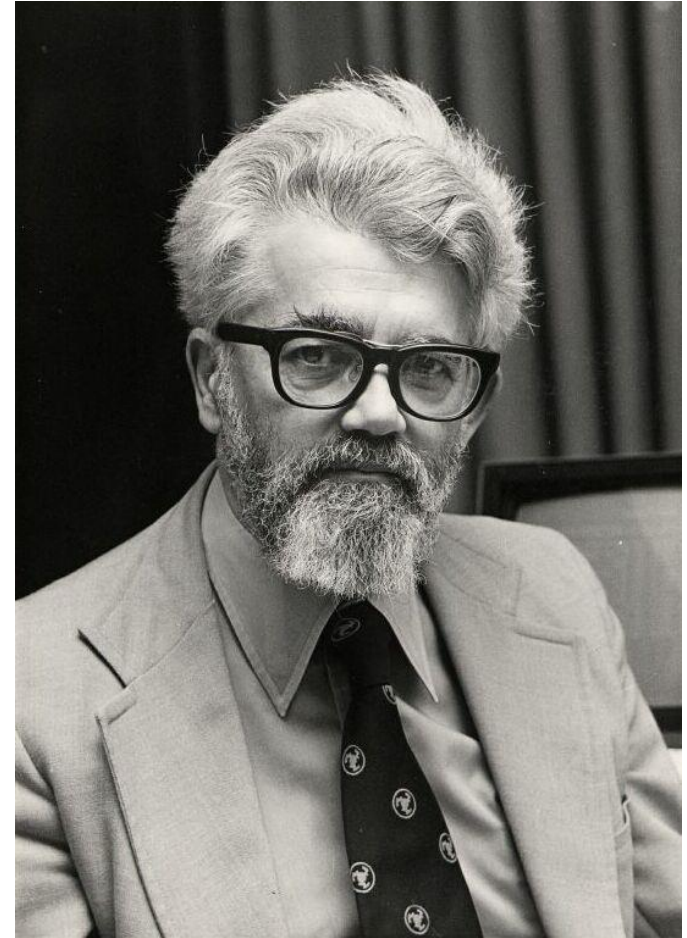
```
template<typename Container>
void abs_sort(Container& c){
    std::sort(
        c.begin(),c.end(),
        [](const auto& x,const auto& y){return abs(x)<abs(y);});
}
```

- Función lambda genérica → compile-time duck typing
- No sólo una sintaxis más concisa
  - Visitor pattern



# Funciones lambda en C++

- Un mecanismo extremadamente flexible para la generación de funciones en tiempo de ejecución
  - Mucho más que una conveniencia para usar los algoritmos de la STL
  - No infravalores el azúcar sintáctico: Leibniz vs. Newton
- Función como contexto de ejecución
- Closure/object equivalence
- Funciones lambda de orden superior: lambdas dentro de lambdas
  - Veremos ejemplos de esto



# std::function

```
int foo(int x){return 2*x;}

auto bar=[](int x){return -x;};

struct baz{
    int operator()(int x)const{return x%2;}
};

int main()
{
    std::vector<...> v={foo,bar,baz()};
    for(const auto& f:v)std::cout<<f(5)<<" ";
}
```

- `function` aloja cualquier tipo de entidad llamable:

```
std::vector<std::function<int(int)>> v={foo,bar,baz()};
```

- Una suerte de puntero a función generalizado
- Programación funcional abstracta
  - Type erasure (run-time duck typing)





## ■ Signatura de una función

function<return\_type (param<sub>1</sub>,...,param<sub>n</sub>)>

function<auto (param<sub>1</sub>,...,param<sub>n</sub>) -> return\_type>

## ■ Signatura ~ interfaz abstracta

```
int f1(const std::string& s){return std::atoi(s.c_str());}  
int f2(const std::string& s1,const std::string& s2){return f1(s1+s2);}
```

```
std::function<int (const std::string&)> f=f1;  
std::cout<<f("123")<<std::endl; // -> 123
```

```
f=std::bind(f2,_1,"000");  
std::cout<<f("123")<<std::endl; // -> 123000
```

```
f=&std::string::size; // wtf?  
std::cout<<f("123")<<std::endl; // -> 3
```

## ■ Signatura compatible → conversión automática

# Casos de estudio





- Dada  $f: \mathbb{R} \rightarrow \mathbb{R}$ , construir  $f': \mathbb{R} \rightarrow \mathbb{R}$  como una entidad llamable
- Esto es, implementar el operador diferencial

$$\begin{aligned} \frac{d}{dx}: (\mathbb{R} \rightarrow \mathbb{R}) &\rightarrow (\mathbb{R} \rightarrow \mathbb{R}) \\ f &\rightarrow f' \end{aligned}$$

- Recordatorio:

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x-h)}{2h}$$

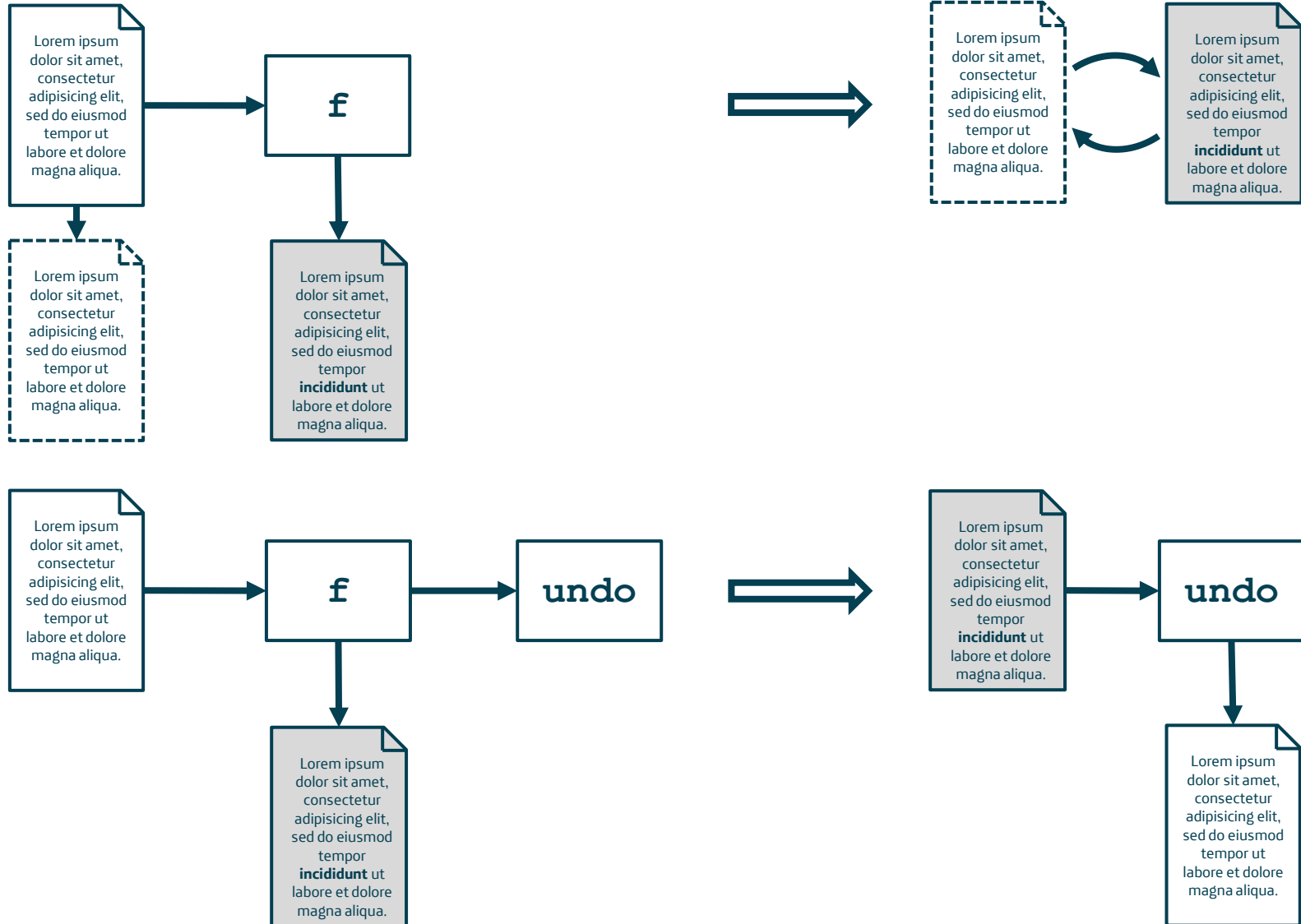


# Caso 2: reversión de comandos

- Implementar un framework para añadir la capacidad de deshacer comandos a un procesador de textos
  - Requisito 1: eficiente
  - Requisito 2: extensible mediante plug-ins

```
class document
{
public:
    typedef vector<string> data;
    bool undo();
    //...
private:
    data dat;
};
//...
append(doc, "martes");
insert(doc, 3, "ustedes");
doc.undo();
doc.undo();
```

# Caso 2: reversión de comandos



# Unas notas antes de partir





# Unas notas antes de partir

- Programación funcional ~ funciones como ciudadanos de 1ª
  - La pureza y la pereza son opcionales
- Función como contexto de ejecución
  - Inversión de control: una nueva perspectiva
- `std::function`: programación funcional abstracta
  - Función como unidad mínima de abstracción
- ¡Queda mucho por aprender!
  - Tipos algebraicos, pattern matching
- Aprende Haskell y/o Lisp

# Introducción a la programación funcional en C++

Gracias

[github.com/joaquintides/usingstdcpp2014](https://github.com/joaquintides/usingstdcpp2014)

using **std::cpp** 2014

Joaquín M<sup>a</sup> López Muñoz <joaquin@tid.es>

Madrid, octubre 2014

*Telefonica*