Programación funcional y orientación a objetos

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

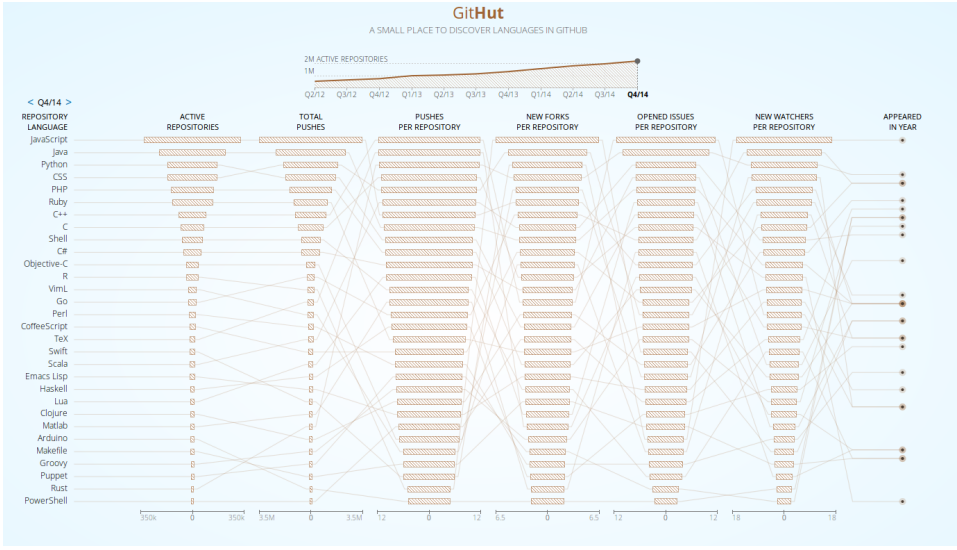
Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20

Motivación


GitHut
A SMALL PLACE TO DISCOVER LANGUAGES IN GITHUB
2M ACTIVE REPOSITORIES
1M
Q2/12 Q3/12 Q4/12 Q1/13 Q2/13 Q3/13 Q4/13 Q1/14 Q2/14 Q3/14 Q4/14
< Q4/14 >
REPOSITORY LANGUAGE
ACTIVE REPOSITORIES
TOTAL PUSHES
PUSHES PER REPOSITORY
NEW FORKS PER REPOSITORY
OPENED ISSUES PER REPOSITORY
NEW WATCHERS PER REPOSITORY
APPEARED IN YEAR
JavaScript
Java
Python
CSS
PHP
Ruby
C++
C
Shell
C#
Objective-C
R
VimL
Go
Perl
CoffeeScript
TeX
Swift
Scala
Emacs Lisp
Haskell
Lua
Clojure
Matlab
Arduino
Makefile
Groovy
Puppet
Rust
PowerShell
350k 0 350k
3.5M 0 3.5M
12 0 12
6.5 0 6.5
12 0 12
18 0 18

Motivación

Los lenguajes funcionales no son los más populares...

Motivación

Los lenguajes funcionales no son los más populares...

... pero los **conceptos de la programación funcional** son útiles y prácticos.

¿Qué **conceptos de programación funcional** podrían ser útiles aplicados a otros paradigmas?

Ideas...

Indica mecanismos de programación funcional útiles en programación orientada a objetos.

Aplicar **programación funcional** en lenguajes orientados a objetos.

Aplicar **programación funcional** en lenguajes orientados a objetos.

¿Cómo?

Funciones

Funciones

En programación funcional, las funciones:

- Son, en sí mismas, un dato.
- Pueden pasarse como parámetro.

¿Qué construcción de Orientación a Objetos cumple con esos requisitos?

En programación funcional, las funciones:

- Son, en sí mismas, un dato.
- Pueden pasarse como parámetro.

¿Qué construcción de Orientación a Objetos cumple con esos requisitos?

Los **objetos**

Problema

1. Representa, mediante una estructura de un lenguaje orientado a objetos, una **función unaria**.
2. Define, en base a lo anterior, dos funciones sobre números reales:
   - Una función que eleve un número al cuadrado.
   - Una función que multiplique un número real por un valor concreto.
3. Genera una función (o método) que se pueda invocar con una función de reales sobre reales como parámetro. Deberá ser **polimórfica**.

En Java (herencia):

```
interface Function<Result, Parameter> {
   public Result eval(Parameter param);
};
```

```
class Sqr implements Function<Double, Double>  {
   public Double eval(Double d) { return d*d; }
};
```

```
class MulBy implements Function<Double, Double> {
   private Double factor;
   public MulBy(Double factor) { this.factor = factor; }
   public Double eval(Double d) { return factor*d; }
};
```

A partir de esta estructura de herencia, se pueden pasar una función como parámetro (polimorfismo)

```
class DoubleBox {
   private Double d;
   public DoubleBox(Double d) { this.d = d; }
   public void apply(Function<Double, Double> f)
   { this.d = f.eval(this.d);  }
};
```

```
class X {
   public static void main(String[] args) {
       DoubleBox db = new DoubleBox(2.0);
       db.apply(new Sqr());
       db.apply(new MulBy(3.0));
   }
};
```

En C++ (programacion genérica, aunque puede hacerse con herencia):

```
class Sqr {
public:
   double eval(double d) { return d*d; }
};
```

```
class MulBy {
   double factor;
public:
   MulBy(double f) : factor(f) { }
   double eval(double d) { return factor*d; }
};
```

Se pueden pasar como parámetro (polimorfismo).

```
class DoubleBox {
   double d;
   public: DoubleBox(double dd) : d(dd){ }
   template<typename F>
   void apply(const F& f)
   { d = f.eval(d);  }
};
```

```
int main(int argc, char** argv) {
   DoubleBox db(2.0);
   db.apply(Sqr());
   db.apply(MulBy(3.0));
}
```

En C++ se puede redefinir el **operador()**

```
class Sqr {
public:
   double operator()(double d) { return d*d; }
};
```

```
class DoubleBox {
   double d;
   public: DoubleBox(double dd) : d(dd){ }
   template<typename F>
   void apply(F f)
   { d = f(d);  }
};
```

En C++ se puede redefinir el **operador()**, y eso permite usar funciones de C++ también.

```
1 double halve(double d) {
2    return d/2.0;
3 }
```

```
1 int main(int argc, char** argv)
2 {
3    DoubleBox db(2.0);
4    db.apply(Sqr());
5    db.apply(halve);
6 }
```

Valoración: ¿es esto una buena idea en lenguajes orientados a objetos?

Valoración: ¿es esto una buena idea en lenguajes orientados a objetos?

Ideas...

¿Para qué podría usarse en un lenguaje orientado a objetos la posibilidad de pasar funciones como parámetros?

Valoración: ¿es esto una buena idea en lenguajes orientados a objetos?

Ideas...

¿Para qué podría usarse en un lenguaje orientado a objetos la posibilidad de pasar funciones como parámetros?

Fantásticas ideas. ¿Cómo no se le ha ocurrido todo esto a nadie antes?

En Java, functiones como parámetros.

```
1  public class AbsurdOrdering implements Comparator<Integer> {
2    public int compare(Integer o1, Integer o2) {
3      if (((o1%2==0)&&(o2%2==0)) || ((o1%2!=0)&&(o2%2!=0))) {
4         if (o1<o2) return -1;
5         else if (o1>o2) return 1;
6         else return 0;
7      } else return -1;
8    }
9  }
10 ...
11 List<Integer> l = new ArrayList<>();
12 l.add(5); l.add(2); l.add(4); l.add(1); l.add(-1);
13 Collections.sort(l, new AbsurdOrdering());
14 ...
```

En C++, funciones como parámetros.

```
1  #include <algorithm>    // std::sort
2  #include <vector>       // std::vector
3  bool absurdOrdering(int o1,int o2) {
4      if (((o1%2==0)&&(o2%2==0)) || ((o1%2!=0)&&(o2%2!=0)))
5         return (o1<o2);
6      else return false;
7  }
8  int main () {
9    std::vector<int> v{5,2,4,1,-1};
10   std::sort(v.begin(), v.end(), absurdOrdering);
11   return 0;
12 }
```

Funciones puras

Funciones puras

Una de las características de la programación funcional es que las funciones son **puras** (sin efectos laterales).

Pregunta

¿Qué es más práctico? ¿Las funciones puras o las funciones estándar de un lenguaje imperativo?

Funciones puras

Una de las características de la programación funcional es que las funciones son **puras** (sin efectos laterales).

Pregunta

¿Qué es más práctico? ¿Las funciones puras o las funciones estándar de un lenguaje imperativo?

Ideas...

Ventajas de las funciones puras sobre las funciones estándar de un lenguaje imperativo.

Funciones puras

¿Que ocurre aqui?

```
1  interface Function<Result, Parameter> {
2     public Result eval(Parameter param);
3  };
4  class Fun1 implements Function<Double, Double> {
5     private Double l;
6     public Fun1() { l=1; }
7     public Double eval(Double d) { l*=2.0; return l+d; }
8  };
9  class Fun2 implements Function<Double, Double> {
10    public Double eval(Double d) { d=0.0; return d+1.0; }
11 };
```

Funciones puras

¿Que ocurre aqui?

```
1  interface Function<Result, Parameter> {
2     public Result eval(Parameter param);
3  };
4  class Fun1 implements Function<Double, Double> {
5     private Double l;
6     public Fun1( ) { l=1; }
7     public Double eval(Double d) { l*=2.0; return l+d; }
8  };
9  class Fun2 implements Function<Double, Double> {
10    public Double eval(Double d) { d=0.0; return d+1.0; }
11 };
```

Funciones puras

```
1  class DoubleBox {
2     private Double d;
3     public DoubleBox(Double d) { this.d = d; }
4     public void apply(Function<Double, Double> f)
5     { this.d = f.eval(this.d);  }
6  };
7  class X {
8     public static void main(String[] args) {
9         Fun1 f1 = new Fun1(); Fun2 f2 = new Fun2();
10        DoubleBox db = new DoubleBox(2.0);
11        db.apply(f1); db.apply(f2);
12        db = new DoubleBox(2.0);
13        db.apply(f1); db.apply(f2);
14    }
15 };
```

Funciones puras

Una de las características de la programación funcional es que las funciones son **puras** (sin efectos laterales).

Pregunta

¿Qué elementos de los lenguajes orientados a objetos impiden que las funciones (y métodos) sean puras?

¿Cómo evitar los efectos laterales?

¿Cómo evitar los efectos laterales?

Java

Con mucho cuidado.

¿Cómo evitar los efectos laterales?

Java

Con mucho cuidado.

C++

Con cuidado y con **const**

C++: ¿tiene efectos laterales?

```
1  class Sqr {
2  public:
3     double operator()(double d) { return d*d; }
4  };
5  class DoubleBox {
6     double d;
7     public: DoubleBox(double dd) : d(dd){ }
8     template<typename F>
9     DoubleBox apply(F f)
10    { return DoubleBox(f(d));  }
11 };
12 int main( ) {
13    Sqr f;
14    DoubleBox a(2.0);          //a<-2
15    DoubleBox b = a.apply(f); //a<-2, b<-4
16    DoubleBox c = a.apply(f); //a<-2, b<-4, c<-4
17 }
```

C++: ¿tiene efectos laterales?

```
1  class Sqr {
2  public:
3     double operator()(double& d) { d*=d; return d; }
4  };
5  class DoubleBox {
6     double d;
7     public: DoubleBox(double dd) : d(dd){ }
8     template<typename F>
9     DoubleBox apply(F f)
10    { return DoubleBox(f(d));  }
11 };
12 int main( ) {
13    Sqr f;
14    DoubleBox a(2.0);          //a<-2
15    DoubleBox b = a.apply(f); //a<-4, b<-4
16    DoubleBox c = a.apply(f); //a<-16, b<-4, c<-16
17 }
```

C++: ¿tiene efectos laterales?

```
1  class Sqr {
2     double a;
3     public: Sqr() :a(0.0) {}
4     double operator()(double d) { a+=1.0; return d*d+a; }
5  };
6  class DoubleBox {
7     double d;
8     public: DoubleBox(double dd) : d(dd){ }
9     template<typename F>
10    DoubleBox apply( F& f )
11    { return DoubleBox(f(d));  }
12 };
13 int main() {
14    Sqr f;
15    DoubleBox a(2.0);          //a<-2
16    DoubleBox b = a.apply(f); //a<-2, b<-5
17    DoubleBox c = a.apply(f); //a<-2, b<-5, c<-6
18 }
```

Todas estas situaciones se pueden evitar con mucho cuidado...

Todas estas situaciones se pueden evitar con mucho cuidado...

...pero es mejor **forzar** errores de compilación...

Todas estas situaciones se pueden evitar con mucho cuidado...

...pero es mejor **forzar** errores de compilación...

... usando **const**.

Funciones puras

```
1  class Sqr {
2  public:
3  double operator()(double& d) { d*=d; return d; }
4  };
5  class DoubleBox {
6     double d;
7     public: DoubleBox(double dd) : d(dd){ }
8     template<typename F>
9     DoubleBox apply(F f)
10    { return DoubleBox(f(d));  }
11 };
12 int main( ) {
13    Sqr f;
14    DoubleBox a(2.0);          //a<-2
15    DoubleBox b = a.apply(f); //a<-4, b<-4
16    DoubleBox c = a.apply(f); //a<-16, b<-4, c<-16
17 }
```

Impedimos la modificación del atributo *d*:

```
1  class Sqr {
2     public:
3     double operator()(double& d) { d*=d; return d; }
4  };
5  class DoubleBox {
6     double d;
7     public: DoubleBox(double dd) : d(dd){ }
8     template<typename F>
9     DoubleBox apply(F f) const
10    { return DoubleBox(f(d));  }
11 };
12 int main( ) {
13    Sqr f;
14    DoubleBox a(2.0);          //a<-2
15    DoubleBox b = a.apply(f); //Error de compilacion^^I
16    DoubleBox c = a.apply(f); //--
17 }
```

Todo arreglado (en tiempo de compilación)

```
1  class Sqr {
2    public:
3    double operator()(const double& d) { return d*d; }
4  };
5  class DoubleBox {
6     double d;
7     public: DoubleBox(double dd) : d(dd){ }
8     template<typename F>
9     DoubleBox apply(F f) const
10    { return DoubleBox(f(d));  }
11 };
12 int main() {
13    Sqr f;
14    DoubleBox a(2.0);          //a<-2
15    DoubleBox b = a.apply(f); //a<-2, b<-4
16    DoubleBox c = a.apply(f); //a<-2, b<-4, c<-4
17 }
```

Intentamos evitar la copia de la función. ¿Qué pasa?

```
1  class Sqr {
2    double a;
3    public: Sqr() : a(0.0) { }
4    double operator()(const double& d) { a+=1.0; return d*d+a; }
5  };
6  class DoubleBox {
7     double d;
8     public: DoubleBox(double dd) : d(dd){ }
9     template<typename F>
10    DoubleBox apply(F& f) const
11    { return DoubleBox(f(d));  }
12 };
13 int main( ) {
14    Sqr f;
15    DoubleBox a(2.0);          //a<-2
16    DoubleBox b = a.apply(f); //a<-2, b<-5
17    DoubleBox c = a.apply(f); //a<-2, b<-5, c<-6
18 }
```

Forzamos a que la función sea inmutable.

```
1  class Sqr {
2    double a;
3    public: Sqr() : a(0.0) { }
4    double operator()(const double& d) { a+=1.0; return d*d+a; }
5  };
6  class DoubleBox {
7     double d;
8     public: DoubleBox(double dd) : d(dd){ }
9     template<typename F>
10    DoubleBox apply(const F& f) const
11    { return DoubleBox(f(d));  }
12  \\Error de compilacion ^^
13 };
14 int main() {
15    Sqr f;
16    DoubleBox a(2.0);          //a<-2
17    DoubleBox b = a.apply(f); //a<-2, b<-5
18    DoubleBox c = a.apply(f); //a<-2, b<-5, c<-6
19 }
```

Forzamos a que la aplicacion de la funcion sea const:

```
1  class Sqr {
2    double a;
3    public: Sqr() : a(0.0) { }
4    double operator()(const double& d) const { a+=1.0; return d*d+a; }
5   \\Error de compilacion                          ^^
6  };
7  class DoubleBox {
8     double d;
9     public: DoubleBox(double dd) : d(dd){ }
10    template<typename F>
11    DoubleBox apply(const F& f) const
12    { return DoubleBox(f(d));  }
13 };
14 int main() {
15    Sqr f;
16    DoubleBox a(2.0);          //a<-2
17    DoubleBox b = a.apply(f); //a<-2, b<-5
18    DoubleBox c = a.apply(f); //a<-2, b<-5, c<-6
19 }
```

Funciones puras

Todo vuelve a compilar, sin efectos laterales.

```
1  class Sqr {
2    double a;
3    public: Sqr() : a(0.0) { }
4    double operator()(const double& d) const { return d*d+a; }
5  };
6  class DoubleBox {
7     double d;
8     public: DoubleBox(double dd) : d(dd){ }
9     template<typename F>
10    DoubleBox apply(const F& f) const
11    { return DoubleBox(f(d));  }
12 };
13 int main() {
14    Sqr f;
15    DoubleBox a(2.0);          //a<-2
16    DoubleBox b = a.apply(f); //a<-2, b<-4
17    DoubleBox c = a.apply(f); //a<-2, b<-4, c<-4
18 }
```

En C++, la palabra clave **const** (o la más avanzada **constexpr**) permiten evitar ciertos efectos laterales.

Es importante ser consistente con el uso de dichas palabras clave desde el principio. De otra forma la cadena de errores de compilación es larga de resolver.

Ventajas:

- Verificabilidad: son fáciles de probar y validar (la ejecución da siempre el mismo resultado).
- Reusabilidad: no dependen de otras partes del sistema, su reutilización no tiene demasiadas consecuencias.
- Mantenibilidad: el efecto de una función pura está limitado a una mínima parte del sistema.
- Paralelización: se puede asegurar que las funciones puras son resistentes a múltiples hilos de ejecución (*thread-safety*).

¿Cómo conseguirlo?

- Todos los parámetros que sean referencias (o punteros) deberán llevar *const* delante.
- Todos los métodos deberán llevar asociada la palabra clave *const*.
- La única modificación de atributos ocurrirá en los constructores (y destructores).

Funciones puras

No siempre es la mejor solución el que todos los métodos sean puros.

Limitaciones:

- Modificación de atributos: para ciertos problemas puede ser más práctico.
- Eficiencia: los efectos laterales no necesitan reservar nueva memoria.
- Sistema operativo: es no funcional.

Funciones puras

No siempre es la mejor solución el que todos los métodos sean puros.

Limitaciones:

- Modificación de atributos: para ciertos problemas puede ser más práctico.
- Eficiencia: los efectos laterales no necesitan reservar nueva memoria.
- Sistema operativo: es no funcional.

En un punto intermedio está la virtud.

Problema

Diseña una función (el esqueleto) que calcule las raices de un polinomio de segundo grado.

Referencia + return

```
1  double roots2(double a,double b,double c,double& sol) {
2      ...
3      sol = s2;
4      return s1;
5  }
6
7  {     ...
8       double s1,s2;
9       s1 = roots2(1.0,-4.0,1.0, s2);
10      ...
11 }
```

Puntero + return

```
1  double roots2(double a,double b,double c,double* sol) {
2      ...
3      *sol = s2;
4      return s1;
5  }
6
7  { ...
8    double s1,s2;
9    s1 = roots2(1.0,-4.0,1.0, &s2);
10   ...
11 }
```

Dos referencias (simetría/uniformidad)

```
1  void roots2(double a,double b,double c,double& sol1,double& sol2) {
2      ...
3      sol1 = s1;
4      sol2 = s2;
5  }
6
7  { ...
8    double s1,s2;
9    roots2(1.0,-4.0,1.0, s1, s2);
10   ...
11 }
```

Dos punteros (simetría/uniformidad)

```
1  void roots2(double a,double b,double c,double* sol1,double* sol2) {
2      ...
3      *sol1 = s1;
4      *sol2 = s2;
5  }
6
7  { ...
8    double s1,s2;
9    roots2(1.0,-4.0,1.0, &s1, &s2);
10   ...
11 }
```

Dos punteros (simetría/unfiromidad)

```
1  void roots2(double a,double b,double c,double* sol1,double* sol2) {
2      ...
3      if (sol1) *sol1 = std::min(s1,s2);
4      if (sol2) *sol2 = std::max(s1,s2);
5  }
6
7  { ...
8    double mins;
9    roots2(1.0,-4.0,1.0, &mins, nullptr);
10   ...
11 }
```

Una lista

```
1  list<double> roots2(double a,double b,double c){
2      list<double> sol;
3      ...
4      sol.push_back(s1); sol.push_back(s2);
5      return sol;
6  }
7
8  { ...
9    list<double> l;
10   l = roots2(1.0,-4.0,1.0);
11   s1 = l.front(); s2 = l.back();
12   ...
13 }
```

Un vector

```
1  vector<double> roots2(double a,double b,double c) {
2      vector<double> sol;
3      ...
4      sol.push_back(s1); sol.push_back(s2);
5      return sol;
6  }
7
8  { ...
9    vector<double> l;
10   l = roots2(1.0,-4.0,1.0);
11   s1 = l[0]; s2 = l[1];
12   ...
13 }
```

Una tupla

```
std::tuple<double,double> roots2(double a,double b,double c) {
    ...
    return std::make_tuple(s1, s2);
}

{ ...
  auto theRoots = roots2(1.0,-4.0,1.0);
  s1 = std::get<0>(theRoots);
  s2 = std::get<1>(theRoots);
  ...
}
```

Una tupla

```
1  std::tuple<double,double> roots2(double a, double b, double c) {
2      ...
3      return std::make_tuple(s1, s2);
4  }
5
6  { ...
7    double s1,s2;
8    std::tie(s1,s2)          = roots2(1.0,-4.0,1.0);
9    std::tie(s1,std::ignore) = roots2(1.0,-4.0,1.0);
10   ...
11 }
```

Una tupla

Structured binding (C++ 17).

```
1  std::pair<double,double> roots2(double a, double b, double c) {
2      ...
3      return std::make_pair(s1, s2);
4  }
5
6  { ...
7    auto [s1, s2] = roots2(1.0,-4.0,1.0);
8    ...
9  }
```

Funciones anónimas

Funciones anónimas

Esto es un poco farragoso de escribir...

```
1  class Sqr {
2    public:
3    double operator()(double d) const { return d*d; }
4  };
5  class DoubleBox {
6     double d;
7     public DoubleBox(double dd) : d(dd){ }
8     template<typename F>
9     public DoubleBox getApply(const F& f) const
10    { return DoubleBox(f(d));  }
11 };
12 int main( ) {
13    Sqr f;
14    DoubleBox a(2.0);          //a<-2
15    DoubleBox b = a.apply(f); //a<-2, b<-4
16    DoubleBox c = a.apply(f); //a<-2, b<-4, c<-4
17 }
```

Funciones anónimas

Se puede hacer lo siguiente:

```
1  class DoubleBox {
2     double d;
3     public DoubleBox(double dd) : d(dd){ }
4     template<typename F>
5     public DoubleBox getApply(const F& f) const
6     { return DoubleBox(f(d));  }
7  };
8  int main( ) {
9     auto f = [] (double d) -> double { return d*d; }
10    DoubleBox a(2.0);          //a<-2
11    DoubleBox b = a.apply(f); //a<-2, b<-4
12    DoubleBox c = a.apply(f); //a<-2, b<-4, c<-4
13 }
```

Funciones anónimas

Incluso omitiendo el tipo de dato devuelto (si se puede deducir)

```
1  class DoubleBox {
2     double d;
3     public DoubleBox(double dd) : d(dd){ }
4     template<typename F>
5     public DoubleBox getApply(const F& f) const
6     { return DoubleBox(f(d));  }
7  };
8  int main( ) {
9     auto f = [] (double d) { return d*d; }
10    DoubleBox a(2.0);          //a<-2
11    DoubleBox b = a.apply(f); //a<-2, b<-4
12    DoubleBox c = a.apply(f); //a<-2, b<-4, c<-4
13 }
```

Funciones anónimas

Se puede pasar como parámetro:

```
1  class DoubleBox {
2     double d;
3     public DoubleBox(double dd) : d(dd){ }
4     template<typename F>
5     public DoubleBox getApply(const F& f) const
6     { return DoubleBox(f(d));  }
7  };
8  int main( ) {
9     DoubleBox a(2.0);          //a<-2
10    DoubleBox b = a.apply( [] (double d) { return d*d; });
11    DoubleBox c = a.apply( [] (double d) { return d*d; });
12    //a<-2, b<-4, c<-4
13 }
```

Funciones anónimas

¿Y qué pasa con esto?

```
1  class Add {
2    double n;
3  public: Add(double nn) : n(nn) { }
4    double operator()(double d) const { return d+n; }
5  };
6  class DoubleBox {
7     double d;
8     public DoubleBox(double dd) : d(dd){ }
9     template<typename F>
10    public DoubleBox getApply(const F& f) const
11    { return DoubleBox(f(d));  }
12 };
13 int main( ) {
14    DoubleBox a(2.0);          //a<-2
15    DoubleBox b = a.apply(Add(2.0)); //a<-2, b<-4
16    DoubleBox c = a.apply(Add(6.0)); //a<-2, b<-4, c<-8
17 }
```

Funciones anónimas

Se pueden hacer funciones anónimas diferentes...

```
1  class DoubleBox {
2     double d;
3     public DoubleBox(double dd) : d(dd){ }
4     template<typename F>
5     public DoubleBox getApply(const F& f) const
6     { return DoubleBox(f(d));  }
7  };
8  int main( ) {
9     DoubleBox a(2.0);          //a<-2
10    DoubleBox b = a.apply( [] (double d) { return d+2.0; } );
11    DoubleBox c = a.apply( [] (double d) { return d+6.0; } );
12    //a<-2, b<-4, c<-8
13 }
```

Funciones anónimas

... o se pueden **capturar** variables (por copia):

```
1  class DoubleBox {
2     double d;
3     public DoubleBox(double dd) : d(dd){ }
4     template<typename F>
5     public DoubleBox getApply(const F& f) const
6     { return DoubleBox(f(d));  }
7  };
8  int main( ) {
9     DoubleBox a(2.0);          //a<-2
10    double n = 2.0;
11    DoubleBox b = a.apply( [n] (double d) { return d+n; }); //a<-2, b<-4
12    n=6.0;
13    DoubleBox c = a.apply( [n] (double d) { return d+n; }); //a<-2, b<-4, c<-8
14 }
```

Funciones anónimas

... o se pueden **capturar** variables (por referencia):

```
1  class DoubleBox {
2     double d;
3     public DoubleBox(double dd) : d(dd){ }
4     template<typename F>
5     public DoubleBox getApply(const F& f) const
6     { return DoubleBox(f(d));  }
7  };
8  int main( ) {
9     DoubleBox a(2.0);          //a<-2
10    double n = 2.0;
11    DoubleBox b = a.apply( [&n] (double d) { return d+n; }); //a<-2, b<-4
12    n=6.0;
13    DoubleBox c = a.apply( [&n] (double d) { return d+n; }); //a<-2, b<-4, c<-8
14 }
```

Funciones anónimas

... o se pueden **capturar** variables (todas por copia):

```
1  class DoubleBox {
2     double d;
3     public DoubleBox(double dd) : d(dd){ }
4     template<typename F>
5     public DoubleBox getApply(const F& f) const
6     { return DoubleBox(f(d));  }
7  };
8  int main( ) {
9     DoubleBox a(2.0);          //a<-2
10    double n = 2.0;
11    DoubleBox b = a.apply( [=] (double d) { return d+n; }); //a<-2, b<-4
12    n=6.0;
13    DoubleBox c = a.apply( [=] (double d) { return d+n; }); //a<-2, b<-4, c<-8
14 }
```

Funciones anónimas

... o se pueden **capturar** variables (todas por referencia):

```
1  class DoubleBox {
2     double d;
3     public DoubleBox(double dd) : d(dd){ }
4     template<typename F>
5     public DoubleBox getApply(const F& f) const
6     { return DoubleBox(f(d));  }
7  };
8  int main( ) {
9     DoubleBox a(2.0);          //a<-2
10    double n = 2.0;
11    DoubleBox b = a.apply( [&] (double d) { return d+n; }); //a<-2, b<-4
12    n=6.0;
13    DoubleBox c = a.apply( [&] (double d) { return d+n; }); //a<-2, b<-4, c<-8
14 }
```

Funciones anónimas

C++ incluye la idea de los *functors* a través de la cabecera **functional**

```
1  #include <functional>
2  #include <iostream>
3
4  using namespace std;
5
6  int main( ) {
7     function<double(double)> dup  = [] (double d) { return d+d; };
8     auto                     sqr  = [] (double d) { return d*d; };
9
10    cout << dup(3.0) << " " << sqr(3.0) << endl;
11 }
```

C++ incluye **binding** (enlazado) que permite aplicación parcial.

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

int sum(int a,int b) {return a+b; }
int main( )
{
  function<int(int)> inc = bind(sum,1,_1);
  cout << inc(3) << endl;
  return 0;
}
```

C++ incluye **binding** (enlazado) que permite aplicación parcial.

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

int sum3(int a,int b,int c) { return a+b+c; }
int main( )
{
  function<int(int,int)> sum2 = bind(sum3,_2,7,_1);
  cout << sum2(3,4) << endl;
  return 0;
}
```

Orden superior

Orden superior

Desde C++11, existen múltiples funciones de orden superior (que reciben funciones como parámetros).

Ventajas:

- Aumentan la legibilidad del código.
- Son genéricos con respecto a estructuras de datos y funciones.
- Están todo lo optimizados que pueden estar.

Ejemplo:

```
#include <iostream>     // std::cout
#include <algorithm>    // std::all_of
#include <array>        // std::array

int main () {
  std::array<int,8> foo = {3,5,7,11,13,17,19,23};
  if (std::all_of(begin(foo), end(foo),
                       [](int i){return (i%2)!=0;}) )
      std::cout << "All the elements are odd numbers.\n";
  return 0;
}
```

Ejemplo:

```
1  #include <iostream>     // std::cout
2  #include <algorithm>    // std::any_of
3  #include <array>        // std::array
4
5  int main () {
6    std::array<int,7> foo = {0,1,-1,3,-3,5,-5};
7    if (std::any_of(begin(foo), end(foo),
8                    [](int i){return i<0;}) )
9      std::cout << "Negative elements in array.\n";
10   return 0;
11 }
```

Ejemplo:

```
1  #include <iostream>     // std::cout
2  #include <algorithm>    // std::find_if
3  #include <vector>       // std::vector
4  int main () {
5    std::vector<int> myvector = {10,25,40,55};
6    auto it = std::find_if(begin(myvector),end(myvector),
7                           [] (int i) { return ((i%2)==1);});
8    std::cout << "First odd value is " << *it << '\n';
9    return 0;
10 }
```

Orden superior

```
1  #include <iostream>     // std::cout
2  #include <algorithm>    // std::transform
3  #include <vector>       // std::vector
4
5  int main () {
6    std::vector<int> foo;
7    std::vector<int> bar;
8    for (int i=1; i<6; i++) foo.push_back(i*10);
9    // foo: 10 20 30 40 50
10   bar.resize(foo.size()); // OJO!!!
11   std::transform(begin(foo),end(foo),begin(bar),
12                  [] (int i) { return i+1; });
13   // bar: 11 21 31 41 51
14   std::transform(begin(foo),end(foo),begin(bar),begin(foo),
15                  [] (int i, int j) { return i+j; });
16   // foo: 21 41 61 81 101
17 }
```

Orden superior

```
1  #include <iostream>     // std::cout
2  #include <algorithm>    // std::transform
3  #include <vector>       // std::vector
4
5  int main () {
6    std::vector<int> foo;
7    std::vector<int> bar;
8    for (int i=1; i<6; i++) foo.push_back(i*10);
9    // foo: 10 20 30 40 50
10   // NO NEED TO bar.resize(...)
11   std::transform(begin(foo),end(foo),std::back_inserter(bar),
12                  [] (int i) { return i+1; });
13   // bar: 11 21 31 41 51
14 }
```

En C++20 aparece el módulo **ranges**, que simplifica el uso de funciones de orden superior:

```
1  vector<int> ints = {0,1,2,3,4,5};
2  vector<int> evens;
3
4  transform(begin(ints),end(ints),back_inserter(evens1),
5            [](int i) { return (i%2)==0; } );
6
7  for (auto x : evens) cout << x << " ";
```

En C++20 aparece el módulo **ranges**, que simplifica el uso de funciones de orden superior:

```
  vector<int> ints = {0,1,2,3,4,5};
  vector<int> evens;

  ranges::transform(ints,back_inserter(evens),
            [](int i) { return (i%2)==0; } );

  for (auto x : evens) cout << x << " ";

template<typename T>
concept range = requires( T& t ) {
  ranges::begin(t);
  ranges::end  (t);
};
```

En C++20 aparece el módulo **ranges**, que simplifica el uso de funciones de orden superior aún más usando **views**:

```
1  vector<int> ints = {0,1,2,3,4,5};
2
3
4  auto evens = views::transform(ints,
5            [](int i) { return (i%2)==0; } );
6
7  for (auto x : evens) cout << x << " ";
```

Nota MUY importante: las vistas utilizan **evaluación perezosa**.

En C++20 aparece el módulo **ranges**, que simplifica el uso de funciones de orden superior aún más usando **views**:

```
#include <ranges>

int main() {
  auto const ints = {0,1,2,3,4,5};
  auto is_even    = [](int i) { return 0 == i % 2; };
  auto square     = [](int i) { return i * i; };

  // Sintaxis funcional
  for (int i : views::transform( views::filter(ints, is_even) , square) )
      std::cout << i << ' ';
}
```

En C++20 aparece el módulo **ranges**, que simplifica el uso de funciones de orden superior aún más usando **views** y definiendo nuevos operadores:

```
#include <ranges>

int main() {
  auto const ints = {0,1,2,3,4,5};
  auto is_even    = [](int i) { return 0 == i % 2; };
  auto square     = [](int i) { return i * i; };

  // Sintaxis "pipe"
  for (int i : ints | views::filter(is_even) | views::transform(square))
      std::cout << i << ' ';
}
```

- Leer documentación (hay gran variedad de funciones de orden superior)
- Están implementadas mediante programación genérica.

Programación funcional y orientación a objetos

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

Escuela de Ingeniería y Arquitectura Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas Universidad Zaragoza

2023-01-20