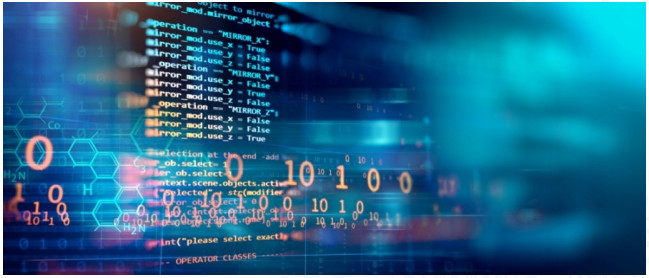
Programación Genérica

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

Introducción

Problema

Una serie de sensores envían regularmente datos a un computador.
Los datos de cada sensor son procesados por un **almacén**, una clase que ofrece al menos dos métodos:

- uno que recibe el dato:

```
void push(???? v)
```

- otro que devuleve el valor guardado:

```
???? value() const
```

La dificultad radica en los sensores pueden enviar datos de distintos tipos:

- Los sensores analógicos trabajan con **reales** (pudiendo ser de simple o doble precisión).
- Los sensores de tipo contador mandan números **enteros**.
- Ciertos sensores eléctricos y magéticos envían números **complejos**.
- Los sensores espectrales envían **vectores** de elementos en punto flotante.

Problema

Diseña una clase `StoreLast` que simplemente

almacene el último dato

de los datos que va enviando un sensor.

Complicándolo un poco. . .

Problema

Diseña una clase `StoreMax` que

guarde el máximo

de todos los datos que va enviando un sensor.

Problema

Diseña una clase `StoreMin` que

guarde el mínimo

de todos los datos que va enviando un sensor.

Complicándolo un poco más. . .

Problema

Diseña una clase `StoreSum` que

guarde la suma

de todos los datos que va enviando un sensor,
pasándole un valor inicial (de cero) en el constructor.

Problema

Diseña una clase `StoreAvg` que

devuelva el valor medio

de todos los datos que va enviando un sensor,
pasándole un valor inicial (de cero) en el constructor.

La Programación Genérica es un estilo de programación que permite escribir estructuras de datos y algoritmos utilizando tipos de datos que estan todavía sin definir, manteniendo la seguridad del control de tipos de un lenguaje.

Implementa el denominado *Polimorfismo Paramétrico*:

- permite escribir código que opera de la misma forma con cualquier tipo de datos, sin que estos tipos tengan que estar relacionados por la herencia.
- permite escribir código reutilizable independientemente de los datos sobre los que se va a aplicar.

Objetivo: un contenedor Java que permita almacenar cualquier objeto.

```
1  class Box
2  {
3    Object  o;
4
5    void save(Object _o)
6    { o = _o; }
7
8    Object restore()
9    { return o; }
10 }
```

```
1  void main(String[] args)
2  {
3    Box b = new Box();
4
5    b.save(new String("Hallo"));
6
7    Integer i = (Integer)b.restore();
8    System.out.println(i);
9  }
```

¿ Cuál es el problema ?

Objetivo: un contenedor Java que permita almacenar cualquier objeto.

```
1  class Box
2  {
3    Object  o;
4
5    void save(Object _o)
6    { o = _o; }
7
8    Object restore()
9    { return o; }
10 }
```

```
1  void main(String[] args)
2  {
3    Box b = new Box();
4
5    b.save(new String("Hallo"));
6
7    Integer i = (Integer)b.restore();
8    System.out.println(i);
9  }
```

La información de tipo se pierde.

La comprobación de tipos ocurre en tiempo de ejecución, y la conversión incorrecta genera una excepción.

Objetivo: un contenedor Java que permita almacenar un par de objetos del mismo tipo.

```
1  class Pair
2  {
3    // Los dos objetos
4    // deben ser del mismo tipo
5    public Object fst,snd;
6
7    Pair(Object f,Object s)
8    {
9      fst = f; snd = s;
10   }
11 }
```

```
1  void main(String[] args)
2  {
3    Pair good = new Pair("Hola","Adios");
4
5    Pair bad  = new Pair("Hola",3.14);
6  }
```

¿ Cuál es el problema ?

Objetivo: un contenedor Java que permita almacenar un par de objetos del mismo tipo.

```java
class Pair
{
  // Los dos objetos
  // deben ser del mismo tipo
  public Object fst,snd;

  Pair(Object f,Object s)
  {
    fst = f; snd = s;
  }
}
```

```java
void main(String[] args)
{
  Pair good = new Pair("Hola","Adios");

  Pair bad  = new Pair("Hola",3.14);
}
```

La información de tipo no se controla, los objetos son de distinto tipo.

Mientras tanto, en C++. . .

```
1  class Box
2  {
3    void* o;
4
5    void save(void* _o)
6    { o = _o; }
7
8    void* restore() const
9    { return o; }
10 };
```

```
1  int main()
2  {
3    Box box;
4
5    box.save(new string("Hallo"));
6
7    int* i = (int*)(box.restore());
8    cout << (*i) << endl;
9  }
```

¿ Cuál es el problema ?

Mientras tanto, en C++. . .

```
1  class Box
2  {
3    void* o;
4
5    void save(void* _o)
6    { o = _o; }
7
8    void* restore() const
9    { return o; }
10 };
```

```
1  int main()
2  {
3    Box box;
4
5    box.save(new string("Hallo"));
6
7    int* i = (int*)(box.restore());
8    cout << (*i) << endl;
9  }
```

La información de tipo se pierde.

No hay comprobación de tipos, se reinterpreta el tipo.

Mientras tanto, en C++. . .

```
1  class Pair
2  {
3    // Los dos objetos
4    // deben ser del mismo tipo
5    void *fst,*snd;
6
7    Pair(void* f,void* s)
8    {
9      fst = f; snd = s;
10   }
11 };
```

```
1  int main()
2  {
3    Pair good( new string("Hola"),
4               new string("Adios"));
5
6    Pair bad( new string("Hola"),
7              new float(3.14));
8  }
```

¿ Cuál es el problema ?

Mientras tanto, en C++. . .

```
1  class Pair
2  {
3    // Los dos objetos
4    // deben ser del mismo tipo
5    void *fst,*snd;
6
7    Pair(void* f,void* s)
8    {
9      fst = f; snd = s;
10   }
11 };
```

```
1  int main()
2  {
3    Pair good( new string("Hola"),
4               new string("Adios"));
5
6    Pair bad( new string("Hola"),
7              new float(3.14));
8  }
```

La información de tipo no se controla, los objetos son de distinto tipo.

Recordando la **sobrecarga**...

```
1  int max(int a, int b)
2  {
3     if (a>b)
4        return a;
5     else
6        return b;
7  }
```

```
1  float max(float a, float b)
2  {
3     if (a>b)
4        return a;
5     else
6        return b;
7  }
```

¿Y si lo necesito para otro tipo de datos?

La programación genérica tiene dos fases:

- **Especificación:** Se implementa la clase genérica (o función / método genérico) indicando una serie de elementos parámetro (tipos de datos, generalmente) desconocidos a priori.
- **Instanciación:** Se utiliza la clase genérica (o función / método genérico) especificando valores concretos para elementos parámetro (tipos de datos específicos). Si se puede, se deducen automáticamente.

Clases – Métodos

Una clase genérica se **especifica** mediante un tipo desconocido (parámetro) en la definición de la clase, que se concretará en su instanciación:

C++

```cpp
template<typename T>
class Generator {
  T state;
  T gimmemore() { ... }
};
```

Java

```java

class Generator<T> {
  T state;
  T gimmemore() { ... }
}
```

También pueden dejarse **varios** tipos desconocidos como parámetro.

C++

```
1 template<typename R,
2          typename P>
3 class Converter {
4   R convert(P src) { ... }
5 };
```

Java

```
1
2
3 class Converter<R,P> {
4   R convert(P src) { ... }
5 }
```

Una clase genérica se utiliza (*instancia*) concretando el tipo.

C++

```cpp
// Se concreta el tipo
Generator<float> fg;


Converter<string,float> cfs;
```

Java

```java
// Se concreta el tipo
Generator<Float> fg
 = new Generator<Float>();

Converter<String,Float> cfs
 = new Converter<String,Float>();
```

En ocasiones se pueden hacer ciertas deducciones (inferencias) en la instanciación:

C++

```
1 //No se infiere el tipo
2 //Generator<float>* fg
3 //  = new Generator<>();
4
5 //Converter<string,float>* cfs
6 // = new Converter<>();
```

Java

```
1 //Se puede inferir el tipo
2 Generator<Float> fg
3  = new Generator<>();
4
5 Converter<String,Float> cfs
6  = new Converter<>();
```

En Java los genéricos sólo se pueden instanciar para clases, no para tipos fundamentales.
Esto es incorrecto:

```
1  Generator<float> fg = new Generator<float>();
```

Solución – *wrapper classes*:

```
1  class Float {
2    float value;
3  }
4  Generator<Float> fg = new Generator<Float>();
```

La clase instanciada dispone de todos los atributos definidos en la clase genérica, reemplazando los tipos parámetro.

```
1  float  f = fg.gimmemore();
2  string s = cfs.convert(3.14f);
3
4  // vv Error de compilacion
5  string s = fg.gimmemore();
6  int    i = cfs.convert("");
7
8  // vv Inferencia (C++11)
9  auto f = fg.gimmemore();
10 auto s = cfs.convert(3.14f);
```

```
1  Float  f = fg.gimmemore();
2  String s = cfs.convert(3.14f);
3
4  // vv Error de compilacion
5  String  s = fg.gimmemore();
6  Integer i = cfs.convert("");
```

Java

```java
class Box<T>
{
  T o;

  void save(T _o)
  { o = _o; }

  T restore()
  { return o; }
}
```

```java
class main
{
  public static void main(String[] args) {
    Box<String> b = new Box<String>();
    b.save("Hallo");

    String  s = b.restore();
    // Esto da un error en compilacion
    Integer i = b.restore();
  }
}
```

Ahora sí se verifica la adecuada correspondencia de tipos en tiempo de compilación.

C++

```
1  template<typename T>
2  class Box
3  {
4    T* o;
5
6    void save(T* _o)
7    { o = _o;   }
8
9    T* restore() const
10   { return o; }
11 };
```

```
1  #include "box.h"
2  #include <string>
3  #include <iostream>
4
5  int main()
6  {
7      Box<string> box;
8      box.save(new string("Hallo"));
9
10     cout << (*(box.restore())) << endl;
11 }
```

Ahora sí se verifica la adecuada correspondencia de tipos en tiempo de compilación.

C++

```
1  template<typename T>
2  class Box
3  {
4    T o;
5
6    void save(const T& _o)
7    { o = _o;   }
8
9    const T& restore() const
10   { return o; }
11 };
```

```
1  #include "box.h"
2  #include <string>
3  #include <iostream>
4
5  int main()
6  {
7      Box<string> box;
8      box.save("Hallo");
9
10     cout << box.restore() << endl;
11 }
```

Ahora sí se verifica la adecuada correspondencia de tipos en tiempo de compilación.

El contenedor almacena un objeto completo.

Java

```java
class Pair<T>
{
  // Los dos objetos
  // deben ser del mismo tipo
  public T fst,snd;

  Pair(T f,T s)
  {
    fst = f; snd = s;
  }
}
```

```java
void main(String[] args)
{
  Pair<String> good  = new Pair<>("Hola","Adios");

  // Esto da un error en compilacion
  Pair<String> bad   = new Pair<>("Hola",3.14);

  // Mucho cuidado con esto....
  Pair<String> weird = new Pair("Hola","Adios");
}
```

Ahora sí se verifica la adecuada correspondencia de tipos en tiempo de compilación.
Los tipos se deducen automáticamente.

C++

```cpp
template<typename T>
class Pair
{
  // Los dos objetos
  // deben ser del mismo tipo
  T fst,snd;

  Pair(const T& f,const T& s)
  {
    fst = f; snd = s;
  }
};
```

```cpp
int main()
{
  Pair good(string("Hola"),string("Adios"));

  // Esto da un error en compilacion
  Pair bad(string("Hola"),3.14);
}
```

Ahora sí se verifica la adecuada correspondencia de tipos en tiempo de compilación.

Los tipos se deducen automáticamente.

Un método/función puede hacerse genérico.

```
1  template<typename T>
2  T max(T t1, T t2) {
3    if (t1>t2) return t1;
4    else return t2;
5  }
6
7  int main() {
8    cout << max<int>(1,3) << endl;
9    cout << max<double>(8.5,3.2) << endl;
10   return 0;
11 }
```

Se pueden deducir los tipos de datos implicados.

```
1  template<typename T>
2  T max(T t1, T t2) {
3    if (t1>t2) return t1;
4    else return t2;
5  }
6
7  int main() {
8    cout << max(1,3) << endl;
9    cout << max(8.5,3.2) << endl;
10   return 0;
11 }
```

¿Qué pasa con el siguiente código?

```
1  template<typename T>
2  T max(T t1, T t2) {
3    if (t1>t2) return t1;
4    else return t2;
5  }
6
7  int main() {
8    cout << max(1,3.5) << endl;
9    return 0;
10 }
```

Ligero cambio en las reglas. . .

De cara a la resolución de ambigüedades, no es lo mismo esto

```
float max(float a,float b) { ... }
int   max(int   a,int   b) { ... }
```

que esto

```
template<typename T>
T max(T t1, T t2) { ... }
```

Retomaremos el tema más adelante. . .

El siguiente código se repite para múltiples tipos de datos.
¿Cómo podemos escribir el código sólo una vez?

```
1  class NonZero {
2    static int count(Integer[] v) {
3      int c = 0;
4      for (Integer e : v) if (e.doubleValue()>0) c++;
5      return c;
6    }
7    static int count(Float[] v) {
8      int c = 0;
9      for (Float e : v) if (e.doubleValue()>0) c++;
10     return c;
11   }
12 }
```

¿Cómo podemos escribir el código sólo una vez?

```
1  class NonZero {
2    static <T> int count(T[] v) {
3      int c = 0;
4      for (T e : v) if (e.doubleValue()>0) c++;
5      return c;
6    }
7  }
```

Esta clase (Java) tiene un problema. ¿Cuál?

¿Cómo podemos escribir el código sólo una vez?

```
1  class NonZero {
2    static <T> int count(T[] v) {
3      int c = 0;
4      for (T e : v) if (e.doubleValue()>0) c++;
5      return c;
6    }
7  }
```

Esta clase (Java) tiene un problema. ¿Cuál?

`e.doubleValue()`

¿Cómo podemos escribir el código sólo una vez?

```
1  class NonZero {
2    static <T> int count(T[] v) {
3      int c = 0;
4      for (T e : v) if (e.doubleValue()>0) c++;
5      return c;
6    }
7  }
```

Esta clase (Java) tiene un problema. ¿Cuál?

`e.doubleValue()`

Utilizamos una operación sobre un tipo sin saber si el tipo la soporta.

En Java, es necesario acotar de alguna forma el rango de tipos sobre los que puede actuar el genérico.
Ese proceso se realiza mediante condiciones de herencia:

```
1  class NonZero {
2    static <T extends Number> int count(T[] v) {
3      int c = 0;
4      for (T e : v) if (e.doubleValue()>0) c++;
5      return c;
6    }
7  }
```

Podemos imponer varias restricciones al tipo aplicable en el genérico.
Dado que Java no soporta herencia múltiple, las condiciones se interpretan como herencia de una clase y obligación de implementar múltiples interfaces:

```
1 class A { /* ... */ }
2 interface B { /* ... */ }
3 interface C { /* ... */ }
4
5 class D<T extends A & B & C> { /* ... */ }
```

Mientras tanto, en C++. . .

```
1  template<typename T>
2  bool before(const T& a,
3              const T& b)
4  {
5    return (a<b);
6  }
```

```
1  #include "before.h"
2  #include <iostream>
3
4  int main()
5  {
6    cout << before(4,3) << endl;
7  }
```

Mientras tanto, en C++. . .

```cpp
template<typename T>
bool before(const T& a,
            const T& b)
{
  return (a<b);
}
```

```cpp
class Person
{
  int age;
};
```

```cpp
#include "before.h"
#include "person.h"
#include <iostream>

int main()
{
  Person one, other;

  cout << before(one,other) << endl;
}
```

¿ Compila ? ¿ Funciona correctamente ?

Mientras tanto, en C++. . .

```
template<typename T>
bool before(const T& a,
            const T& b)
{
  return (a<b);
}
```

```
class Person
{
  int age;
};
```

```
#include "before.h"
#include "person.h"
#include <iostream>

int main()
{
  Person one, other;

  cout << before(one,other) << endl;
}
```

Instancia para la clase `Person` $\Rightarrow$ error de compilación en `'<'`.

Mientras tanto, en C++. . .

```
1  class Person
2  {
3    int age;
4
5    bool operator<(const Person& that) const
6    {
7       return age < that.age;
8    }
9  };
```

Solución:
Añadir a la clase la funcionalidad necesaria.

Implementación

C++

Al instanciar duplica el código genérico, sustituyendo el tipo parámetro.
Compila el código generado.

Java

Compila el código genérico sustituyendo el tipo por `Object`.
Al instanciar, aplica un *casting* de `Object` al tipo parámetro.

Implementación – C++

Al instanciar duplica el código genérico, sustituyendo el tipo parámetro.
Compila el código generado.

Consecuencia

Los errores se detectan al compilar el código que **instancia el código genérico**:
al **usar** la clase/método genérico para un tipo concreto.

Implementación – Java

Compila el código genérico sustituyendo el tipo por `Object`.
Al instanciar, aplica un *casting* de `Object` al tipo parámetro.

Consecuencia

Los errores se detectan al compilar el código que **define el código genérico**:
al **implementar** la clase/método genérico.

Implementación – C++

Al instanciar duplica el código genérico, sustituyendo el tipo parámetro.

Compila el código generado.

Consecuencia

Los errores se detectan al compilar el código que **instancia el código genérico**:
al **usar** la clase/método genérico para un tipo concreto.

Implicaciones:

El programador debe saber qué requisitos debe cumplir un tipo de datos para poder usar un genérico con él (C++ futuro: *concepts*).

Implementación – Java

Compila el código genérico sustituyendo el tipo por `Object`.

Al instanciar, aplica un *casting* de `Object` al tipo parámetro.

Consecuencia:

Los errores se detectan al compilar el código que **define el código genérico**:
al **implementar** la clase/método genérico.

Implicaciones:

Al desarrollar el código genérico hay que imponer restricciones al tipo, para saber el conjunto de operaciones que podemos realizar con él.

Implementación

C++	Java
Al instanciar duplica el código genérico, sustituyendo el tipo parámetro.	Compila el código genérico sustituyendo el tipo por `Object`.
Compila el código generado.	Al instanciar, aplica un *casting* de `Object` al tipo parámetro.

Ninguno de los dos lenguajes implementa realmente el **polimorfismo paramétrico**.
(más sobre esto en próximas entregas. . .)

Genéricos y Sobrecarga

Recordemos que en C++ existe la sobrecarga como método para simular el polimorfismo:

```
1  int    big(int a,int b)
2  {
3    return (a>b ? a : b);
4  }
5
6  double big(double a,double b)
7  {
8    return (a>b ? a : b);
9  }
```

```
1  int main()
2  {
3    cout << big(1,2)      << endl;
4    cout << big(1.0,2.0)  << endl;
5
6    return 0;
7  }
```

Si ninguna de las versiones encaja exactamente, se pueden hacer ciertas conversiones implícitas de tipos:

```
int    big(int a,int b)
{
  return (a>b ? a : b);
}

double big(double a,double b)
{
  return (a>b ? a : b);
}
```

```
int main()
{
  // OK: float -> double
  cout << big(1.0f,2.0) << endl;
  // Error: no int -> double
  cout << big(1,2.0)    << endl;

  return 0;
}
```

Un genérico puede verse como una factoría de generación automática de versiones sobrecargadas de una función:

```
1  int    big(int a,int b)
2  {
3    return (a>b ? a : b);
4  }
5
6  double big(double a,double b)
7  {
8    return (a>b ? a : b);
9  }
```

```
1  int main()
2  {
3    cout << big(1,2)     << endl;
4    cout << big(1.0,2.0) << endl;
5
6    return 0;
7  }
```

Un genérico puede verse como una factoría de generación automática de versiones sobrecargadas de una función:

```
template<typename T>
T big(T a,T b)
{
  return (a>b ? a : b);
}
```

```
int main()
{
  cout << big<int>(1,2)       << endl;
  cout << big<double>(1.0,2.0) << endl;

  return 0;
}
```

Un genérico puede verse como una factoría de generación automática de versiones sobrecargadas de una función:

```
1  template<typename T>
2  T big(T a,T b)
3  {
4    return (a>b ? a : b);
5  }
```

```
1  int main()
2  {
3    cout << big(1,2)     << endl;
4    cout << big(1.0,2.0) << endl;
5
6    return 0;
7  }
```

. . .incluso de forma más cómoda con la deducción automática de tipos del genérico. . .

Un genérico puede verse como una factoría de generación automática de versiones sobrecargadas de una función:

```
1  template<typename T>
2  T big(T a,T b)
3  {
4    return (a>b ? a : b);
5  }
```

```
1  int main()
2  {
3    cout << big(1,2)      << endl;
4    cout << big(1.0,2.0)  << endl;
5    // Error !!
6    cout << big(1.0f,2.0) << endl;
7    // Error !!
8    cout << big(1,2.0)    << endl;
9
10   return 0;
11 }
```

. . .que añade un pequeño problema. . .
(SFINAE)

Un genérico puede verse como una factoría de generación automática de versiones sobrecargadas de una función:

```
1  template<typename T>
2  T big(T a,T b)
3  {
4    return (a>b ? a : b);
5  }
```

```
1  int main()
2  {
3    cout << big(1,2)              << endl;
4    cout << big(1.0,2.0)          << endl;
5    // OK
6    cout << big<double>(1.0f,2.0) << endl;
7    // OK
8    cout << big<double>(1,2.0)    << endl;
9
10   return 0;
11 }
```

. . .que puede solucionarse fácilmente, junto con otros.

¿Qué ocurre cuando combinas funciones genéricas y sobrecarga?

```
1  class Complex {
2    float real,imag;
3    Complex(float r, float i = 0.0)
4      : real(r), imag(i) { }
5  };
6
7  template<typename T>
8  T     module(const T&       x) {
9    return (x>=0 ? x : -x);
10 }
11
12 float module(const Complex& z) {
13   return std::hypot(z.real,z.imag);
14 }
```

```
1  int main() {
2     Complex z(3.0f, 4.0f);
3     float   r(2.0f);
4     cout << module(z) << endl;
5     cout << module(r) << endl;
6  }
```

Aparecen **ambigüedades**.

¿A qué función se está llamando en cada caso?

Resolución de ambigüedades/conflictos:

- Reglas de la sobrecarga: se elige a qué función (o método) se llama en base al número y tipo de parámetros.
  El compilador puede aplicar conversores o constructores de tipos.
- Reglas de la programación genérica: se deducen automáticamente los tipos instanciados y se sustituye.

```
1 module(z) : float module(const Complex& z)
2             T     module(const T& x) [with T = Complex]
3
4 module(r) : float module(const Complex& z) [with z = Complex(r)]
5             T     module(const T& x) [with T = float]
```

Resolución de ambigüedades/conflictos:

```
1  module(z) : float module(const Complex& z)
2              T     module(const T& x) [with T = Complex]
3
4  module(r) : float module(const Complex& z) [with z = Complex(r)]
5              T     module(const T& x) [with T = float]
```

¿ Cómo se resuelve esta ambigüedad ?

C++

Se elige la función (o método) priorizando los más específicos:

- menos conversiones
- menos tipos parámetro
- parámetros más precisos

Java

Reglas mucho más simples. . .
Menos posibilidades.
En Java se prefiere usar la herencia.

C++

Se prioriza sobrecarga y genéricos.

```cpp
class Complex {
   float real, imag;
};

string to_string(const Complex& c)
{
   return to_string(c.real)
        + " +i"
        + to_string(c.imag);
}

template<typename T>
void print(const T& t) {
   cout << to_string(t) << endl;
}
```

Java

Se prioriza la herencia.

```java
class Complex // extends Object
{
   Float real, imag;
   String toString( ) {
      return  real.toString( )
            + " +i "
            + imag.toString( );
   }
};

class Utils {
   void print(Object o) {
      System.out.println(o.toString( ));
   }
}
```

Las reglas combinadas de sobrecarga y programación genérica en funciones permiten definir una implementación genérica y **especializarla** para algún tipo de datos concreto.

```
1  template<typename T>
2  T module(const T& x) {
3     return (x>=0 ? x : -x);
4  }
```

```
1
2  float module(const Complex& c) {
3     return hypot(z.real,z.imag);
4  }
```

Esto es útil (muy útil) (extremadamente útil).
Es similar a la herencia pero sin obligar a relaciones entre clases.
Nos gustaría poder aplicarlo para clases.

En C++ se permite la **especialización de *templates***:
definir comportamientos específicos para ciertos tipos de datos.

```
1  template <typename T>
2  class Storage8 {
3    T data[8];
4
5    void set(int i, const T& v) {
6      data[i] = v;
7    }
8
9
10
11   const T& get(int i) const {
12     return data[i];
13   }
14 };
```

En C++ se permite la **especialización de *templates***:
definir comportamientos específicos para ciertos tipos de datos.

```
template <typename T>
class Storage8 {
  T data[8];

  void set(int i, const T& v) {
    data[i] = v;
  }




  const T& get(int i) const {
    return data[i];
  }
};
```

```
template <>
class Storage8<bool> {
  uint8_t data;

  void set(int i, bool v) {
    unsigned char mask = 1 << i;
    if (v) data |=  mask;
    else   data &= ~mask;
  }

  bool get(int i) {
    return (data & (1<<i)) != 0;
  }
};
```

Es un mecanismo del que se puede abusar.

Se pueden evaluar expresiones en **tiempo de compilación**:

template metaprogramming

```
1  template <unsigned int N> struct Factorial {
2    static const unsigned long value = N * Factorial<N-1>::value;
3  };
4
5  template <> struct Factorial<0> {
6    static const unsigned long value = 1;
7  };
8
9  int main() {
10   int many[Factorial<6>::value];
11   // ...
12 }
```

Como el uso de ese tipo de metaprogramación se fue de las manos se introdujo la palabra clave

`constexpr`

que permite evaluar expresiones en **tiempo de compilación**.

```
constexpr int factorial( int i )
{
  return (i == 0) ? 1 : i * factorial(i - 1);
}

int main() {
  int many[factorial(6)];
  // ...
}
```

```
class Data {
  int many[factorial(6)];
  // ...
}
```

Se utiliza para realizar identificación de tipos en tiempo de compilación: *inferencia de tipos*.

```
1   template<typename T> struct is_pointer
2   {  static constexpr bool value = false; };
3
4   template<typename T> struct is_pointer<T*>
5   {  static constexpr bool value = true; };
6
7   template<typename X>
8   void f(X x) {
9       if (is_pointer<X>::value)
10        ...
11  }
```

Se utiliza para realizar identificación de tipos en tiempo de compilación: *inferencia de tipos*.

```
1   template<typename T> struct remove_pointer
2   {  using type = T; };
3
4   template<typename T> struct remove_pointer<T*>
5   {  using type = T; };
6
7   template<typename X>
8   void f(X x) {
9       typename remove_pointer<X>::type local_x;
10        ...
11  }
```

En C++ hay muchas reglas sintácticas y semánticas, que en situaciones usuales pasarán desapercibidas:

```
1  template<typename T>
2  bool before(const T& a,const T& b)
3  {
4    return (a<b);
5  }
```

```
1  int main()
2  {
3    int    a = 1;
4    int    b = 2;
5
6    cout << before(a,b) << endl;
7  }
```

OK

En C++ hay muchas reglas sintácticas y semánticas, que en situaciones usuales pasarán desapercibidas:

```
1  template<typename T>
2  bool before(const T& a,const T& b)
3  {
4    return (a<b);
5  }
```

```
1  class Person
2  {
3    int age;
4  };
```

```
1  int main()
2  {
3    Person a = { ... };
4    Person b = { ... };
5
6    cout << before(a,b) << endl;
7  }
```

¡Error!

¿ Por qué ?

En C++ hay muchas reglas sintácticas y semánticas, que en situaciones usuales pasarán desapercibidas:

```
1  template<typename T>
2  bool before(const T& a,const T& b)
3  {
4    return (a<b);
5  }
```

```
1  class Person
2  {
3    int age;
4    bool operator< (...)
5  };
```

```
1  int main()
2  {
3    Person a = { ... };
4    Person b = { ... };
5
6    cout << before(a,b) << endl;
7  }
```

OK

Implementamos en nuestra clase la funcionalidad requerida para poder usar el genérico.

En C++ hay muchas reglas sintácticas y semánticas, que en situaciones usuales pasarán desapercibidas:

```
1  template<typename T>
2  bool before(const T& a,const T& b)
3  {
4    return (a<b);
5  }
```

```
1  int main()
2  {
3    const char* a = "hallo";
4    const char* b = "aufwiedersehen";
5
6    cout << before(a,b) << endl;
7  }
```

¡Error!

¿ Por qué ?

En C++ hay muchas reglas sintácticas y semánticas, que en situaciones usuales pasarán desapercibidas:

```
template<typename T>
bool before(const T& a,const T& b)
{
  return (a<b);
}
```

```
bool before(const char* a,const char* b)
{
  return (strcmp(a,b)<=0);
}
```

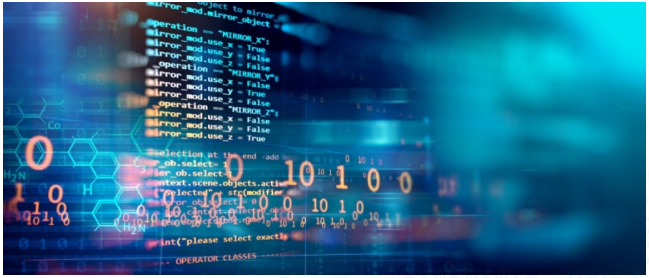
```
int main()
{
  const char* a = "hallo";
  const char* b = "aufwiedersehen";

  cout << before(a,b) << endl;
}
```

OK

Implementamos una especialización del genérico que funcione correctamente con nuestro tipo de datos.

Programación Genérica

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