Programación en C++ (10) : patrones (*templates*)

Dr. J.B. Hayet

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS

Octubre 2009

Outline

Outline

Templates : organización en archivos

Se podría pensar que el código de los cuerpos de los métodos de clases template van en archivos .cpp, y las definiciones en .h, es una posibilidad.

Otra posibilidad es considerar que eso no es verdadera implementación (la verdadera implementación la genera el compilador) entonces se pondría todo en *headers*.

Una propuesta intermediaria y razonable :

- Definiciones en Arreglo.h
- Código patrón en Arreglo_template.h
- Implementación en Arreglo.cpp

Templates : organización en archivos

Arreglo.h :

```
#ifndef _ARREGLO_H
#define _ARREGLO_H
template<class T>
class Arreglo {
  T *datos;
  unsigned int size;
public:
   Arreglo(unsigned int s=1);
   ~Arreglo();
   T& operator[](unsigned int index);
};
#endif
```

Templates : organización en archivos

Arreglo_template.h :

```
#include "Arreglo.h"

template<class T>
Arreglo<T>::Arreglo(unsigned int s) {...};

template<class T>
Arreglo<T>::~Arreglo() {...};

template<class T>
T& Arreglo<T>::operator[](unsigned int index);
};
```

Templates : organización en archivos

Arreglo.cpp :

```
#include "Arreglo_template.h"
// Explicit instanciations
template class Arreglo<int>;
template class Arreglo<double>;

typedef Arreglo<int> iArreglo;
typedef Arreglo<double> dArreglo;
```

Templates : organización en archivos

En este ejemplo, la linea :

```
template class Arreglo<int>;
```

pide explícitamente al compilador que me compile el código correspondiendo a las versiones particulares que menciono.

Templates : ejemplo, pila

Definición :

```
template<class T>
class Stack {
  unsigned int maxS;
  unsigned int s;
  T *datos;
public:
  Stack(unsigned int maxSize=100);
  void push(const T& o);
  T pop();
  unsigned int size();
};
```

Templates : ejemplo, pila

Implementación template :

```
template<class T>
Stack<T>::Stack(unsigned int maxSize) : s(0),
                                        maxS(maxSize) {
   if (!maxSize) throw 1;
   datos = new T[maxSize];
}
template<class T>
void Stack<T>::push(const T& o) {
   if (s>=maxS) throw 1;
   datos[s++] = o;
}
```

Observar que hay operadores (aquí, el de asignación) que están usados pero eventualmente no existen o no son públicos. El compilador marcaría el error compilando la versión Stack<Toto>.

Templates : ejemplo, pila

Implementación template :

```
template<class T>
T Stack<T>::pop() {
if (s==0)
        throw 1;
return datos[--s];
}

template<class T>
unsigned int Stack<T>::size() {
        return s;
}
```

Templates : ejemplo, pila

```
Stack<int> s;
s.push(1);
s.push(-1);
s.push(10);
while (s.size()) {
   cout << s.pop() << endl;
Stack<double> s2;
Stack<string> s3;
Stack<Fresa> s4;
```

Templates

Notar que no habrá en la ejecución objetos de tipo

`Stack`

sino varios tipos deducidos del código que tú escribiste para Stack :

```
Stack<int>
Stack<Integer>...
```

Templates

- Los templates pueden implicar varios "tipos parametros" :

```
template <class T, class U, class V>
class claseConMuchosTipos {
...
};
```

- Se le puede pasar en la lista objetos que non son tipos sino objetos de los tipos de base (int...), con eventualmente valores por default :

```
template <class T, class U, int v=100>
class claseConMuchosTipos {
...
};
```

Con enteros, referencias o apuntadores a objetos estáticos; es un poco similar al mecanismo de las macros.

Templates : pasaje de valores constantes

Al momento de compilar, se puede invocar :

```
claseConMuchosTipos<int ,string ,200> obj;
claseConMuchosTipos<int ,string> obj2; // Valor 100
```

El 200 es un parámetro constante para una instancia de una clase, podría ser típicamente el $\mathrm{maxSize}$ del Stack precedente.

Templates : valores por default

Si sí se puede poner valores default a los parámetros constantes que podemos usar para definir el template, también se puede considerar "tipos por default" en la definición de una clase template :

```
template <class T=int>
class Stack {
...
};
...
Stack<> oneStack;
```

El la libstdc++ :

```
template<class T, class Allocator = allocator<T> >
class vector;
```

Templates : valores por default

```
template<class T, class Allocator = allocator<T> >
class vector;
```

- T es reusado en la lista de parámetros del template.
- Notar el $> >$ (espacio !).

Funciones templates

En lugar de *clases* templates, se puede considerar simples funciones templates :

```
template<class T>
T someFonc(T val = T()) {
    T retVal = T();
    ...
   return retVal;
}
```

Funciones templates

Igualmente a con clases, se compilarán las versiones de esa "familia de funciones" que corresponden a las llamadas hechas o cuya compilación es pedida explicitamente :

```
cout << someFonc(1.8f);
cout << someFonc(2);
```

Me da :

```
000001b2 S __Z8someFoncIfET_S0_
000001ec S __Z8someFoncIiET_S0_
```

Funciones templates

Una diferencia :

```
template<class T=int>
T someFonc(T val = T()) {
    ...
}
```

no compila :

```
error: default template arguments may not be used in
    function templates
```

Pero sí se puede usar valores por default de los argumentos de la función definidos en términos del template (como en el ejemplo de someFonc).

Funciones templates

Otra diferencia : puedo usar en mi código un nombre “incompleto”, el compilador se encarga de usar el buen nombre.

```
cout << someFonc(1.8f);
cout << someFonc(2);
```

Eso ya no es posible cuando tengo mas de dos parámetros templates :

```
template<class T, class U>
T someFonc(T val = T()) {...};
```

```
error: no matching function for call to 'someFonc(float)'
error: no matching function for call to 'someFonc(int)'
```

```
cout << someFonc<float, int>(1.8f, 2);
```

Parámetro template de template

Hay la posibilidad de pasar como parametro de un template otro template, típicamente para escribir código en términos de containers (genéricos). Ahora, las posibilidades son limitadas porque se requieren interfaces suficientemente unificadas :

```
template<class T, template<class> class Seq>
class Container {
  Seq<T> seq;
public:
  void append(const T& t) { seq.push_back(t); }
  T* begin() { return seq.begin(); }
  T* end() { return seq.end(); }
};
```

La palabra llave typename

Problema aquí :

```
template<class T> class Nested {
  T::subc sub;
public:
  void someMethod() { sub.otherMethod(); }
};
```

¿Por qué será?

```
test1.cpp:82: error: expected ';' before 'sub'
```

La palabra llave typename

Cada vez que el compilador verá mención de un sub-elemento de una clase genérica (sin poder saber cómo es la clase realmente), puede vacilar en la interpretación de subc :

- o es un sub-objeto static,
- o es una sub-clase o estructura (o sea un tipo).

Por default, toma la primera opción !

La palabra llave typename

Para salir de este problema, se necesita usar la palabra llave typename :

```
template<class T> class Nested {
  typename T::subc sub;
public:
  void someMethod() { sub.otherMethod(); }
};
```

que especifica al compilador que tiene que leer lo que sigue como tipo.

La palabra llave typename

Otro ejemplo tipico :

```
template<class T>
void printList(list<T>& l) {
  for(typename list<T>::iterator b = l.begin();
        b != l.end();)
    cout << *b++ << endl;
}
```

Métodos miembros templates

Como funciones, métodos pueden estar definidas como templates, dentro de clases normales, o aun dentro de clases templates. Por ejemplo, los constructores por copia son candidatos privilegiados para ser templates-dentro-de-clases-templates :

```
template <class T>
class Point {
        T X;
        T Y;
public:
        Point(T &xx,T &yy);
        template <class U>
        Point(const Point<U> &otro);
        ...
}
```

Clases miembros templates

Igual :

```
template <class T>
class Toto {
        public:
        template <class U> class Tutu {
        };
};
Toto<int>::Tutu<float> a;
```

Outline

Otro uso de la palabra llave typename

Podrán ver a veces que, en la declaración de un patrón, no se usa la palabra class sino la palabra typename :

```
template <typename T>
class Point {
        ...
};
```

Aunque el resultado es lo mismo, unos lo prefieren porque class podría parecer demasiado reductor.

Funciones patrones

Contenido de stl_algobase :

```
template<typename _Tp>
    inline const _Tp& min(const _Tp& __a,
                          const _Tp& __b) {
      if (__b < __a)
        return __b;
      return __a;
    }
```

Sería pesado invocar sistemáticamente min<int> o min<double>...

Funciones patrones : deducción del tipo

En práctica, no es necesario :

```
double x;
...
double m = min(0.0,x);
```

La deducción es posible porque paso 2 objetos de mismo tipo !
Pero :

```
int i;
...
double m = min(0.0,i);
```

```
error: no matching function for call
to 'min(double, int&)'
```

Ambigüedad!

Funciones patrones : deducción del tipo

Como siempre (valores por default...), hay que darle suficiente información al compilador para que pueda determinar la buena versión. Opciones posibles:

- Hacer un cast de uno de los dos argumentos hacia el tipo del otro.
- Escribir las funciones patrones con argumentos desdoblados, pero, aquí, ¿qué regresar ?

Se necesitan los tipos correspondiendo al patrón.

Funciones patrones : deducción del tipo

Cuidado con una función patrón con un tipo parámetro de regreso :

- No deducción si el patrón se aplica sólo al valor de regreso ,

```
template <class T>
T someTest1() { ... }
int i = someTest1(); // NO !
```

- Declarar primero el tipo de regreso para poder omitir el tipo del argumento.

```
template <class T, class U>
T someTest2(U &a) { ... }
int j = someTest2(0); // NO !
int k = someTest2<int>(0); // OK !
```

Funciones patrones sobrecargadas

Se puede hacer coexistir funciones template y versiones especificas

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

y

```
const char* min(const char* a, const char* b) {
  return (strcmp(a, b) < 0) ? a : b;
}
int min(int x, int y) {
  return (x < y) ? x : y;
}
```

Funciones patrones sobrecargadas

Y en este caso, se elige la versión más adecuada al caso, o, eventualmente, se fuerza el uso de una versión patrón :

```
double a    = min(1,0); // ¿ Cuál ?
double b    = min(1.0,0.0); // ¿ Cuál ?
double c    = min(1,0.0); // ¿ Cuál ?
const char *d    = min("tutu","toto"); // ¿ Cuál ?
const char *e    = min<>("tutu","toto"); // ¿ Cuál ?
```

Funciones patrones : dirección

Apuntadores a funciones con patrones, en dos casos :

- Manipulación de apuntadores a funciones que son implementadas a partir de patrones.

```
template <class T>
T toto(T &) {
  return T();
}
double eval(double x,double (*fptr)(double&)) {
  return (*fptr)(x);
};
eval(1.0,&toto<double>);
```

- Funciones patrones que manipulen a apuntadores a funciones.

```
template<typename T> void tutu(T (*pf)(T&)) {}
tutu<int>(&toto<int>);
```

Funciones patrones : dirección

Otra vez, se puede inferir el buen tipo que usar sobre el patrón :

```
eval(1.0,&toto); // OK, toto con double
tutu<int>(&toto); // OK, toto con int
tutu(&toto<float>); // OK, tutu con float
```

Funciones patrones : contenedores

Para funciones patrones sobre contenedores genéricos, el único requerimiento es la existencia de un iterador sobre este contenedor.
Ejemplo : una meta-función que llama un método dado de una clase sobre todos los objetos contenidos dentro de un contenedor :

```
template <class Container, class T, class R, class X>
void apply(Container& c, R (T::*method)(X &), X &arg)
  typename Container::iterator it = c.begin();
  while(it != c.end())
    ((*it++).*method)(arg);
}
template <class Container, class T, class R, class X>
void apply(Container& c, R (T::*method)(const X &), X&
  typename Container::iterator it = c.begin();
  while(it != c.end())
    ((*it++).*method)(arg);
}
```

Funciones patrones : contenedores

...y se llamaría simplemente de esa manera:

```
class someClass {
  double data;
public:
  int someMethod1(const double &) {};
};

int main() {
  someClass s1,s2,s3;
  std::list<someClass> l;
  l.push_back(s1); l.push_back(s2);  l.push_back(s3);
  double a=1.0;
  apply(l,&someClass::someMethod1,a);
}
```

Notar aquí que el contenedor contiene instancias de objetos.

Patrones : orden de implementación

Cuando se sobrecarga funciones patrones, puede haber ambigüedades,

```
template<class T> void f(T);
template<class T> void f(T*);
template<class T> void f(const T*);
```

Esos tres patrones son más y más especializados, y, por ejemplo, el último está ya implícitamente implementable con los dos primeros patrones !

Patrones : orden de implementación

Por eso existe un orden de prioridades en la selección de las implementaciones correctas, para el compilador : se elegirá implementar la versión mas especializada que se puede !

```
double x=1.0;
const double y=2.0;
f(x); // Implementa la primera
f(&x);// Implementa la segunda
f(&y);// Implementa la tercera
```

Si no se puede encontrar forma adecuada, reportará error...

Especialización explícita

En lugar de definir funciones “normales” para sobrecargar un patrón en un caso dado,

```
template<typename T>
const T& min(const T& a,const T& b) {
  return (a < b) ? a : b;
}
const char* min(const char* a, const char* b) {
  return (strcmp(a, b) < 0) ? a : b;
}
```

Se puede definir esa segunda como versión especializada del patrón !
Pero tiene que cumplir precisamente la definición del patrón.

```
template<>  // Indica que es una especialización
const char* min<const char *>(const char* a,
                              const char* b) {
  return (strcmp(a, b) < 0) ? a : b;} // NO !
```

Especialización explícita

```
template-id 'min<const char*>' for 'const char*
min(const char*, const char*)' does not match
any template declaration
```

Para que corresponda exactamente al patrón :

```
template<>
const char*
const &min<const char *>(const char* const &a,
                         const char* const &b) {
  return (strcmp(a, b) < 0) ? a : b;
} // OK !
```

Igual para clases pero todos los métodos tienen que estar reescritos (cf. vector<bool> en stl_bvector.h).

Especialización parcial

En el caso de varios parámetros para el patrón, se puede definir versiones parcialmente especializadas :

```
template<typename _Alloc>
class vector<bool, _Alloc> :
public _Bvector_base<_Alloc> {
...
```

Híbrido entre una definición de patrón y una especialización...

Especialización parcial

Con especialización parcial, surgen los mismos problemas de ambigüedades que con patrones sobrecargados (de hecho una versión especializada parcialmente se puede ver como sobrecargando el patrón). Las mismas reglas se aplican: es la versión mas especializada que esta implementada; si hay ambigüedad, saldrá error del compilador,

Ejemplo :

```
template<class T, class U> class myClass {
public: myClass() { cout << "Generica" << endl;}
};
```

Especialización parcial

```
template<class U> class myClass<int,U> {
public: myClass() { cout << "T como int" << endl;}
};
template<class T> class myClass<T,double> {
public: myClass() { cout << "U como double" << endl;}
};
template<class T> class myClass<T,int> {
public: myClass() { cout << "U como int" << endl;}
};
myClass<int,int> c1; // NO
myClass<int,float> c2; // OK
myClass<int,double> c3; // NO
```

Herencia con patrones

Se hace igual que para clases normales:

```
template<class T,class U>
class otherClass:public myClass<T,U> {
public:
  void someMethod() {};
};
template<class T>
class otherClass<T,double>:public myClass<T,double> {
public:
  void someMethod() {};
};
template<class T,class U>
class otherCass<T,U*>:public myClass<T,U*> {
public:
  void someMethod() {};
};
```

Código generado a partir de patrones

- Sólo las funciones/clases necesarias son implementadas.
- Aun mejor, dentro de una clase instanciada por patrón, sólo los métodos necesarios son generados

```
template<class T, class U>
class otherClass:public myClass<T,U> {
public:
   void someMethod1() {};
   void someMethod2() {};
};
myClass<int ,double> oA;
myClass<int ,int> oB;
oA.someMethod2();
oB.someMethod1();
```

Problemas de nombres...

Para todos los símbolos (funciones, variables...) que contiene el código, el compilador normalmente intenta determinar

- su tipo,
- su scope,
- eventualmente su duración.

Para los patrones no puede pasar así: hasta que veamos como se va a instanciar efectivamente el patrón, no se puede adivinar todo...

Problemas de nombres...

Antes de haber instanciado objetos a partir de patrones, el compilador pasa una primera vez sobre el código e intenta asociar los nombres a funciones, con lo que puede deducir :

- de calificaciones de nombres, por ejemplo

```
template<class T, class U>
class otherClass:public myClass<T,U> {
public:
  void someMethod1() {someNameSpace::fonc();};
};
```

- de Argument Dependent Lookup : si no encuentra calificación entera, busca en los espacios de nombres de los argumentos :

```
  void someMethod2() {std::cout << var;};
```

Problemas de nombres...

```
void someMethod2() {std::cout << var;};
```

- Argument Dependent Lookup: aquí el compilador no sabe en que espacio de nombre buscar por el operador <<
- Busca en
  - std porque uno de los argumentos tiene una calificación con este espacio de nombre,
  - eventualmente con el segundo argumento.

Problemas de nombres...

La búsqueda de funciones calificadas vale para funciones ya conocidas

```
void fonc(double) {};
template<class T,class U>
class otherClass:public myClass<T,U> {
public:
  void someMethod1() {fonc(0);}; // OK
};
void fonc(int) {};
```

Problemas de nombres...

Al recorrer un patrón desde el compilador :

- Si los símbolos encontrados no dependen de un patrón, se busca una interpretación calificada aceptable o se busca una por exploración de los espacios de nombres de los argumentos (ADL), con lo que ya sabemos.
- Si los símbolos son dependientes de una manera de parámetros del patrón, están resueltos al momento de la instanciación, excepto si no son suficientemente calificados : en este caso, se necesita ligarlos con objetos del buen espacio de nombre.
- Si estamos heredando de una clase patrón, no olvidar que los símbolos de esta clase base no estarán vistos en el primer recorrido !

Problemas de nombres...

```
#include <algorithm>
#include <iostream>
#include <typeinfo>
using std::cout;
using std::endl;

void g() { cout << "global g()" << endl; }

template<class T> class Y {
public:
  void g() {cout << "Y<" << typeid(T).name() << ">::g()" << endl;}
  void h() {cout << "Y<" << typeid(T).name() << ">::h()" << endl;}
  typedef int E;
};

typedef double E;

template<class T> void swap(T& t1, T& t2) {
  cout << "global swapÓ <<endl;
  T temp = t1;
  t1 = t2;
  t2 = temp;
}
```

Problemas de nombres...

```
template<class T> class X : public Y<T> {
public:
  E f() {
    g();
    this->h();
    T t1 = T(), t2 = T(1);
    cout << t1 << endl;
    swap(t1, t2);
    std::swap(t1, t2);
    cout << typeid(E).name() << endl;
    return E(t2);
  }
};

int main() {
  X<int> x;
  cout << x.f() << endl;
}
```

Muchas ambigüedades, que no obstante estan arregladas por esas reglas.

Problemas de nombres...

- Valor de regreso de $X :: f()$, E es un nombre no dependiente, entonces desde la primera fase, esta asociado al E global (notar que no hubiera pasado así con una clase normal).
- Llamada a $g()$: sigue el mismo principio (no es un nombre dependiente de un parámetro del patrón), y se asocia a la función $g()$ global, aunque hay una $g()$ en Y.
- Llamada a $h()$ calificada y dependiente (por el this depende de X y $Y < T >$); en este caso se deja la asociación para el momento de la instanciación.
- $t1$ y $t2$ tienen declaraciones dependientes.

Problemas de nombres...

- la llamada al operador $<<$ es dependiente; la liga se hace mas tarde con el de std esperando a un entero,
- llamada a swap: dependiente y no calificada, eso hace que se asociará en la instanciación del patrón a la función patrón global,
- la llamada a *std* :: *swap*() no es "suficientemente" dependiente (es de std, que no depende de T).

Patrones y friends

Normalmente, el compilador busca las funciones/clases friend no calificadas en el espacio de nombre justo arriba del espacio en que se menciona la friendship,

```
#include <iostream>
using namespace std;
class Otorga {
  int i;
public:
  Otorga(int ii) { i = ii; }
  friend void fonc(const Otorga&); // Hara el link con ::fonc
  void method() { fonc(*this); }
};
void h() {
  fonc(Otorga(1));
} // Como funciona ?
void fonc(const Otorga& fo) { cout << fo.i << endl;}
int main() {
  h();
  Otorga(2).method();
  fonc(1);
}
```

Patrones y friends

Notar que lo siguiente normalmente no va a compilar:

```
void h() {
  fonc(1);
}
```

por que no hay ningún indice de como asociar fonc a una declaración existente!

Patrones y friends

En el caso de templates, hay un cambio en el que se necesita declarar la función template antes de poder otorgarla friendship :

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

template<class T> class Otorga;
template<class T> void fonc(const Otorga<T>&);
template <class T>
class Otorga {
  T i;
public:
  Otorga(T ii) { i = ii; }
  friend void fonc<>(const Otorga<T>&); // Hara el link con ::f
  void method() { fonc(*this); }
};
void h() { fonc(Otorga<int>(1));} // Como funciona ?
template <class T> void fonc(const Otorga<T>& fo) { cout << fo.i << e
int main() {
  h();
  Otorga<double>(2).method();
}
```

Patrones y friends

Unas remarcas :

- Se puede otorgar friendship solamente a una versión del patrón

```
friend void fonc<>(const Otorga<double>&);
```

- La función declarada friend puede no depender de los parámetros del patrón, y en este caso esta friend de todas las clases implementadas por el patrón.

```
friend void fonc();
```

- Si la función sí es template, se puede otorgar la friendship a todas sus versiones, desde todas las versiones de la clase que otorga :

```
template<class U>
friend void fonc<>(const Otorga<U>&);
```

Patrones : traits

Es una técnica que sale naturalmente del uso de los patrones : cuando tienes que manejar varias tipos/clases distintas, consiste en definir clases patrones intermediarias que contienen, aparte, las características de las clases que se manipulan en el patrón.

```
class MobileRobot {
 friend ostream& operator<<(ostream& os, const MobileRobot &m) {
    return os << "a mobile robot ";}
};
class  Arm{
 friend ostream& operator<<(ostream& os, const Arm &a) {
    return os << "an arm ";}
 };
class WheelMotors{
 friend ostream& operator<<(ostream& os, const WheelMotors &w) {
    return os << "wheels motors ";}
};
class Humanoid {
 friend ostream& operator<<(ostream& os, const Humanoid &h) {
    return os << "a humanoid robot ";}
};
```

Patrones : traits

Un patrón para características :

```
template<class RobotType> class RobotTraits;
template<> class RobotTraits<MobileRobot> {
public:
  typedef WheelMotors actuator_type;
};
template<> class RobotTraits<Humanoid> {
public:
  typedef Arm actuator_type;
};
```

Patrones : traits

Luego se puede usar en una clase combinando objetos/características, como alternativa a herencia, y eventualmente con excepciones a la asociación default :

```
template<class RobotType, class traits = RobotTraits<RobotType> >
class Robot {
  RobotType type;
  typedef typename traits::actuator_type actuator_type;
  actuator_type actuator;
public:
  Robot(const RobotType& r) : type(r) {}
  void actuate() {
    cout << "Actuating " << type
         << " through " <<  actuator << endl;
  }
};
int main() {
  Humanoid h; MobileRobot m;
  Robot<Humanoid> r1(h);
  Robot<MobileRobot> r2(m);
  r1.actuate(); r2.actuate();
  Robot<MobileRobot,ArmTrait> r3(m);
}
```

Patrones : traits

Ventajas:

- mas genericidad y flexibilidad en el código: permite en particular manejar de manera transparente optimizaciones que hacer para tal o tal tipo, dentro de patrones,
- meta-razonamiento sobre los tipos,
- puedes escribir esa clase patrón aunque no tengas acceso a la clase original.

Patrones : traits

```
template <typename T>
struct CallTraits {
    template <typename U, bool Big> struct CallTraitsImpl;
    template <typename U> struct CallTraitsImpl<U, true> {
        typedef const U& Type;
    };

    template <typename U> struct CallTraitsImpl<U, false> {
        typedef U Type;
    };

    typedef typename CallTraitsImpl<T, (sizeof(T) > 8)>::Type ParamTy
};
```

Patrones : traits

```
template <typename T>
T Min(typename CallTraits<T>::ParamType X,
      typename CallTraits<T>::ParamType Y)
{
    return X < Y ? X : Y;
}

std::string a = "Hola";
std::string b = "Que tal";
std::string c = Min<std::string>(a,b); // por referencia

int i = 1;
int j = 2;
int k = Min<int>(i,j); // por valor
```