TAD genéricos

Lección 4

Esquema

- Concepto de genericidad
- TAD genérico
- Implementación de TAD genérico:
  - En el lenguaje modular seudocódigo
  - En esta asignatura en C++

Genericidad → Facilita la reutilización de código y de algoritmos

- La genericidad es un mecanismo que permite:
  - Escribir trozos de ***código fuente genérico*** (subprogramas, módulos, clases…), que incluye menciones a uno o varios ***nombres de tipos*** de datos o incluso a ***algoritmos***, sin que exista una declaración de los mismos (se denominan ***parámetros de tipo, de procedimiento*** o ***de función***).
  - Especificar ***restricciones*** para los parámetros de tipo del código genérico.
  - Particularizar (***concretar***) el código anterior, indicando los tipos o algoritmos concretos existentes (es decir, ya declarados), en los que se convierten los parámetros de tipos, de procedimiento o de función, y generando código concreto (todo esto hecho por el compilador en tiempo de compilación, o en la fase de *enlace* sobre código objeto genérico).

TAD genérico

- Es un TAD en cuya especificación aparece un *parámetro formal* que es el nombre de un tipo desconocido (no definido).

  →Un TAD genérico pueden tener varios *parámetros formales*

Podremos:

- Especificar TAD genéricos

  → implementar TAD genéricos

TAD genérico

- Especificar TAD genéricos

  → implementar TAD genéricos

  En seudocódigo

Ejemplo: máquina expendedora de frutas…

TAD genérico

Estos dos TAD ***no*** son genéricos:

espec monedas
usa naturales
género moneda
{Los valores del TAD moneda representan valores posibles de una moneda}
operaciones
c1: → moneda
{devuelve una moneda de 1 céntimo}
c10: → moneda
{devuelve una moneda de 10 céntimos}
c50: → moneda
{devuelve una moneda de 50 céntimos}
e1: → moneda
{devuelve una moneda de 1 euro}
precio: moneda m → natural
{devuelve el valor de la moneda m en céntimos}
fespec

espec monederos
usa monedas, naturales
género monedero
{Los valores del TAD monedero representan valores posibles de un multiconjunto (saco) de monedas}
operaciones
vacío: → monedero
{devuelve un monedero vacío, sin monedas}
meter: monedero s , moneda m → monedero
{devuelve un monedero igual al resultante de añadir la moneda m a s}
sacar: monedero s , moneda m → monedero
{devuelve un monedero igual al resultante de extraer una moneda m de s; si no hay ninguna moneda m en s, devuelve un monedero igual a s}
cuántas: monedero s , moneda m → natural
{devuelve el nº de monedas de valor m en s}
valor: monedero s → natural
{devuelve la suma de los valores de todas las monedas de s}
fespec

TAD genérico

El código resultante de TADs ***no*** genéricos es ***no*** genérico.

```
módulo monedas
exporta
  tipo moneda=(c1,c10,c50,e1)   {todo tipo enumerado tiene automáticamente
     definidas operaciones como: =, !=, >,<,≤,≥, predecesor, sucesor, primero,
     último, pos, ord, copia o asignación(:=)}
  función precio (m:moneda) devuelve natural
implementación
  función precio (m:moneda) devuelve natural
  principio
    selección
      m=c1: devuelve 1;
      m=c10: devuelve 10;
      m=c50: devuelve 50;
      m=e1: devuelve 100;
    fselec
  fin
fin
```

TAD genérico

El código resultante de TADs ***no*** genéricos es ***no*** genérico.

```
módulo monederos
importa monedas
exporta
  tipo monedero
  procedimiento vacio(sal s:monedero)
  procedimiento meter(e/s s:monedero;
                      ent m:moneda)
  procedimiento sacar(e/s s:monedero;
                      ent m:moneda)
  función cuántas(s:monedero; m:moneda)
          devuelve natural
  función valor(s:monedero)devuelve natural
implementación
  tipo monedero=vector[moneda] de natural
  procedimiento vacio(sal s:monedero)
  variable m:moneda
  principio
    para m:=c1 hasta e1 hacer
      s[m]:=0
    fpara
  fin
  procedimiento meter(e/s s:monedero;
                      ent m:moneda)
  principio
    s[m]:=s[m]+1
  fin
  procedimiento sacar(e/s s:monedero;
                      ent m:moneda)
  principio
    si s[m]>0 entonces s[m]:=s[m]-1 fsi
  fin
  función cuántas(s:monedero; m:moneda)
          devuelve natural
  principio
    devuelve s[m]
  fin
  función valor(s:monedero) devuelve natural
    variables v:natural; m:moneda
  principio
    v:=0;
    para m:=c1 hasta e1 hacer
      v:=v+cuántas(s,m)*precio(m)
    fpara;
    devuelve v
  fin
fin
```

TAD genérico

Otros dos TAD ***no*** genéricos.

espec frutas
usa naturales
género fruta
{Los valores del TAD fruta representan valores posibles de una fruta}
operaciones
pera: → fruta
{devuelve una pera}
manzana: → fruta
{devuelve una manzana}
limón: → fruta
{devuelve un limón}
pomelo: → fruta
{devuelve un pomelo}
papaya: → fruta
{devuelve una papaya}
precio: fruta f → natural
{devuelve el valor de la fruta f}
fespec

La especificación es ***idéntica*** a la de los monederos (***cambiando*** moneda ***por*** fruta)

espec frutero
usa frutas, naturales
género frutero
{Los valores del TAD frutero representan valores posibles de un multiconjunto (saco) de frutas}
operaciones
vacío: → frutero
{devuelve un frutero vacío, sin frutas}
meter: frutero s , fruta f → frutero
{devuelve un frutero igual al resultante de añadir la fruta f a s}
sacar: frutero s , fruta f → frutero
{devuelve un frutero igual al resultante de extraer una fruta f de s; si no hay ninguna fruta f en s, devuelve un frutero igual a s}
cuántas: frutero s , fruta f → natural
{devuelve el nº de frutas de valor f en s}
valor: frutero s → natural
{devuelve la suma del precio de todas las frutas de s}
fespec

TAD genérico

Parametrización del TAD para ahorrar código y tiempo de desarrollo

→ especificación del TAD ***genérico*** saco**:**

espec sacosGenéricos
usa naturales, booleanos
parámetro formal
género elemento
operaciones
precio: elemento e → natural *{devuelve…}*
_=_: elemento e1, elemento e2 → booleano*{devuelve…}*

el **parámetro formal** es un tipo no definido (*elemento*) **al que se le exigirá** tener definida una operación con el perfil que tiene la operación *precio* y una operación de *comparación por igualdad*

Las ***restricciones*** sobre los tipos se expresan como ***operaciones***

fpf
género saco ← es el nombre del TAD genérico
{Los valores del TAD genérico saco representan valores posibles de un multiconjunto (saco) de datos de tipo elemento…}
operaciones
vacío: → saco *{devuelve un saco vacío, sin elementos}*
meter: saco s , elemento e → saco
{devuelve un saco igual al resultante de añadir el elemento e a s}
sacar: saco s , elemento e → saco
{devuelve un saco igual al resultante de extraer un elemento e de s; si no hay ningún elemento e en s, devuelve un saco igual a s}
cuántas: saco s , elemento e → natural *{devuelve el nº de elementos iguales a e en s}*
valor: saco s → natural
{devuelve la suma del precio de todos los elementos de s; para su cálculo será preciso usar la operación precio, que deberá estar definida para los datos de tipo elemento}
fespec

Implementación del módulo genérico saco

```
módulo genérico sacosGen
parámetros
   tipo elemento
   con función precio(e:elemento) devuelve natural{doc...}
   con función "="(e1,e2:elemento) devuelve booleano{doc...}
exporta
   tipo saco
   procedimiento vacio(sal s:saco)
   procedimiento meter(e/s s:saco; ent e:elemento)
   procedimiento sacar(e/s s:saco; ent e:elemento)
   función cuántas(s:saco; e:elemento) devuelve natural
   función valor(s:saco) devuelve natural
implementación
...
```

parámetro de tipo

parámetros de función: precio, "="

Implementación del módulo genérico saco

```
  //sigue …
Implementación

  constante maxNum = 1000
  tipo unElemto=registro
                          elElemto:elemento;
                          numVecesRepetido:natural
                      freg
       elementos = vector[1..maxNum] de unElemto
       saco = registro
                      losElementos:elementos;
                      numDistintos:natural
                      freg

  procedimiento vacio(sal s:saco)
  variable e:elemento
  principio
    s.numDistintos:=0
  fin   //sigue…
fin
```

Esta implementación limita los sacos a contener información de un máximo de 1000 elementos distintos (por tanto esto debería reflejarse en la Interfaz del módulo, y no siempre se podrá añadir, etc).
Implementación robusta:
procedimiento meter(**e/s** s:saco;
ent e:element; **sal** metido:booleano)

TAD genérico

- Uso del módulo genérico

```
programa titi
importa monedas,frutas,sacosGen
   {en los módulos 'monedas' y 'frutas' están definidos los
    tipos moneda y fruta, respectivamente, cada uno con una
    función 'precio', y en ambos casos también la operación
    "="}
   módulo monedero concreta sacosGen(moneda,precio,"=");
   módulo frutero concreta sacosGen(fruta,precio,"=");
    {sintaxis alternativa:
     módulo frutero = sacosGen(fruta,precio,"="); }
   variables m:monedero.saco; f:frutero.saco
principio
   ...
   vacío(m);
   meter(m,e1);
   vacío(f);
   meter(f,pera);
//meter(m,pera); ← Error en compilación: tipos incompatibles
   ...
fin
```

Implementación código genérico en C++

- C++ no implementa realmente la genericidad, es decir, no se puede obtener código objeto (compilado) que sea genérico.
- La técnica que usaremos para simular la genericidad en C++ son las plantillas *(**templates**)*
  - El compilador hace simplemente una sustitución del texto del parámetro formal por el parámetro actual.
  - Por tanto no se genera código objeto genérico sino que se genera código objeto distinto para cada particularización del parámetro formal.

Diferencia implementación TAD en C++, genérico versus no genérico

- Si un **TAD es no genérico,** su implementación se dividirá en:
  - un fichero ***.hpp*** (con las declaraciones: interfaz + representación interna)
  - y un fichero ***.cpp*** (con la implementación de las operaciones)

- Si un **TAD es genérico**, su implementación se hará:
  - en **un único fichero *.hpp*** (declaración e implementación de las operaciones)

TAD genérico

Parametrización del TAD para ahorrar código y tiempo de desarrollo

→ <u>especificación</u> del TAD ***genérico*** saco**:**

espec sacosGenéricos
usa naturales, booleanos
parámetro formal
género elemento
operaciones
precio: elemento e → natural ***{devuelve…}***
_=_: elemento e1, elemento e2 → booleano***{devuelve…}***

el **parámetro formal** es un tipo no definido (*elemento*) **al que se le exigirá** tener definida una operación con el perfil que tiene la operación *precio* y una operación de *comparación por igualdad*

Las ***restricciones*** sobre los tipos se expresan como ***operaciones***

fpf
género saco ← es el nombre del TAD genérico
{Los valores del TAD genérico saco representan valores posibles de un multiconjunto (saco) de datos de tipo elemento…}
operaciones
vacío: → saco *{devuelve un saco vacío, sin elementos}*
meter: saco s , elemento e → saco
{devuelve un saco igual al resultante de añadir el elemento e a s}
sacar: saco s , elemento e → saco
{devuelve un saco igual al resultante de extraer un elemento e de s; si no hay ningún elemento e en s, devuelve un saco igual a s}
cuántas: saco s , elemento e → natural *{devuelve el nº de elementos iguales a e en s}*
valor: saco s → natural
{devuelve la suma del precio de todos los elementos de s; para su cálculo será preciso usar la operación precio, que deberá estar definida para los datos de tipo elemento}
fespec

Implementación en C++

En C++ podremos indicar las restricciones sobre los tipos únicamente en los comentarios (por tanto el compilador no podrá comprobar que el genérico se usa con tipos que las cumplan)

```
// Interfaz del TAD (inicio parte pública). Pre-declaraciones:
template<typename Elemento> struct Saco;
// El tipo Elemento requerirá tener definida las funciones:
//        int precio( const Elemento&  e);  {… descripción…}
//        bool operator== (const Elemento& e1, const Elemento& e2); {… descripción…}
template<typename Elemento> void vacio(Saco<Elemento>& s);
template<typename Elemento> bool meter(Saco<Elemento>& s, const Elemento& e);
template<typename Elemento> void sacar(Saco<Elemento>& s, const Elemento& e);
template<typename Elemento> int cuantos(const Saco<Elemento>& s, const Elemento& e);
template<typename Elemento> int valor(const Saco<Elemento>& s);
// Fin interfaz del TAD (fin parte pública).
// Parte INTERNA u "OCULTA": declaraciones, implementación y operaciones auxiliares
template<typename Elemento> int buscar(const Saco<Elemento>& s, const Elemento& e);
template<typename Elemento> struct Saco {
    friend void vacio<Elemento>(Saco<Elemento>& s);
    friend bool meter<Elemento>(Saco<Elemento>& s, const Elemento& e);
    friend void sacar<Elemento>(Saco<Elemento>& s, const Elemento& e);
    friend int cuantos<Elemento>(const Saco<Elemento>& s, const Elemento& e);
    friend int valor<Elemento>(const Saco<Elemento>& s);
    friend int buscar<Elemento> (const Saco<Elemento>& s, const Elemento& e);
private:
   … // Representación de los valores del TAD.
}; … // Implementación de las operaciones…
```

→ *Ver completo en el material de clase*

Implementación en C++: Cuidado!!

```
template<typename Elemento>
int cuantos(const Saco<Elemento>& s, const Elemento& e) {
    ...
    ... s.elementos[i].dato == e ...
    ... return ...
    ...
}
```

{Para los tipos definidos por enumeración, quedan definidas automáticamente operaciones como: los operadores para comparaciones de igualdad, de orden, sucesor, predecesor, primero, último… }

- Si **al concretar** el parámetro formal, por ejemplo:

```
Saco<fruta> miFrutero;    // en el main
Saco<producto> miCompra;
```

el tipo de dato no tiene definido la operación (en este caso el operator==) → **error**

- **Error en tiempo de compilación**: no existe la operación == para (en este caso, si no la tuviese) producto
- **PERO**, *C++* no detecta el error de compilación hasta que no se compile un código que llame a la operación **cuantos**, que es la que intenta usar la operación que no existe

IMPORTANTE: si implementamos un TAD genérico en C++, hay que asegurarse de probar la versión final de todas y cada una de sus operaciones, o puede quedarse alguna sin compilar

Ejemplo de módulo genérico

```
módulo genérico ordenaciónGen
parámetros
  tipos ind = cualquier tipo discreto
        elem = cualquier tipo
        vect = vector[subrango de ind] de elem
  con función ">"(a,b:elem) devuelve booleano
exporta
  procedimiento ordena(e/s v:vect)
  {Procedimiento que ordena los elementos de v por
    valores crecientes según ">"}
implementación
  procedimiento ordena(e/s v:vect)
    variables i,j:ind; m,t:elem; n:entero
  principio
    ...
  fin
fin
```

parámetros de tipo

restricciones para los parámetros de tipo

parámetro de función (podría haber otros, incluso de procedimientos)

Uso de un módulo genérico

es genérico ➔ no puede usarse antes de concretarlo

```
programa prueba
importa ordenaciónGen
  tipos color = (rojo,azul,gris)
        dia = (lu,ma,mi,ju,vi,sa,do)
        miVect = vector[subrango de dia] de color
  módulo miord concreta ordenaciónGen(dia,color,miVect,">")
  variables x:vect[ma..vi]:=(gris,azul,rojo,gris)
principio
  ...
  ordena(x);
  ...
Fin
```

`miord` ya es un módulo concreto ➔ puede usarse

Visto en el curso pasado (templates en C++): búsqueda y ordenación en vectores.