

Estructuras de Datos y Algoritmos

TAD genéricos

LECCIÓN 4

© All wrongs reversed – bajo licencia CC-BY-NC-SA 4.0



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2023/2024

Grado en Ingeniería Informática

UNIVERSIDAD DE ZARAGOZA

Aula 0.04, Edificio Agustín de Betancourt



Adaptadas de diapositivas de Javier Campos

Índice

1 Concepto de genericidad

2 TAD genéricos

3 Implementación en C++

Índice

- 1 Concepto de genericidad
- 2 TAD genéricos
- 3 Implementación en C++

¿Qué permite la genericidad?

- Escribir trozos de **código genérico** (subprogramas, módulos, clases, etc.) que incluyen referencias a uno o varios nombres de tipos de datos (o clases o incluso algoritmos)
 - Sin declaración explícita: *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 concretos de dato en los que se convierte los parámetros de tipo de código genérico
 - Generando código concreto
 - Estas dos tareas son realizadas por el compilador en tiempo de compilación

Índice

1 Concepto de genericidad

2 TAD genéricos

3 Implementación en C++

TAD en cuya especificación aparece un (o varios) *parámetro formal*

- El parámetro es el nombre de un tipo básico no concretado (i.e., no definido)

TAD genéricos

■ 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 de monedas, es decir,
   pueden estar repetidas}
operaciones
  vacío: -> monedero
  {devuelve un monedero vacío, sin monedas}
  meter: monedero s, moneda m -> monedero
  {devuelve el monedero resultante de añadir
   un ejemplar de la moneda m a s}
  sacar: monedero s, moneda m -> monedero
  {devuelve el monedero resultante de extraer
   un ejemplar de la 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 unidades de la moneda m
   en s}
  valor: monedero s -> natural
  {devuelve la suma de los valores de todas
   las monedas de s}
```

fespec

TAD genéricos

El código resultante de TAD no genéricos es no genérico

```
módulo monedas
exporta
  tipo moneda = (c1, c10, c50, e1)
  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 genéricos

El código resultante de TAD no genéricos es no genérico

```
módulo monederos
importa monedas
exporta
  tipo monedero
  procedimiento vacío(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) dev. natural
implementación
  tipo monedero = vector[moneda] de natural

  procedimiento vacío(sal s: monedero)
  variable m: moneda
  principio
    para m := c1 hasta el 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 el hacer
    v := v + cuántas(s, m)*precio(m)
  fpara
  devuelve v
fin
fin {módulo monederos}
```

TAD genéricos

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

espec frutero

usa frutas, naturales

género frutero

{Los valores del TAD frutero representan valores posibles de un multiconjunto de frutas, es decir, pueden estar repetidas}

operaciones

vacío: -> frutero

{devuelve un frutero vacío, sin frutas}

meter: frutero s, fruta f -> frutero

{devuelve el frutero resultante de añadir un ejemplar de la fruta f a s}

sacar: frutero s, fruta f -> frutero

{devuelve el frutero resultante de extraer un ejemplar de la 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 n° de unidades de la fruta f en s}

valor: frutero s -> natural

{devuelve la suma del precio de todas las frutas de s}

fespec

TAD genéricos

- Codificación idéntica a los monederos (cambiando moneda por fruta)
- Parametrización del TAD para ahorro de código y tiempo de desarrollo: el TAD *genérico* saco

```
espec sacosGenéricos
  usa naturales
  parámetro formal ←
    género elemento
  operaciones
    precio: elemento e -> natural
    iguales: elemento e1, elemento e2 -> bool

fpf
género saco ←
{Los valores del TAD saco representan valores posibles de un
 multiconjunto de elementos, es decir, pueden estar repetidos}

operaciones
vacío: -> saco
{devuelve un saco vacío, sin elementos}
meter: saco s, elemento e -> saco
{devuelve el saco resultante de añadir un ejemplar del elemento
 e a s}
sacar: saco s, elemento e -> saco
{devuelve el saco resultante de extraer un ejemplar del
 elemento e de s; si no hay ningún elemento e en s, devuelve
 un saco igual a s}
cuántos: saco s, elemento e -> natural
{devuelve el n° de unidades del elemento e que hay 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 deber
 á estar definida para los datos de tipo elemento}
```

El parámetro es un tipo no definido (elemento) al que se le exige tener definidas unas operaciones con los perfiles que tienen la operación precio e iguales

Nombre del TAD genérico

TAD genéricos

Implementación del módulo genérico sacco

```
módulo genérico sacosGen
```

```
parámetros
```

```
  tipo elemento
```

```
  con función precio(e: elemento) devuelve natural
```

```
  con función iguales(e1, e2: elemento) devuelve booleano
```

```
exporta
```

```
  tipo sacco
```

```
  procedimiento vacío(sal s: sacco)
```

```
  procedimiento meter(e/s s: sacco; ent e: elemento)
```

```
  procedimiento sacar(e/s s: sacco; ent e: elemento)
```

```
  función cuántos(s: sacco; e: elemento) devuelve natural
```

```
  función valor(s: sacco) devuelve natural
```

```
implementación
```

```
  ...
```

Parámetro de tipo (sin restricciones)

Parámetros de función

TAD genéricos

Implementación del módulo genérico saco

```
...
implementación
  constante maxNum = 10000000
  tipo unElemto = registro
                    elElemto: elemento;
                    numVecesRepetido: natural
  freg
  elementos = vector[1..maxNum] de unElemto
  saco = registro
          losElementos: elementos;
          numDistintos: natural
  freg

  procedimiento vacío(sal s: saco)
  variable e: elemento
  principio
    s.numDistintos := 0
  fin
  ...
fin {módulo sacosGen}
```

TAD genéricos

Uso de módulos genéricos

```
procedimiento foobar
importa monedas, frutas, sacosGen
{en los módulos 'monedas' y 'frutas' están definidos los tipos moneda,
  fruta, y una función 'precio' para monedas y otra para frutas;
  suponemos que en pseudocódigo está predefinida
  la comparación de igualdad, "=", para datos enumerados}
módulo monedero = sacosGen(moneda, precio, "=");
módulo frutero = sacosGen(fruta, precio, "=");
variables m: monedero.saco; f: frutero.saco
principio
  ...
  vacío(m);
  meter(m, el);
  vacío(f);
  meter(f, pera);
  ...
fin
```

Índice

- 1 Concepto de genericidad
- 2 TAD genéricos
- 3 Implementación en C++**

Implementación en C++

■ C++ no implementa realmente la genericidad

- No se puede obtener un código objeto (compilado) que sea genérico

■ La genericidad en C++ se puede *simular* mediante plantillas (llamadas *templates*)

- El compilador sustituye el texto del parámetro formal por el parámetro actual
- No se genera un código objeto genérico, sino que se genera un código objeto distinto para cada particularización del parámetro formal

¡OJO! En las *templates*, **el archivo de cabecera (extensión .hpp) incluirá la implementación de las operaciones**

Implementación en C++

Ejemplo: módulo genérico saco (archivo «saco.hpp»)

```
#ifndef _SACO_HPP_
#define _SACO_HPP_

const int MAX_NUM_ELEMENTOS = 1000;

// Interfaz del TAD. Pre-declaraciones:

template<typename Elemento> struct Saco;
/* El tipo Elemento requerirá tener las funciones
 * int precio(const Elemento& e);
 * bool operator==(const Elemento& e1, const Elemento& e2);
 */

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);

// Operación auxiliar (interna)
template<typename Elemento> int buscar(const Saco<Elemento>& s, const Elemento& e);
...
```

```
// Declaración
```

```
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:  
    struct Repeticiones {  
        Elemento dato;  
        int numRep;  
    };  
    Repeticiones elementos [MAX_NUM_ELEMENTOS];  
    int total;  
};  
...
```

```

// Implementación de las operaciones
template<typename Elemento> void vacio(Saco<Elemento>& s) {
    s.total = 0;
}

template<typename Elemento> int buscar(const Saco<Elemento>& s, const Elemento& e) {
    ...
}

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) {
    ...
}

#endif

```

[ver implementación completa en el material de clase]

Trabajo encargado

Leer lección 4 de los apuntes y código C++ en la Web

Estructuras de Datos y Algoritmos

TAD genéricos

LECCIÓN 4

© All wrongs reversed – bajo licencia CC-BY-NC-SA 4.0



Universidad
Zaragoza

Dpto. de Informática e Ingeniería de Sistemas
Universidad de Zaragoza, España

Curso 2023/2024

Grado en Ingeniería Informática

UNIVERSIDAD DE ZARAGOZA

Aula 0.04, Edificio Agustín de Betancourt



Adaptadas de diapositivas de Javier Campos