En esta página › Sinopsis

Tema 8: Genericidad.

Genericidad.

- La *genericidad* **no** es una propiedad esencial para que un lenguaje sea considerado OO.
- Podríamos decir que la *genericidad* es otro tipo de polimorfismo.
- Recordad lo que decíamos en las primeras prácticas cuando hablabamos de `List` y `Stack`:

```
typedef char Element; // Hasta que veamos genericidad usaremos este typedef.
```

`Element` era el tipo de los elementos almacenados en *listas* y *pilas*.

- Estaría bien disponer de una característica del lenguaje que nos permitiera hacer esto no solo para un tipo, sino para todos los tipos que queramos.
- De este modo podríamos crear simultáneamente *listas y/o pilas de caracteres*, *de enteros*, *de droides*, etc...

Genericidad y C++.

- **C++** la incorpora mediante el concepto de *plantilla* (*template*) y nos permite aplicar la idea tanto a clases como a funciones.
- Un *template* o plantilla es un mecanismo que permite al programador usar tipos como parámetros para una clase o una función.
- Veamos un ejemplo muy sencillo aplicado a funciones, imaginad que queremos crear una función que intercambie dos datos del mismo tipo, p.e. enteros, cadenas o droides:

```
// [1]                          // [2]
void swap (int& a , int& b) { | void swap (string& a , string& b) {
```

```
// [3]
void swap (Droid& a , Droid& b) {
  Droid t = a;
        a = b;
        b = t;
}
```

- ¿Cómo son todas las funciones anteriores?
- ¿Y si hubiera una forma de escribir el código una sola vez parametrizando el tipo de dato con el que trabajan?

Plantillas de funciones.

- La hay, mediante el uso de *templates* o *plantillas* de funciones, mira:

```
// Así declaramos una plantilla de una función.
// La función todavía no existe.
template <class T>  // Podemos emplear typename en lugar de class
void swap (T& a , T& b) {
  T t = a;
    a = b;
    b = t;
}
```

- Para que la función exista hay que **instanciar** su plantilla, p.e.:

```
int s, t;
string m, n;
...
swap (s, t);                    // swap (int&, int&)
swap (m, n);                    // swap (string&, string&)
```

- Para poder instanciar la plantilla de una función genérica tenemos que tener accesible el código fuente de su declaración.

Plantillas de clases.

- De forma similar a como hemos hecho para funciones podemos hacerlo para clases.
- Si hacemos genérica la clase `Node` empleada en `Tree` :

```
template <typename T>
class Node {                // El tipo de los elementos es T
  public:
    Node (T e);
    ...
  private:
    T _key;
    Tree lefts, rights;
};
```

- La instanciamos así:

```
Node<char> nc;                  // nc variable de tipo Node de caracter
Node<Droid> nd;                 // nd variable de tipo Node de Droid
```

- ¿ Y si *Tree* también es genérica ?
- Entonces la clase *Node* anterior quedaría así:

```
template <typename T>
class Node {                // El tipo de los elementos es T
  public:
    Node (T e);
    ...
  private:
    T _key;
    Tree<T> lefts, rights;      // ¿Declaración o instanciación?
};
```

Parámetros de un *template*.

```
template <typename T, int N> struct Array {
  T elements[N];
  ...
};
Array<char,12> ca;              // Array de 12 caracteres
Array<Droid, 512> da;           // Array de 512 droides
```

- Se permiten valores por defecto:

```
template <typename T = int, int N = 100> struct Array {
  T elements[N];
  ...
};
Array<> ca;                     // Array<int, 100>
Array<Droid> da;                // Array<Droid, 100>
Array<Array<char, 120>> aa;     // Matriz de 100x120 caracteres
```

En C++ previo al estándar de 2011 los ángulos de cierre dobles (\(>>\)) de un template deben estar separados, no pueden ir juntos: **\(> .. >\)**.

Un ejemplo más completo.

```
// -- The C++ programming language, 4th. ed.
////////////////////////
// Contenido del '.h' //
////////////////////////
template<typename T>
struct Link {
  Link* pre;
  Link* suc;
  T val;
};

template<typename T>
class List {
```

```
  ...
  void print_all() const;
};
```

```
//////////////////////
// Contenido del '.cc' //
//////////////////////

// Implementación del método "List<T>::print_all() const"
template<typename T> void List<T>::print_all() const {
  for (Link<T>* p = head; p; p = p -> suc) //p depende de T
  cout << *p;
}
```

Programación genérica.

- Como podrás imaginar esta característica abre las puertas a un nuevo modo de crear programas.
- Ya no solo debes pensar en el código que escribes, sino a qué dará lugar cuando se instancie. *Requiere llevar bastante más cuidado*.
- A principios de la década de los 90 Alexander Stepanov comenzó a desarrollar lo que por aquel entonces se denominó la biblioteca STL y que hoy en día forma parte de la biblioteca estándar de **C++**.
- En esencia *STL* consistía en una colección de contenedores, iteradores, algoritmos y funciones programados bajo el paradigma de la genericidad.
- Obligó a los fabricantes de compiladores a incorporar todo lo que demandaba este nuevo estilo de programación. En cierto modo, *STL* se convirtió en el código de testeo de muchos fabricantes de compiladores de **C++**.
- La genericidad que proporciona **C++** está años luz a la genericidad que podemos encontrar en otros lenguajes (*C#*, *Java*, etc...) donde básicamente consiste en un

- El lenguaje **C++** no es sólo el compilador.
- Es también su biblioteca estándar, gran parte de la cual está, escrita usando genericidad (*STL*).
- En ella nos encontramos, junto a otros componentes, con una serie de clases que implementan *TADS* que representan *contenedores* (listas, pilas, colas, etc...).
- Para usar cualquier componente de esta biblioteca debemos incluir un archivo de cabecera que contiene el código que lo implementa. A continuación (si es un elemento genérico) debemos proceder a instanciarlo, p.e:

```
#include <list>
#include <queue>

std::list<int> il;               // Lista de enteros
std::list<Droid> dl;             // Lista de droides
std::list<std::queue<char>> ql; // Lista de colas de caracteres
std::queue<std::list<char>> lq; // Cola de listas de caracteres
```

- Dispones de más información sobre esta biblioteca aquí.

Ejemplo de uso de std::vector.

```
// Compilar con std=c++11
#include <iostream>
#include <vector>

class Robot {
 public:
  static int rn;
  const static Robot zero;

  Robot(std::string n, float bl = 1.0) {
    theName = n;
```

```
  friend std::ostream& operator<< (std::ostream& os, const Robot& r);

  virtual ~Robot() {}

private:
  std::string theName;
  float theBattLevel;
};
```

```
std::ostream&
operator<< (std::ostream& os, const Robot& r) {
  os << "-----------------\n";
  os << "    Name: " << r.theName << '\n';
  os << "    BL: " << r.theBattLevel<< '\n';
  os << "-----------------\n";

  return os;
}

int Robot::rn = 0;
const Robot Robot::zero = Robot ("", 0.0);

int main(int argc, char *argv[])
{
  std::vector<Robot*> rv;

  for (int i = 0; i < 10; i++)
    rv.push_back (new Robot ("robot-" + std::to_string(Robot::rn),
                             static_cast<float>(.1*i)));


  std::cout << "\nBuilt robots: " << Robot::rn << "\n\n";

  for (int i = 0; i < 10; i++)
    std::cout << *rv[i] << '\n'; // ¿porqué *rv[i] y no rv[i]?

  return 0;
```

Aclaraciones:

- **Este contenido no es la bibliografía completa de la asignatura**, por lo tanto debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.

Página anterior
← Tema 7: Relaciiones entre objetos. Herencia. Polimorfismo. Enlace dinámico.

Siguiente página
Tema 9: Excepciones. Patrón RAII. →