Algoritmos II 2019

Iteradores en C++

Definición

- Un ***Iterador*** es una manera de recorrer elementos de tipo T2, de una instancia de tipo T1.

¿Que?

Definición

- Un ***Iterador*** es una manera de recorrer elementos de tipo **T2**, de una instancia de tipo **T1**.

- **Ej1**: Queremos recorrer los elementos de tipo **T2** de una instancia de **Conjunto(T2)** //**T1**
- **Ej2**: …de tipo **T2** de una instancia de **Arreglo estático(T2)** //**T1**

Motivación

En clase ya vimos maneras de recorrer

Conjunto

$$\begin{aligned} \text{dameUno}(c) \in c &\equiv \text{true} \\ \text{sinUno}(c) &\equiv c - \{\text{dameUno}(c)\} \end{aligned}$$

Arreglo

$$\begin{aligned} &\text{definido}(\text{crearArreglo}(n), m)) \equiv \text{false} \\ &\text{definido}(a\ [\ n\ ] \leftarrow e, m) \equiv n = m \vee \text{definido?}(a, m) \\ &(a\ [\ n\ ] \leftarrow e)\ [\ m\ ] \equiv \textbf{if}\ n = m\ \ \textbf{then}\ \ e\ \ \textbf{else}\ \ a\ [\ m\ ]\ \ \textbf{fi} \end{aligned}$$

Motivación

Cada TAD se recorre de otra manera…

¡Nos gustaría tener una manera uniforme de recorrer!

Enfoque TAD Iterador

TAD Iterador(T1,T2)

Generadores

Inicio: → Iterador(T1,T2)

Avanzar: Iterador(T1,T2) → Iterador(T1,T2)

-Recordar que los elementos son de genero T2

Enfoque TAD Iterador

Observadores

dameActual(T1 it, Iterador(T1,T2)) -> T2

{it tiene al menos un elemento}

Axiomas (para el caso it:Conjunto(T2))

dameActual(t1, inicio) ≡ t1.dameUno()

dameActual(t1, avanzar(it)) ≡
    dameActual(t1.sinUno(), it)

Enfoque TAD Iterador

¿Cuando termina la lista de axiomas?

…

Axiomas (para el caso it:Secuencia(T2))

…

Axiomas (para el caso it:TipoDelUsuario(T2))

Enfoque C++

Cada clase da su propio iterador, pero todas las clases respetan la misma especificación

vector<int> v = {1, 2, 3, 4};

vector<int>::**iterator** it = v.begin(); ← C++

//it es un **Iterador(Vector,int)** ← TAD

Esto soluciona el problema de “la lista infinita” porque la lista se construye por demanda.

Enfoque C++

Como vimos en Templates, notar que cada iterador es de otro tipo:

```
vector<int> v = {1, 2, 3, 4};
vector<int>::iterator it = v.begin();
list<int> l = {1, 2, 3, 4};
list<int>:: iterator it2 = l.begin();
   it = it2;
```

Enfoque C++

List(lista enlazada) es una implementación de Secuencia que veremos la clase siguiente!

Para esta clase hay dos cosas que nos interesan:

Agregar(…): O(1)

Iesimo(…): O(n)

Enfoque C++

Como vimos en Templates, notar que cada iterador es de otro tipo:

```
vector<int> v = {1, 2, 3, 4};
vector<int>::iterator it = v.begin();
list<int> l = {1, 2, 3, 4};
list<int>:: iterator it2 = l.begin();
   it = it2;
```

← Error de tipos!

```
Class '_List_iterator<int>' is not compatible with class 'vector<int, std::allocator<int> >::iterator'
```

Mas motivación

¿Hay forma de recorrer **n** elementos en menos de $O(n^2)$?

```
for (int i=0; i<5; i++){
        j = l.iesimo(i);
}
```

Iterator

Si it es de tipo T::iterator y col de tipo T

*it	Obtiene el elemento actual
It→campo	Equivalente a (*it).campo
++it	Avanza al siguiente elemento
--it	Retrocede
col.begin()	Referencia al 1er elemento de col
col.end()	Referencia al ultimo

Ejemplo1: Recorrer

```
vector<int> v = {1, 2, 3, 4};
vector<int>::iterator it = v.begin();
Int x  = v.begin();
```

Ejemplo1: Recorrer

```
vector<int> v = {1, 2, 3, 4};
vector<int>::iterator it = v.begin();
Int x  = v.begin(); ¡No tipa!
```

¿Que nos falto?

Ejemplo1: Recorrer

```
vector<int> v = {1, 2, 3, 4};
vector<int>::iterator it = v.begin();
Int x  = v.begin();
Int x  = *v.begin();
```

Ejemplo1: Recorrer

```
vector<int> v = {1, 2, 3, 4};
vector<int>::iterator it = v.begin();
while (it != v.end()) {
        cout << *it;
        ++it;
}
for(vector<int>::iterator it = v.begin();
    it != v.end(); ++it){
        cout << *it;
}
```

Ejemplo2: Insertar

```
vector<int> v = {1, 2, 3, 4};
vector<int>::iterator it = v.end();
--it;
v.insert(it, 10);  // 1 2 3 10 4
```

Ejemplo3: auto

Muchas veces el compilador puede inferir los tipos:

```
vector<int> v = {1, 2, 3, 4};
auto it = v.end();
--it;
v.insert(it, 10);
```

¡No abusar de esta funcionalidad!

Ejemplo4: eliminar

```
vector<int> v = {1, 2, 3, 4};
vector<int>::iterator it = v.begin();
it += 2;
v.erase(it); // 1 2 4
```

Mutables vs Inmutables

```
void mostrar(const vector<int>& v) {
        for (vector<int>::iterator it = v.begin();
             it != v.end(); ++it) {
            cout << *it;
        }
}
```

Mutables vs Inmutables

```
void mostrar(const vector<int>& v) {
        for (vector<int>::iterator it = v.begin();
             it != v.end(); ++it) {
            cout << *it;
        }
}
```

```
Class 'vector<int, std::allocator<int> >::const_iterator' is not compatible with class 'vector<int, std::allocator<int> >::iterator'
```

¡El iterador es mutable, pero el vector no!

Mutables vs **Inmutables**

```
void mostrar1(const vector<int>& v) {
        for (vector<int>::const_iterator it = v.begin();
             it != v.end(); ++it) {
            cout << *it;
        }
}
Void main() {
    vector<int> v{1, 2, 3, 4};
    mostrar(v);
}
```

Ejemplo2: Insertar

```
vector<int> v = {1, 2, 3, 4};
vector<int>::iterator it = v.end();
--it;
v.insert(it, 10);  // 1 2 3 10 4
```

¡Mutable!

Typename

-Es para indicar que lo que sigue es un tipo.

-Se relaciona con ***auto**** y es muy útil cuando se usan templates

*dependiendo la versión de C++:

https://www.oreilly.com/library/view/effective-modern-c/9781491908419/ch01.html

Ejemplo5: typename

```
template<class Iterador>
   bool pertenece(Iterador desde, Iterador hasta, typename
   Iterador::value_type& x) {
      for (auto it = desde; it != hasta; ++it) {
         if (x == *it) {   return true;   }
      }
      return false;
}
…
int dos = 2;
cout << pertenece(v.begin(), v.end(), dos);
```

Ejemplo5: value_type

value_type hace que C++ infiera el tipo del iterador, según el tipo de v