ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

Iteradores constantes

Manuel Montenegro Montes
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

Ejemplo

- Volvemos al ejemplo de la suma de una lista de enteros:

```
int suma_elems(ListLinkedDouble<int> &l) {
  int suma = 0;
  for (ListLinkedDouble<int>::iterator it = l.begin();
       it != l.end();
       it.advance()) {

    suma += it.elem();
  }
  return suma;
}
```

const.

le pasamos la referencia NO CONSTANTE.

Acordarnos de que aprendernos esta sintaxis es importante.

No modificamos la lista por lo que sería honesto decir que

¿Qué pasa si ponemos const realmente? Habríamos cometido un error, el compilador se queja. Begin y end no son metodos constantes.

¿Podemos pasar `l` por referencia constante?

- No, porque `begin()` y `end()` no son métodos constantes.

```
int suma_elems(const ListLinkedDouble<int> &l) {
  int suma = 0;
  for (ListLinkedDouble<int>::iterator it = l.begin();
       it ≠ l.end();
       it.advance()) {

    suma += it.elem();
  }
  return suma;
}
```

El compilador se queja.

¿Y si begin( ) y end( ) fueran constantes?

```
template <typename T>
class ListLinkedDouble {
public:
  …
  iterator begin() const;
  iterator end() const;

};
```

- Técnicamente, el compilador no se queja.
- Pero devuelven un `iterator`, a través del cual yo puedo modificar los elementos de la lista.

Aunque no la modifiquen, al devolver un iterador, este puede modificar la lista.

¿Por qué `iterator` puede modificar los elementos de la lista?

```
class iterator {
public:
  void advance();
  T & elem();
  bool operator=(const iterator &other) const;
  bool operator≠(const iterator &other) const;
   ...
};
```

Por esta referencia de aquí.

- Porque `elem()` devuelve una referencia al elemento apuntado por el iterador.
- A partir de esa referencia puedo cambiar el valor de ese elemento.

¿Y si elem( ) devolviera una referencia constante?

```
class iterator {
public:
  void advance();
  const T & elem();
  bool operator=(const iterator &other) const;
  bool operator≠(const iterator &other) const;
   ...
};
```

Si hicieramos esto, begin y end podrían ser constantes.

¿Y si `elem()` devolviera una referencia constante?

- Ya no podría tener programas como este:

```
void multiplicar_por(ListLinkedDouble<int> &l, int num) {
  for (ListLinkedDouble<int>::iterator it = l.begin();
       it ≠ l.end();
       it.advance()) {
    it.elem() = it.elem() * num;
  }
}
```

Modificamos la lista luego no puede ser constante.

Moraleja

- A veces quiero que los iteradores me devuelvan referencias no constantes, porque quiero utilizar un iterador para alterar los elementos de la lista.

```
void multiplicar_por(ListLinkedDouble<int> &l, int num) {
  for (ListLinkedDouble<int>::iterator it = l.begin();
       it =/= l.end();
       it.advance()) {
    it.elem() = it.elem() * num;
  }
}
```

Aquí los altera.

Moraleja

- A veces quiero que los iteradores me devuelvan referencias no constantes, porque quiero utilizar un iterador para alterar los elementos de la lista.
- Otras veces quiero iterar sobre una lista sin modificar sus elementos.

```
int suma_elems(const ListLinkedDouble<int> &l) {
  int suma = 0;
  for (ListLinkedDouble<int>::iterator it = l.begin();
       it =/= l.end();
       it.advance()) {

    suma += it.elem();
  }
  return suma;
}
```

No modificamos la lista.

Moraleja

- A veces quiero que los iteradores me devuelvan referencias no constantes, porque quiero utilizar un iterador para alterar los elementos de la lista.
- Otras veces quiero iterar sobre una lista sin modificar sus elementos.
- Tengo que distinguir dos tipos de iteradores:
  - **Iteradores no constantes** (`iterator`) Podríamos modificar.
  - **Iteradores constantes** (`const_iterator`) No podríamos modificar nada.

Iteradores constantes y no constantes

```
class iterator {
public:
  void advance();
  T & elem();
  bool operator==(const iterator &other) const;
  bool operator≠(const iterator &other) const;
   ...
};

class const_iterator {
public:
  void advance();
  const T & elem();
  bool operator==(const iterator &other) const;
  bool operator≠(const iterator &other) const;
   ...
};
```

Misma clase pero con la diferencia de que una es constante y de que cuando accedemos a un elemento en la normal se puede modificar la lista pero en la const no

Las implementaciones de ambas clases son exactamente iguales

Iteradores constantes y no constantes

```
template <typename T>
class ListLinkedDouble {
public:
  …
  iterator begin();
  iterator end();

  const_iterator cbegin() const;
  const_iterator cend() const;

};
```

no constantes

Funciones no constantes. Devuelven iteradores que me permiten modificar la lista.

Funciones constantes. Garantizan que no puedo modificar la lista con el iterador que devuelvan.

Consecuencias

- Podemos utilizar iteradores para modificar elementos de la lista.
- Para ello utilizamos la clase `iterator`, como siempre.

```
void multiplicar_por(ListLinkedDouble<int> &l, int num) {
  for (ListLinkedDouble<int>::iterator it = l.begin();
       it =/= l.end();
       it.advance()) {
    it.elem() = it.elem() * num;
  }
}
```

Consecuencias

- Podemos utilizar iteradores para recorrer la lista sin modificarla, y así poder declarar el objeto correspondiente como constante.
- Para ello utilizamos la clase `const_iterator`, y los métodos `cbegin()` y `cend()`.

```
int suma_elems(const ListLinkedDouble<int> &l) {
  int suma = 0;
  for (ListLinkedDouble<int>::const_iterator it = l.cbegin();
       it =/= l.cend();
       it.advance()) {

    suma += it.elem();
  }
  return suma;
}
```

No obstante, hay un pequeño problema de implementación

Cuánta duplicación, ¿no?

```
class iterator {
public:
  void advance();
  T & elem();
  bool operator==(const iterator &other) const;
  bool operator≠(const iterator &other) const;
  ...
};
```

U

Utilizaremos un genérico U que pueda ser T y const T

Son idénticas las implementaciones. La única diferencia es elem()

```
class const_iterator {
public:
  void advance();
  const T & elem();
  bool operator==(const iterator &other) const;
  bool operator≠(const iterator &other) const;
  ...
};
```

Sólo difieren en el tipo de retorno de `elem()`!

Cuánta duplicación, ¿no?

- Podemos utilizar las plantillas de C++:

```
template <typename U>
class gen_iterator {
public:
  void advance();
  U & elem();
  bool operator==(const gen_iterator &other) const;
  bool operator≠(const gen_iterator &other) const;
   ...
};
```

En iteradores no constantes: `U = T`
En iteradores constantes: `U = const T`

Cuánta duplicación, ¿no?

- Podemos utilizar las plantillas de C++:

```
template <typename U>
class gen_iterator {
public:
  void advance();
  U & elem();
  bool operator==(const gen_iterator &other) const;
  bool operator≠(const gen_iterator &other) const;
   ...
};

using iterator = gen_iterator<T>;
using const_iterator = gen_iterator<const T>;
```

Las definimos como instancias particulares de esta plantilla

En resumen, tenemos

```
template <typename T>
class ListLinkedDouble {
public:
   ...

   template <typename U>
   class gen_iterator {  ...  }

   using iterator = gen_iterator<T>;
   using const_iterator = gen_iterator<const T>;

   iterator begin();
   iterator end();

   const_iterator cbegin() const;
   const_iterator cend() const;

};
```