Práctica 4: T.D.A. No Lineales

Gustavo Rivas Gervilla

UNIVERSIDAD
DE GRANADA

Objetivos

Iteradores

La Práctica

Objetivos

Iteradores

La Práctica

Objetivos

- Trabajar con la STL.
- Definir iteradores para una estructura de datos.
- Seguir practicando con `templates`.

Objetivos

Iteradores

La Práctica

Iteradores

Un iterador es un objeto que permite recorrer un contenedor. Cada uno de los contenedores de la STL cuenta con un conjunto de iteradores que podemos emplear para recorrer dicha estructura. Tenemos distintos tipos de iteradores:

- **`iterator`**.
- **`const_iterator`**.
- **`reverse_iterator`**.
- **`const_reverse_iterator`**.

Estos iteradores nos sirven para recorrer cualquier contenedor de la STL. Para crearlos seguiremos la sintaxis:

`contenedor::iterator`

Donde ponemos el tipo de contenedor donde queremos iterar y el tipo de iterador que queremos crear. Así tendríamos la sintaxis `map<string, int>::reverse_iterator it;`, para declarar un iterador que recorra de forma no constante lo elemento de un **`map`** cuya clave es una cadena, y su valor un entero, de atrás hacia delante.

ITERADORES

Podemos ver el código de la librería SQL[1] para cada uno de los contenedores disponibles y ver cómo todos tienen dentro de su definición varias clases públicas que son los distintos iteradores disponibles para esa clase. Así por ejemplo, podemos ver el **operator**++ del que disponen los iteradores para pasar al siguiente elemento del contenedor:

```
1  _LIBCPP_INLINE_VISIBILITY
2  __list_iterator& operator++()
3  {
4  #if _LIBCPP_DEBUG_LEVEL >= 2
5    _LIBCPP_ASSERT(__get_const_db()->__dereferenceable(this),
6                                    "Attempted to increment
                                        non-incrementable list
                                        ::iterator");
7  #endif
8    __ptr_ = __ptr_->__next_;
9    return *this;
10 }
```

[1] Un código de la librería STL proporcionado por Google.

Iteradores

Pero no siempre vamos a querer recorrer de forma lineal un contenedor de la STL. Nosotros vamos a crear nuestras propias clases, las cuáles contendrán datos que puede ser interesante recorrer de distintas formas. Pensemos por ejemplo en los distintos órdenes en los que podemos recorrer un árbol. Para ello no nos bastaría con sobrecargar el `operator[]`. Tendremos que definir varios iteradores que partan de la raíz del árbol en este caso, y en cada paso nos den el siguiente nodo del árbol a explorar.

Con lo que aprender a definir nuestros propios iteradores puede resultar muy útil.

Iteradores

Ejemplo

A continuación vamos a ver un ejemplo con el que vamos a:

- Comprender en profundidad la implementación de iteradores.
- Conocer algunos errores comunes.
- Ver una implementación de una plantilla separando la implementación en `.h` y `.cpp`.

Iteradores

Ejemplo

En este ejemplo vamos a implementar el T.D.A. `Collection`, donde se almacenan una serie de pares de valores. Estos pares de valores pueden ser de cualquier tipo. Por ello vamos a emplear un `template` con dos parámetros, igual que el T.D.A. Diccionario con el que tenéis que trabajar.

```
1  #ifndef __COLLECTION__
2  #define __COLLECTION__

4  #include <iostream>
5  #include <utility>
6  #include <vector>

8  using namespace std;

10 template <class T, class U>
```

Iteradores

Ejemplo

```
11  class Collection {
12  private:
13    vector<pair<T, U>> elements;

15  public:
16    Collection();
17    void add(pair<T, U> element);
18    void remove(int pos);
19    int size();
```

Iteradores

Ejemplo

Para añadir iteradores a nuestra clase, seguiremos la filosofía que hay en el desarrollo de la STL. Añadiremos clases públicas a nuestro T.D.A., que serán los distintos iteradores de los que disponga nuestra clase.

```
21    class iterator {
22    private:
23      typename vector<pair<T, U>>::iterator vit;
24      iterator(typename vector<pair<T, U>>::iterator it)
          ;
25      friend class Collection<T, U>;

27    public:
28      iterator();
29      iterator(const iterator &it);
```

Iteradores

Ejemplo

```
31      iterator &operator=(const iterator &it);

33      iterator &operator++();
34      iterator &operator--();
35      pair<T, U> &operator*();

37      bool operator!=(const iterator &it) const;
38      bool operator==(const iterator &it) const;
39    };

41    iterator begin();
42    iterator end();
```

Iteradores

Ejemplo

Ya aquí vemos algunas cosas a destacar. En primer lugar, tenemos que tener claro que un iterador no es más que una clase que se declara dentro del T.D.A., la cual tiene una serie de métodos. Métodos que nos servirán para recorrer nuestra estructura de datos.

En nuestro caso, como nuestro T.D.A. almacena su información en un contenedor de la STL, aprovecharemos la utilidad de los iteradores de estos contenedores para implementar los nuestros. Por ello como atributo de la clase `Collection<T,U>::iterator` tenemos un iterador para un `vector` de la STL.

```
23      typename vector<pair<T, U>>::iterator vit;
```

Iteradores

Ejemplo

```
23      typename vector<pair<T, U>>::iterator vit;
```

En esta misma línea nos encontramos con algo interesante, y es el uso de la palabra clave `typename`. Esto se usa tanto aquí, como en la línea 24, puesto que de no ser así obtendríamos el siguiente error:

```
error: need 'typename' before
'std::vector<std::pair<_T1, _T2> >::iterator' because
'std::vector<std::pair<_T1, _T2> >' is a dependent
scope
```

Este error, a grandes rasgos se produce porque el compilador no sabe a qué nos referimos. Es decir, a la hora de *parsear* o interpretar nuestro código, hay veces que éste le puede resultar ambiguo, y empleando esta palabra reservada le indicamos que cuando ponemos `vector<pair<T, U>>::iterator` nos estamos refiriendo a un tipo, y no a otra cosa.

De todos modos, este error por lo general es fácil de solucionar, ya que como vemos el compilador nos suele decir que necesitamos poner un `typename` en algún lugar determinado. En otras ocasiones puede ser que nos encontremos con errores algo más extraños, y que viendo en qué línea se producen, podamos pensar que lo que ocurre es una ambigüedad como hemos comentado[2].

Una buena idea, a la hora de compilar un programa, es hacerlo en inglés, de este modo, para aquellos errores que obtengamos, los podremos copiar y pegar en Google, y es más fácil encontrar información en inglés que en español. Para ello, en una terminal podemos hacer:

```
LANG=EN make
```

[2] Para más información sobre este error podemos leer esto.

Iteradores

Ejemplo

```
24      iterator(typename vector<pair<T, U>>::iterator it)
            ;
25      friend class Collection<T, U>;
```

En estas líneas también encontramos algo curioso, ¿por qué declaramos este constructor con parámetros en la parte privada de la clase `iterator`?
Lo hacemos así puesto que nosotros queremos controlar cómo se recorre nuestra estructura. Esto lo hacemos impidiendo que se puedan crear iteradores para nuestra estructura que comiencen apuntando a un elemento cualquiera de ella.
Sólo se podrán crear iteradores apuntando al principio o al final de nuestra estructura, con los métodos `begin` y `end` que proporcionamos en la parte pública de nuestro T.D.A.

Iteradores

Ejemplo

```
41    iterator begin();
42    iterator end();
```

Y es en estos métodos donde haremos uso de ese constructor privado, para generar iteradores apuntando a las posiciones de nuestra estructura que nosotros deseamos. Es por eso que la clase `Collection<T,U>` se pone como clase `friend` de la clase `iterator`; para que pueda acceder a esa parte privada.

Iteradores

Ejemplo

Otra cosa que debemos tener en cuenta es que los siguiente operadores unarios se están implementando como operadores prefijo, es decir, se usarán como `++it, --it, *it`.

```
33      iterator &operator++();
34      iterator &operator--();
35      pair<T, U> &operator*();
```

Si quisiéramos implementar la verión postfijo entonces deberíamos añadir otra función, como se explica aquí

Iteradores

Ejemplo

```
44    class const_iterator {
45    private:
46      typename vector<pair<T, U>>::const_iterator vit;
47      const_iterator(typename vector<pair<T, U>>::
            const_iterator it);
48      friend class Collection<T, U>;

50    public:
51      const_iterator();
52      const_iterator(const const_iterator &it);

54      const_iterator &operator=(const const_iterator &it
            );

56      const_iterator &operator++();
```

Iteradores

Ejemplo

```
57      const_iterator &operator--();
58      const pair<T, U> &operator*() const;

60      bool operator!=(const const_iterator &it) const;
61      bool operator==(const const_iterator &it) const;
62    };
```

Aquí nos encontramos otro iterador, el que sería el iterador de acceso constante. Es decir, este es el iterador que usaremos cuando queramos recorrer la estructura en una función `const`, o cuando se itere sobre un `Collection` que se ha pasado como parámetro `const` a una función. Es decir, allí donde por definición no se pueda modificar la estructura recorrida.

Iteradores

Ejemplo

Como vemos la única diferencia con el iterador de acceso no constante se da en el `operator*`, donde ahora indicamos que se trata de un método `const`, y que además el objeto devuelto por él se devuelve de forma que no se pueda modificar:

```
58      const pair<T, U> &operator*() const;
```

Si se usase el iterador de acceso no constante, para recorrer una estructura que no pueda ser modificada, obtendríamos un error como el siguiente:

```
./src/Collection.cpp:75:57: error: passing 'const
Collection<int, int>' as 'this' argument discards
qualifiers [-fpermissive]
```

Iteradores

Ejemplo

Nos está diciendo algo así como que estamos usando un método de acceso “inseguro” en una parte del código que pretende ser “segura”.

Esto también lo tenemos que tener en cuenta a la hora de implementar los métodos que nos dan iteradores constantes apuntando al principio y al final de la estructura, que los llameremos como sigue, en la línea de los métodos de la STL.

```
64    const_iterator cbegin() const;
65    const_iterator cend() const;
```

Ahora estos métodos son declarados como métodos `const`.

Iteradores

Ejemplo

Por último vamos a añadir un iterador algo menos común. Vamos a imaginar que queremos implementar un iterador de modo que se itere la estructura en el orden marcado por la primera componente de cada par almacenada en ella.

Con esto se pretende poner de manifiesto que los iteradores no siempre tienen que recorrer la estructura de una forma lineal desde el primer elemento almacenado en ella hasta el último, sino que pueden seguir otro orden para recorrer la estructura.

```
67    class ordered_iterator {
68    private:
69      typename vector<pair<T, U>>::const_iterator vit;
70      typename vector<pair<T, U>>::const_iterator vbegin
            ;
71      typename vector<pair<T, U>>::const_iterator vend;
```

Iteradores

Ejemplo

```
72      ordered_iterator(typename vector<pair<T, U>>::
            const_iterator vit,
73                       typename vector<pair<T, U>>::
                             const_iterator vitbegin,
74                       typename vector<pair<T, U>>::
                             const_iterator vitend);
75      friend class Collection<T, U>;

77    public:
78      ordered_iterator();
79      ordered_iterator(const ordered_iterator &it);

81      ordered_iterator &operator=(const ordered_iterator
            &it);
```

Iteradores

Ejemplo

```
83      ordered_iterator &operator++();
84      const pair<T, U> &operator*() const;

86      bool operator!=(const ordered_iterator &it) const;
87      bool operator==(const ordered_iterator &it) const;
88    };

90    ordered_iterator obegin() const;
91    ordered_iterator oend() const;
92  };
```

Iteradores

Ejemplo

De aquí sólo merece la pena señalar que en un iterador puede ser necesario almacenar más información que simplemente un iterador al contenedor de la STL que podamos haber empleado para implementar nuestro T.D.A. En este caso, necesitaremos un iterador que apunte al punto en el que se encuentra el iterador en ese momento. Además de dos iteradores que apunten al principio y al final de la estructura, ya veremos en la implementación por qué necesitamos esta información adicional.

```
69      typename vector<pair<T, U>>::const_iterator vit;
70      typename vector<pair<T, U>>::const_iterator vbegin
            ;
71      typename vector<pair<T, U>>::const_iterator vend;
```

Iteradores

Ejemplo

Finalmente, podemos ver que estamos haciendo uso de la instanciación implícita, incluyendo el `.cpp` al final del `.h`.

```
94  template <class T, class U>
95  ostream &operator<<(ostream &os, const Collection<T, U
        > &c);

97  #include "Collection.cpp"

99  #endif
```

Iteradores

Ejemplo

Además, como hemos implementado nuestro T.D.A. de modo que se pueda acceder a la información almacenada en él a través de métodos público, como son los iteradores, no es necesario declarar el `operator<<` como `friend` de la clase `Collection<T,U>` para que pueda acceder a la parte privada de la misma.

Esto es una buena práctica, ya que hay que tratar de hacer el código lo más seguro posible, de modo que sólo se pueda acceder directamente a la parte privada de la clase cuando sea estrictamente necesario.

Iteradores

Ejemplo

A continuación vemos cómo se definen los métodos declarados en el archivo `src/Collection.cpp`:

```
1  template <class T, class U> Collection<T, U>::
       Collection() {}

3  template <class T, class U> void Collection<T, U>::add
       (pair<T, U> element) {
4    elements.push_back(element);
5  }

7  template <class T, class U> void Collection<T, U>::
       remove(int pos) {
8    elements.remove(elements.begin() + pos);
9  }
```

Iteradores

Ejemplo

```
11  template <class T, class U> int Collection<T, U>::size
        () {
12    return elements.size();
13  }
```

Como vemos lo que hacemos es:

- Especificar que `T` y `U` son parámetro de un `template`.
- Y especificamos a qué clase pertenece el método, en este caso dicha clase es la clase `Collection<T,U>`.

Iteradores

Ejemplo

No tenemos que olvidar que las plantillas **no son clases**, se generarán clases a partir de ellas. Así tendremos la clase `Collection<int, int>` o la clase `Collection<string, double>`, por ejemplo.
Y por tanto tendremos los métodos:

- `void Collection<int, int>::add(pair<int, int> element);`
- `void Collection<string, double>::add(pair<string, double> element);`

Clases y métodos que se generarán cuando hagamos sendas **instanciaciones** de nuestra plantilla.

Iteradores

Ejemplo

Veamos cómo se hace lo propio para implementar los métodos de los iteradores:

```
 98  template <class T, class U>
 99  typename Collection<T,U>::const_iterator &
100  Collection<T, U>::const_iterator::operator=(const
        const_iterator &it) {
101    if (this != &it) {
102      vit = it.vit;
103    }

105    return *this;
106  }
```

Iteradores

Ejemplo

- Hemos de indicar que se devuelve un `const_iterator` de la clase `Collection<T,U>`, porque si no el compilador nos dirá que `const_iterator` a secas no nombra a un tipo.
- Y luego habrá que indicar que se está implementando el `operator=` de la clase `const_iterator` de la clase `Collection<T,U>`.

Iteradores

Ejemplo

A continuación vemos cómo aprovechamos la funcionalidad de los iteradores de la STL para implementar la funcionalidad de nuestros iteradores:

```
115 template <class T, class U>
116 typename Collection<T, U>::const_iterator &
117 Collection<T, U>::const_iterator::operator--() {
118   vit--;
119   return *this;
120 }

122 template <class T, class U>
123 const pair<T, U> &Collection<T, U>::const_iterator::
      operator*() const {
124   return *vit;
125 }
```

Iteradores

Ejemplo

Finalmente, vamos a ver cómo se ha implementado el `operator++` del iterador que recorre la estructura siguiendo el orden que marca la primera componente de los pares almacenados en ella. De este modo veremos cómo una cosa es la semántica que tiene este operador desde el punto de vista de los iteradores, y otra cómo se implementa para cada iterador particular dependiendo del comportamiento deseado.

```
181 template <class T, class U>
182 typename Collection<T, U>::ordered_iterator &
183 Collection<T, U>::ordered_iterator::operator++() {
184   typename vector<pair<T, U>>::const_iterator explorer
        ;
185   T current_key = (*vit).first;
186   T current_next = current_key;
187   vit = vend;
```

Iteradores

Ejemplo

```
189     for (explorer = vbegin; explorer != vend &&
            current_next <= current_key;
190           explorer++)
191       if (current_key < (*explorer).first) {
192         vit = explorer;
193         current_next = (*explorer).first;
194       }

196     for (; explorer != vend; explorer++)
197       if ((*explorer).first > current_key && (*explorer)
              .first < current_next) {
198         vit = explorer;
199         current_next = (*explorer).first;
200       }
```

Iteradores

Ejemplo

```
202     return *this;
203   }
```

También podemos ver cómo usamos los iteradores de acceso constante a nuestra estructura para implementar el `operator<<`:

```
248   template <class T, class U>
249   ostream &operator<<(ostream &os, const Collection<T, U
          > &c) {

251     for (typename Collection<T, U>::const_iterator it =
            c.cbegin();
252           it != c.cend(); ++it) {
253       os << '(' << (*it).first << ',' << (*it).second <<
              ')' << endl;
```

Iteradores

Ejemplo

```
254     }
255     return os;
256   }
```

Iteradores

Ejemplo

Aquí tenemos el archivo `src/main.cpp` donde tenemos un ejemplo de uso de esta plantilla:

```
1  #include <iostream>

3  #include "Collection.h"

5  using namespace std;

7  int main(int argc, char *argv[]) {
8    Collection<int, int> cInt;
9    Collection<string, double> cString;

11   for (int i = 1; i < 5; i++) {
12     cInt.add(pair<int, int>(10 * (i * 2 + 1), 10 * (i
           * 2 + 1)));
```

Iteradores

Ejemplo

```
13      cString.add(
14          pair<string, double>(to_string(10 * (i * 2 +
                1)), 10 * (i * 2 + 1)));
15    }

17    for (int i = 1; i < 5; i++) {
18      cInt.add(pair<int, int>(10 * i * 2, 10 * i * 2));
19      cString.add(pair<string, double>(to_string(10 * i
            * 2), 10 * i * 2));
20    }

22    cout << "Recorremos las estructuras de forma lineal.
          " << endl;
23    cout << cInt << endl << cString << endl;
```

Iteradores

Ejemplo

```
25    cout << "Recorremos las estructuras por orden de
          clave." << endl;
26    for (Collection<int, int>::ordered_iterator it =
          cInt.obegin();
27          it != cInt.oend(); ++it)
28      cout << '(' << (*it).first << ',' << (*it).second
            << ')' << endl;


30    cout << endl;


32    for (Collection<string, double>::ordered_iterator it
          = cString.obegin();
33          it != cString.oend(); ++it)
34      cout << '(' << (*it).first << ',' << (*it).second
            << ')' << endl;
```

Iteradores

Ejemplo

```
36      return 0;
37  }
```

Iteradores

Ejemplo

A continuación mostramos el Makefile para compilar este proyecto:

```
1  BIN = bin
2  SRC = src
3  INC = inc
4  OBJ = obj

6  $(BIN)/main: $(OBJ)/main.o
7          echo Creando el ejecutable.
8          g++ $< -o $@

10 $(OBJ)/main.o: $(SRC)/main.cpp $(SRC)/Collection.cpp $
       (INC)/Collection.h
11         echo Creando main.o
12         g++ -g -c -Wall -I./$(INC) -I./$(SRC) $< -o $@
```

Iteradores

Ejemplo

```
14  clean:
15          -rm $(OBJ)/* $(BIN)/*
```

De él hemos de señalar dos cosas:

- En la orden `g++` no pondremos explícitamente el archivo `src/Collection.cpp` ya que si no el compilador tratará de procesarlo como un `.cpp` normal, y obtendremos errores de compilación, como por ejemplo el que nos dice que `Collection` no nombra a un tipo.

Iteradores

Ejemplo

- Además, en la orden para generar el fichero objeto, habrá que especificar que busque los archivos necesarios para incluir, tanto en el directorio `inc` como en el directorio `src`. De este modo podemos hacer un `include` del `cpp` en el `h` sin más que poner su nombre, sin necesidad de poner la ruta relativa desde el directorio `inc` al archivo (`#include ''../src/Collection.cpp''`).

Objetivos

Iteradores

La Práctica

La Práctica

Esta práctica **es individual**. En ella se han de completar dos T.D.A.. Para cada uno de los T.D.A. se ha de hacer lo siguiente:

- Añadir un iterador no-constante y uno constante.
- Para cada iterador como mínimo implementaremos los métodos:
  - Constructor por defecto.
  - Constructor de copias.
  - `operator++`.
  - Operador de asignación.
  - `operator≠`.
- Añadir 4 métodos adicionales a cada uno de los T.D.A.
- Implementar un archivo de prueba donde se prueben los iteradores y estos métodos adicionales. (Los archivos de prueba proporcionados no tiene por qué emplearse.)
- **Todo** estará debidamente documentado con Doxygen.
- Aunque no es obligatorio es recomendable separar las clases en `.h` y `.cpp`.

Objetivos

Iteradores

Ejemplo

La Práctica

El T.D.A. Diccionario

El T.D.A. Guía de Teléfonos

Un Diccionario es una `list` de elementos del siguiente tipo:

```
1 template <class T, class U> struct data {
2   T clave;
3   list<U> info_asoci;
4 };
```

Las claves no se repiten en el Diccionario.
Habrá que añadir las correspondientes funciones `begin` y `end` para los dos iteradores a implementar, ignorando las que aparecen ya implementadas:

```
1    typename list<data<T, U>>::iterator &begin() { datos
         .begin(); }
2    typename list<data<T, U>>::iterator &end() { datos.
         end(); }

4    typename list<data<T, U>>::const_iterator &begin()
         const { datos.begin(); }
5    typename list<data<T, U>>::const_iterator &end()
         const { datos.end(); }
```

La Práctica

T.D.A. Diccionario

- Hay ya muchos métodos implementados.
- Incluso en el archivo `usodiccionario.cpp` tenéis implementados los métodos de E/S para el `Diccionario<string,string>`.

Sugerencia de métodos que podéis añadir al Diccionario:

1. Borrar un elemento por su clave.
2. Unión de Diccionarios. Os podéis inspirar en el `operator+` de la Guía de Teléfonos. Teniendo en cuenta que si la misma clave se encuentra en dos Diccionarios, entonces se fusionarán las informaciones asociadas a dichas claves.
3. Devolver los elementos dentro de un rango de claves: desde “asa” hasta “casa”, por ejemplo.
4. Implementar la diferencia de Diccionarios. Nuevamente os podéis inspirar en el `operator-` de la Guía de Teléfonos.

Objetivos

Iteradores

Ejemplo

La Práctica

El T.D.A. Diccionario

El T.D.A. Guía de Teléfonos

La Práctica

T.D.A. Guía de Teléfonos

Una Guía de Teléfonos almacena elementos en la siguiente estructura:

```
1    map<string, string> datos; // si admites que haya
         nombres repetidos tendrias que usar un multimap
```

Sugerencias de métodos a añadir a este T.D.A.:

1. Intersección de Guías.
2. Modificar el teléfono asociado a un nombre. Habrá que implementar las dos versiones (con y sin elementos repetidos) como se hace en otros métodos ya implementados en la Guía.
3. Devolver los teléfonos de aquellos nombres que comiencen por una letra determinada.
4. Devolver los teléfonos cuyos nombres asociados estén dentro de un rango: entre “Francisco” y “Manuel”, por ejemplo.

Además de estos métodos, es obligatorio implementar los siguientes métodos que se encuentran comentados en la declaración de la clase:

- Constructor por defecto.
- Constructor de copias.
- Destructor.
- Operador de asignación.

¿Alguna pregunta?
Buena semana.