OO Programozás

Iterátorok

Dr. Hatwagner F. Miklós

Széchenyi István Egyetem, Győr

https://github.com/wajzy/GKxB_INTM085

2023. november 18.

Tervezési minta (design pattern)

„Az informatikában programtervezési mintának (angolul Software Design Patterns) nevezik a gyakran előforduló programozási feladatokra adható általános, újrafelhasználható megoldásokat. Egy programtervezési minta rendszerint egymással együttműködő objektumok és osztályok leírása."*

Első jelentős irodalom

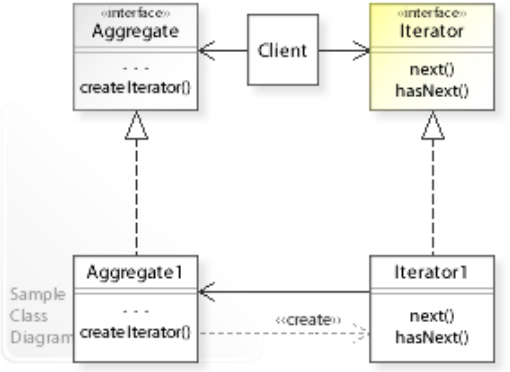
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns (*Elements of Reusable Object-Oriented Software*), Addison-Wesley, 1994

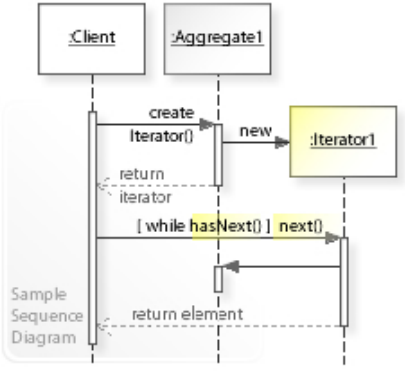
A szerzők által meghatározott kategóriák

- Létrehozási minták
- Szerkezeti minták
- Viselkedési minták

  Iterátor

  „Az Iterátor (...) minta lényege, hogy segítségével szekvenciálisan érhetjük el egy aggregált objektum elemeit, a mögöttes megvalósítás megismerése nélkül.”*


«interface»
Aggregate
- - -
createIterator()
Client
«interface»
Iterator
next()
hasNext()
Aggregate1
- - -
createIterator()
«create»
Iterator1
next()
hasNext()
Sample
Class
Diagram


:Client
:Aggregate1
create
Iterator()
new
:Iterator1
return
iterator
[while hasNext()] next()
return element
Sample
Sequence
Diagram

Forrás

Feladat:

- Készítsünk saját iterátor interfészt, és azt megvalósító tényleges iterátorokat, melyekkel bejárható egy karakterlánc összes betűje, vagy egy láncolt lista elemei!
- A megvalósítás során törekedjünk a C/C++ programozók által jól ismert, mutatókhoz kapcsolódó operátorok alkalmazására!

  `hasNext()`

  `operator!=` → két iterátor ugyanazt az elemet teszi-e elérhetővé?

  `next()`

  `operator++` → iterátor léptetése a következő elemre

  `operator*` → az iterátor által kijelölt elem lekérése

Iterator.h

```
4  template<class T>
5  class Iterator {
6    protected:
7      T* p;
8    public:
9      virtual bool operator!=(const Iterator&) const = 0;
10     virtual Iterator& operator++() = 0; // prefix
11     virtual T& operator*() const = 0;
12 };
```

`operator++`

A postfix alak operátorának felültöltése:

```
virtual Iterator operator++(int) = 0;
```

Probléma: absztrakt osztály nem példányosítható → a visszatérési érték típusa nem lehet `Iterator`!

Message5.h

```
3  #include <cstring>
4  #include <stdexcept>
5  #include "Iterator.h"
6
7  class MessageIterator : public Iterator<char> {
8    public:
9      MessageIterator(char* s) {
10       p = s;
11     }
```

Message5.h

```
13      bool operator!=(const Iterator<char>& it) const override {
14        return p != static_cast<const MessageIterator&>(it).p;
15      }
16
17      MessageIterator& operator++() override {
18        ++p;
19        return *this;
20      }
21
22      char& operator*() const override {
23        return *p;
24      }
25  };
```

`override`

Azt állítjuk, hogy a tagfüggvény (felül)definiálja az öröklött függvényt → ha nem így van (pl. elgépelés), akkor hibaüzenettel leáll a fordítás.

`static_cast`

Az implicit és a felhasználó által definiált típuskonverzió kombinációja, szükség esetén hívja a konverziós konstruktort. *Fordítási időben* ellenőrzi a típusokat és eldönti, hogy az átalakítás végrehajtható-e. Használható primitív típusok közötti átalakításhoz, a származtatási hierarchiában történő fel- és lefelé lépéshez, vagy `void` mutatóról/ra történő konverzióhoz is.

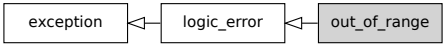
Message5.h

```
27 class Message {
28   private:
29     char* pStr;
30     int len; // The length of str is also stored
31   public:
32     Message() { // default ctor
33       pStr = new char('\0');
34       len = 0;
35     }
36
37     Message(const char* s) { // conversion ctor
38       len = strlen(s);
39       pStr = new char[len + 1];
40       strcpy(pStr, s);
41     }
```

Message5.h

```
43      Message(const Message& m) : Message(m.pStr) {} // copy ctor
44
45      ~Message() { // dtor
46        delete[] pStr;
47      }
48
49      Message& operator=(const Message& m); // assignment op.
50
51      friend std::ostream& operator<<(std::ostream& os, const Message& m);
52
53      char& operator[](int i) {
54        if (i < 0 || i >= len) {
55          throw std::out_of_range("Message::operator[]");
56        }
57        return pStr[i];
58      }
```

std::out_of_range



Message5.h

```
60      int length() const {
61        return len;
62      }
63
64      MessageIterator begin() const {
65        return MessageIterator(pStr);
66      }
67
68      MessageIterator end() const {
69        return MessageIterator(pStr + len);
70      }
71  };
```

Message5.cpp

```
3  Message& Message::operator=(const Message& m) {
4    if(&m == this) return *this;
5    delete[] pStr;
6    len = strlen(m.pStr);
7    pStr = new char[len + 1];
8    strcpy(pStr, m.pStr);
9    return *this;
10 }
11
12 std::ostream& operator<<(std::ostream& os, const Message& m) {
13   os << m.pStr;
14   return os;
15 }
```

LinkedList.h

```
3  #include <iostream>
4  #include "Iterator.h"
5
6  template<class T>
7  struct ListItem {
8    T value;
9    ListItem<T>* next;
10 };
11
12 template<class T>
13 std::ostream& operator<<(std::ostream& os, const ListItem<T>& li) {
14   os << li.value;
15   return os;
16 }
```

LinkedList.h

```
18 template<class T>
19 class ListIterator : public Iterator<ListItem<T>> {
20   public:
21     ListIterator(ListItem<T>* i) {
22       // use 'this' to force the compiler to look
23       // for the name 'p' in the base class
24       this->p = i;
25       // Iterator<ListItem<T>>::p = i; // also OK
26     }
27
28     bool operator!=(const Iterator<ListItem<T>>& it) const override {
29       return this->p != static_cast<const ListIterator<T>&>(it).p;
30     }
```

LinkedList.h

```
32      ListIterator<T>& operator++() override {
33        this->p = this->p->next;
34        return *this;
35      }
36
37      ListItem<T>& operator*() const override {
38        return *(this->p);
39      }
40  };
```

LinkedList.h

```
42 template<class T>
43 class LinkedList {
44   private:
45     ListItem<T>* front;
46     ListItem<T>* tail;
47   public:
48     LinkedList() {
49       front = tail = nullptr;
50     }
```

LinkedList.h

```
52      ~LinkedList() {
53        for(ListItem<T>* current = front; current != nullptr;) {
54          ListItem<T>* next = current->next;
55          delete current;
56          current = next;
57        }
58      }
```

LinkedList.h

```
60      void append(const T& i) {
61        ListItem<T>* latest = new ListItem<T>;
62        *latest = { i, nullptr };
63        if(tail == nullptr) {
64          front = latest;
65        } else {
66          tail->next = latest;
67        }
68        tail = latest;
69      };
```

LinkedList.h

```
71      ListIterator<T> begin() {
72        return ListIterator<T>(front);
73      }
74
75      ListIterator<T> end() {
76        return ListIterator<T>(nullptr);
77      }
78  };
```

iteratorMain.cpp

```
1  #include <iostream>
2  #include "Message5.h"
3  #include "LinkedList.h"
4
5  int main() {
6    Message m = "Hello␣C++␣world!";
7
8    // using iterator
9    for (auto i = m.begin(); i != m.end(); ++i) {
10     std::cout << *i;
11   }
12   std::cout << std::endl;
```

iteratorMain.cpp

```
14    // range-based for loop, C++11
15    for (const auto& c : m) {
16      std::cout << c;
17    }
18    std::cout << std::endl;
19
20    // operator[]
21    try {
22      for (auto i = 0; i <= m.length(); i++) {
23        std::cout << m[i];
24      }
25    } catch(const std::out_of_range& e) {
26      std::cerr << "\nException caught: " << e.what() << std::endl;
27    }
```

iteratorMain.cpp

```
29    LinkedList<int> l;
30    l.append(1); l.append(2); l.append(3);
31    for (auto i = l.begin(); i != l.end(); ++i) {
32      std::cout << *i << '\t';
33    }
34    std::cout << std::endl;
35  }
```

Kimenet

```
Hello C++ world!
Hello C++ world!
Hello C++ world!
Exception caught: Message::operator[]
1 2 3
```

Range-based for loop (C++11)

- Bejárhatók vele tömbök,
- iterátort (`begin()`, `end()`) biztosító *gyűjtemények* (ld. később),
- és kapcsos zárójelek között felsorolt értékek.
- A soron következő elem elérhető érték szerint és referenciával is.

range.cpp

```
3  int main() {
4    int array[] = {1, 2, 3};
5    for(const auto& i : array) { // reference
6      std::cout << i << ' ';
7    }
8    std::cout << std::endl;
9
10   for(auto i : array) { // value
11     std::cout << i << ' ';
12   }
13   std::cout << std::endl;
14
15   for(auto i : {4, 5, 6}) { // braced-init-list
16     std::cout << i << ' ';
17   }
18   std::cout << std::endl;
```

range.cpp

```
20    std::string text = "C++ is so cool!\n";
21    // explicit iterator usage
22    for(auto i=text.begin(); i!=text.end(); ++i) {
23      std::cout << *i;
24    }
25
26    // implicit iterator usage, range-based for loop
27    for(const auto& c : text) {
28      std::cout << c;
29    }
30
31    // overloaded [] operator
32    for(size_t i=0; i<text.length(); ++i) {
33      std::cout << text[i];
34    }
```

Az `std::size_t` típus

- Előjel nélküli egész típus, ami tetszőlegesen nagy objektum méretét ($\rightarrow$ `sizeof`) képes megadni bájtokban mérve.
- Általában indexelésre és ciklusszámlálóként használják az ilyen típusú változókat.

C++ iterátorok

„Iterátor bármely olyan objektum, amely adatok (például egy tömb vagy egy gyűjtemény) valamely elemére mutatva operátorok egy halmazának (ami legalább a növelő (++) és indirekció * műveleteket tartalmazza) segítségével képes az adatok között iterálni.”*

A legkézenfekvőbb iterátor típus a *mutató*, de az összetett adatszerkezetek általában bonyolultabb megoldásokat igényelnek..

Minden iterátor közös jellemzői:

- Azonos típusú értékből vagy referenciából másolással létrehozható (copy constructible).
- Azonos típusú érték vagy referencia hozzárendelhető (copy assignable).
- Megsemmisíthető (destructible), azaz skalár típus (felültöltés nélkül is értelmezhető rajta az összeadás művelet) vagy elérhető destruktorral rendelkező osztály, melynek minden nem statikus tagja is megsemmisíthető.
- Értelmezett rajta a növelés (++) operátor (prefix és suffix alakban is).

Kategóriák:

Input
: Csak olvasási célra, azaz *jobbértékként* indirekcióval (*, ->) elérhető a mutatott adat, támogatja az egyenlőségi (==, !=) operátorokat.

Output
: Csak írási célra, azaz *balértékként* elvégezhető rajta az indirekció, majd a hozzárendelés.

Forward
: Rendelkezik az input iterátorok képességeivel, és ha módosítható, akkor az output iterátorokéval is. Paraméterek vagy inicializáló értékek nélkül is létrehozható (default constructible). A szabványos gyűjtemények legalább ezt a típust megvalósítják.

Bidirectional

A Forward iterátorok képességein túl a csökkentés (--) operátort is támogatja, bejárás hátrafelé.

Random access

Mint Bidirection, de támogatja még az összeadást (+, +=), kivonást (-, -=), az egyenlőtlenségi relációkat (<, <=, >, >=) és az indexelést ([]) is.

range.cpp

```
36    // random access iterator
37    auto it = text.begin();
38    for(size_t i=0; i<text.length(); ++i) {
39      std::cout << it[i];
40    }
41  }
```