Paradygmaty programowania

Paradygmaty programowania

Dr inż. Andrzej Grosser

Częstochowa, 2013

Spis treści

1. Zadanie 2

1.1. Wprowadzenie

Iterator jest pośrednikiem w dostępie do elementów obiektu złożonego (np. elementów tablicy, listy), umożliwia ukrycie reprezentacji wewnętrznej tego obiektu. Takie podejście pozwala uprościć interfejs klasy agregującej elementy, ponadto pozwala na zmianę sposobów przechodzenia po elementach (np. poprzez zdefiniowanie hierarchii klas).

Lista jednokierunkowa to dynamiczna struktura danych, której elementy przechowują oprócz jakieś danej zawiera również wskaźnik pokazujący na następny element. W przypadku listy elementy mogą być rozrzucone w pamięci, nie muszą być umieszczone w jednym ciągłym bloku tak jak w przypadku tablicy. Takie podejście daje możliwość łatwego wstawiania elementów do listy bez potrzeby przesuwania wszystkich kolejnych elementów. Rozważmy następującą implementację tej struktury danych:

```
template <typename T>
class SingleList {
    class SingleListNode {
    public:
        explicit SingleListNode(const T& elem = T()) : value(elem)
            , next(nullptr)
        {}
        ~SingleListNode() {
            delete next;
        }
        SingleListNode* add(const T& elem = T()){
            SingleListNode *tmp = new SingleListNode(elem);
            next = tmp;
            return tmp;
        }
        friend class Iterator;
        friend class SingleList;
    private:
        T value;
        SingleListNode* next;
    };
```

```
public:
    class Iterator {
    public:
        Iterator() : node(nullptr) {}
        Iterator(SingleListNode* n) : node(n){}
        bool operator!=(const Iterator& i) const {
            return node != i.node;
        }
        Iterator& operator++() {
            node = node->next;
            return *this;
        }
        Iterator operator++(int) {
            Iterator iter(*this);
            ++(*this);
            return iter;
        }
        T& operator*() const {
            return node->value;
        }
        T* operator->() const {
             return &node->value;
        }
    private:
        SingleListNode* node;
    };
    SingleList() : head(nullptr),last(nullptr)
    {}
    ~SingleList() {
        delete head;
    }
    SingleList(T* b, T* e) : head(nullptr),last(nullptr){
        while(b != e) {
            push_back(*b++);
        }
    }
    SingleList(const SingleList&) = delete;
    SingleList& operator=(const SingleList&) = delete;
    void push_back(const T& elem) {
         if(!head) {
             head = new SingleListNode(elem);
             last = head;
         } else {
             last = last->add(elem);
```

```
    }
    }
    Iterator begin() {
        return Iterator(head);
    }
    Iterator end() {
        return Iterator(last->next);
    }
private:
    SingleListNode* head;
    SingleListNode* last;
};
```

Z następującym przykładem użycia tego szablonu:

```
int main()
{
    int tab[] = {2,3,5,7,11};
    SingleList<int> lst(tab, tab + 5);
    for(auto iter = lst.begin(); iter != lst.end(); ++iter) {
        std::cout << *iter << std::endl;
    }
    for(auto iter = lst.begin(); iter != lst.end(); ++iter) {
        std::cout << *iter << std::endl;
     }
     return 0;
}
```

Klasa `SingleList` zawiera dwa wskaźniki head i last pokazujące odpowiednio na początek listy i na ostatni element listy zawierający wartość (ułatwia to wstawianie nowych elementów do listy). Implementuje także metody zwracające iterator do pierwszego elementu listy (begin()) i ostatniego elementu za listą (end()). Z ciekawostek przedstawionego kodu należy zwrócić uwagę na:

```
SingleList(const SingleList&) = delete;
SingleList& operator=(const SingleList&) = delete;
```

Informuje to, że nie będzie implentowany ani operator przypisania ani konstruktor kopiujący (jest to element z nowego standardu C++, jeżeli kompilator będzie zgłaszał zastrzeżenia to wystarczy przenieść deklaracje tych metod do sekcji prywatnej, oczywiście bez = delete). Słowo nullptr jest odpowiednikiem 0 i stałej NULL z wcześniejszych wersji standardu (należy zastąpić zerem lub stałą NULL, jeżeli kod się nie kompiluje).

Klasa `SingleListNode` z kolei implementuje kolejne elementy listy (węzły listy). Zawiera wartość danej (value) a także wskaźnik na następny element (typu SingleListNode; to wskaźnik, więc kompilator zna jego rozmiar). Klasa `SingleListNode` implementuje jedynie operację wstawiania elementu za bieżący węzeł (add()).

Najciekawszym elementem jest przedstawionego kodu jest klasa iteratora ukrywa ona sposób dostępu do kolejnych węzłów listy, dopiero za jej pośrednictwem można się odwołać do konkretnych wartości znajdujących się na liście. Dlatego są implementowane operatory wyłuskania i dostępu do składowej. Ponieważ iterator będzie służył, oprócz dostępu do elementów, do przechodzenia po kolejnych węzłach listy dlatego są zaimplementowane operatory inkrementacji, operator różny (operator dekrementacji nie implementuje się tutaj ze względu na to, że da się poruszać jedynie do przodu, wskaźnik w węźle pokazuje jedynie na następny węzeł). Popatrzmy teraz na implementację obydwu operatorów inkrementacji:

```
Iterator& operator++() {
    node = node->next;
    return *this;
}
Iterator operator++(int) {
    Iterator iter(*this);
    ++(*this);
    return iter;
}
```

Pierwszy z nich jest operatorem preinkrementacji (++iter) natomiast drugi jest operatorem postinkrementacji. Kompilator rozróżnia obie wersje na podstawie przekazywanego argumentu (int). Implementacja operatora postinkrementacji korzysta z operatora preinkrementacji, chodzi o zapewnienie symetrii tych operatorów (żeby każdy działał według tego samego schematu i ewentualne zmiany kodu ograniczały się tylko do jednego miejsca.

1.2. Wskazówki do zadania

Implementacja klasy Table jest prostsza od pokazanej wcześniej klasy SingleList implementującej listę jednokierunkową. Wymagane jest tutaj jedynie pomysł co będzie przechowywał wewnątrz iterator, na czym się będzie przesuwał pokazując kolejne elementy (w tym przypadku iterator będzie mógł przesuwać się do przodu i do tyłu). Małą podpowie-

dzią z mojej strony jest, że kolejne elementy w tablicy zajmują ciągły obszar w pamięci, proszę też sobie przypomnieć o zamiennym stosowaniu pewnych elementów w C++ (nie ma potrzeby implementacji klasy TableNode). Innymi słowy przypomnijcie sobie w jaki sposób dostęp do elementów tablicy był realizowany za pośrednictwem wskaźników. Tutaj będzie zachowana duża analogia, przy czym iterator będzie można powiedzieć dodatkową warstwą abstrakcyjną.