

Коллекции и итераторы

Коллекции

- *Коллекция* — объект, содержащий в себе группу однотипных объектов, подлежащих стереотипной обработке.
- Для обращения к конкретному элементу коллекции могут использоваться различные методы, в зависимости от ее логической организации
- Допускается выполнение операций над коллекциями в целом

Линейный односвязный список

```
struct el{  
    int item;  
    el* next;  
};
```

```
class List{
public:
    class Error: public exception{
        public:
            Error (): exception() { }
            void what() {
                cout<<"В списке нет элементов"<<endl;
            }
    };
private:
    el* head;
    el* tail;
    Error err;
```

```
public:  
List() {  
    head=nullptr; tail =nullptr;  
}  
List (const List& s);  
~List();  
bool isEmpty() const;  
void inHead(int val);  
void inTail(int val);
```

```
int getFirst()const;
int getLast()const;
void delFirst();
void delLast();
friend ostream& operator<<(ostream & os,
                           const List& s);
//в полной реализации могут быть еще методы
```

Конструктор копии

```
List::List (const List& s)
{
    head=nullptr; tail =nullptr;
    el* q=s.head;
    while (q)
    {
        inTail(q->item);
        q=q->next;
    }
}
```

Деструктор

```
List::~~List()
{
    el* cur;
    while (head)
    {
        cur=head;
        head=head->next;
        delete cur;
    }
    tail =head=nullptr;
}
```


Список пуст?

```
bool List::isEmpty() const
{
    return (head==nullptr); // return !head;
}
```

Добавить в начало

```
void List :: inHead ( int val )
{
    el* t = new el;
    t->item=val;
    t->next=head;
    if (!head)
        tail =t;
    head=t;
}
```

Добавить в конец

```
void List::inTail ( int val )  
{  
    el* t = new el;  
    t->item = val;  
    t->next=0;  
    if (head) {  
        t->next = t; tail = t;  
    }  
    else {  
        head = tail =t;  
    }  
}
```

Первый/последний элемент

```
int List::getFirst() const
{
    if (head)    return head->item;
    else        throw err;
}
int List::getLast()const
{
    if (head)    return tail ->item;
    else        throw err;
}
```

Пример вызова

```
List s1;
```

```
try{
```

```
    cout << s1.getFirst()<<endl;
```

```
}
```

```
catch (List::Error e)
```

```
{
```

```
    e.what();
```

```
}
```

Удалить первый

```
void List::delFirst()
{
    if (head){
        el *t=head;
        head=head->next;
        delete t;
        if (!head) tail = nullptr;
// был только один в списке
    }
    else throw err;
}
```

Удалить последний

```
void List::delLast() {  
    if (head){  
        if (head== tail) {delete tail; head=nullptr;}  
        else {  
            el *t=head;  
                while (t->next != tail)    t=t->next;  
            delete tail;  
            tail =t;  t->next=nullptr;  
        }  
    }  
    else throw err;  
}
```

Выдать в поток

```
ostream& operator<<(ostream & os, const List& s)
{
    el *p=s.head;
    while(p) {
        os<<p->item<<" ";
        p=p->next;
    }
    os<<endl;
    return os;
}
```


Итераторы

- *Итератор* — объект, позволяющий программисту перебирать все элементы коллекции без учета особенностей ее реализации
- В простейшем случае итератором в низкоуровневых языках является указатель
- Операцию индексирования можно считать примитивной формой итератора

Цели введения итераторов

- Возможность обращаться к любому элементу коллекции при этом скрывая ее внутреннюю структуру
- Позволяет организовать хранение элементов любым способом, давая возможность работы с ними как с простой последовательностью или списком

Возможности итераторов

- доступ к элементу коллекции
- перебор элементов коллекции
- дополнительные операции с коллекцией или различные варианты поведения

- Если коллекция представляет собой структуру, основанную на указателях, итератор может быть реализован в виде дополнительного поля – текущего указателя

Например

```
struct el {  
    int item;  
    el* next;  
};
```

```
class list {  
    private:  
        el* head  
        el* tail;  
        el * cur;
```

```
    . . .  
};
```

Недостатки

- следует заботиться о его состоянии при любых модификациях коллекции, даже если его не придется использовать
- такой встроенный итератор обычно только один, а могут возникнуть задачи, в которых потребуется наличие двух, трех или даже большего количества итераторов

Итераторы-объекты

- Имеют преимущество по сравнению с встроенным текущим указателем, т.к. позволяют создавать в любой момент работы с коллекцией любое количество итераторов, действующих независимо
- Позволяет вынести рабочий указатель за пределы коллекции, но при этом дает возможность перемещаться по объектам, содержащимся в коллекции

- Как правило, итератор имеет операцию доступа к элементу коллекции по ссылке (*)
- Для итераторов предусматриваются также операции перемещения вперед и назад по коллекции(++ , --)
- операции == и != обычно перегружаются для проверки равенства итераторов

Итератор для списка

```
class ListIterator {
private:
    const List *collection;
    el *cur;
public:
    ListIterator(const List *s, el *e) : collection(s), cur(e){ }
    const int &operator *()
    {
        return cur->item;
    }
    ListIterator operator++(); //префиксный ++
    int operator == (const ListIterator &ri) const;
    int operator != (const ListIterator &ri) const;
    friend class List;
};
```

Реализация методов

```
Listlterator Listlterator:: operator++() {  
    cur = cur->next;  
    return *this;  
}
```

```
int Listlterator:: operator==(const Listlterator &ri) const {  
    return ((collection == ri.collection) && (cur == ri.cur));  
}
```

```
int Listlterator:: operator!=(const Listlterator &ri) const {  
    return !(*this==ri);  
}
```

- В классе-коллекции для использования итератора обычно создаются две функции:
iterator begin() – инициализация итератора ссылкой на первый элемент коллекции;
iterator end() – значение, сообщающее, что итератор достиг конца коллекции.

Аналогия с диапазонами массивов !!!

```
ListIterator List::begin() const
{
    return ListIterator (this , head);
}
```

```
ListIterator List::end() const
{
    ListIterator iter(this , nullptr );
    return iter;
}
```

- Метод `end()` нельзя заменить простым сравнением указателя с `nullptr`, т.к. в операциях `==` и `!=` для итератора прежде всего проверяется, относятся ли два итератора к одному и тому же списку

Пример использования

```
List s1;  
ListIterator it = s1.begin();  
while (it != s1.end())  
{  
    cout << *it<<endl;    ++it;  
}
```

или

```
for (ListIterator it = s1.begin(); it != s1.end(); ++it)  
    cout<<*it <<endl;
```