Коллекции и итераторы

2013

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 отдельных операций над коллекциями в целом

Линейный односвязный список

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

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

```
public:
List() {
      head=0; tail =0;
 }
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=0; tail =0;
    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=NULL;
}
```

Список пуст?

```
bool List::isEmpty() const
{
  return (head==0); // return !head;
}
```

Добавить в голову

```
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=NULL;
  if (head) {
    end->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;
}
```

Удалить первый

```
void List::delFirst()
{
  if (head){
    el *t=head;
    head=head->next;
    delete t;
  }
  else    throw err;
}
```

Удалить последний

```
void List::delLast()  {
  if (head){
     if (head== tail) {delete tail; head=0;}
     else  {
              el *t=head;
                    while (t->next != tail)    t=t->next;
                delete tail;
                tail =t;   t->next=0;
       }
  }
  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;
 }
```

Пример вызова

```
List s1;

try{
        cout << s1.getFirst()<<endl;
}
 catch (List::Error e)
{
        e.what();
}
```

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 контейнера при сокрытии внутренней структуры контейнера от пользователя
- Позволяет контейнеру хранить элементы любым способом при допустимости работы с ним как с простой последовательностью или списком

Возможности итераторов

- Указание одного отдельного элемента в коллекции объектов (доступ к элементу)
- Изменение своего значения так, чтобы указывать на следующий элемент (перебор элементов)
- Могут поддерживать дополнительные операции или определять различные варианты поведения

Типы итераторов

- однонаправленные
- обратные (реверсные)
- двунаправленные итераторы
- итераторы ввода и вывода
- константные итераторы (защищающие контейнер или его элементы от изменения)

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

```
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
- Обычно итераторы реализуются через дружественные классы

- 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
- Для того чтобы коллекция могла использовать итератор, он должен объявить класс коллекции дружественным

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

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

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

int ListIterator:: operator!=(const ListIterator &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,NULL);
  return iter;
}
```

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

- Кроме того, в классе-коллекции имеется доступ ко всем приватным полям и методам итератора, поскольку он является дружественным классом к классу итератора

```
class IterException { };

ListIterator List::next(ListIterator it)
            throw (IterException)
{
  if (it.collection != this)
              throw IterException();
  if (it == end())
              throw IterException();
  return (++it);
}
```

Пример использования

```
List s1;
ListIterator it = s1.begin();

try {
    while (it != s1.end())
{
      cout << *it<<" ";
      it=s1.next(it);     //  или   ++it;
    }
  }
  catch (IterException){
  //обработка исключения IterException
  }
```