Коллекции и итераторы

Коллекции

- *Коллекция* — объект программы, содержащий в себе набор значений одного или различных типов и позволяющий обращаться к этим значениям.
-
- Обычно коллекции используются для хранения групп однотипных объектов, подлежащих стереотипной обработке.

- Для обращения к конкретному элементу коллекции могут использоваться различные методы, в зависимости от её логической организации

- Допускается выполнение отдельных операций над коллекциями в целом

Итераторы

- *Итератор* — объект, позволяющий программисту перебирать все элементы коллекции без учета особенностей её 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/предыдущий 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)

Встроенные итераторы

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

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

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

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

- Итераторы-объекты имеют преимущество по сравнению со встроенным текущим указателем, так как позволяют создавать для одной коллекции любое количество итераторов, действующих независимо.

- Таким образом, итератор позволяет вынести рабочий указатель за пределы коллекции, но при этом даёт возможность перемещаться по объектам, содержащимся в коллекции.

- Обычно итераторы реализуются через дружественные классы

Операции над итераторами

- Как правило, итератор имеет операцию доступа к элементу коллекции по ссылке. Такая операция может быть реализована путём перегрузки операции разыменования *

- Для итераторов предусматриваются также операции перемещения вперёд и назад по коллекции объектов. Чаще всего для этого перегружаются операции ++ и —

- Кроме того, операции = = и ! = обычно перегружаются для проверки равенства итераторов.

- 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 !!!

Класс список List содержит элементы, описываемые структурой node

```
struct node{
  int data;
  node* next;
  node* prev;
  node(int data, node* next, node* prev) {
    this->data = data;
    this->next = next;
    this->prev = prev;
  }
};
```

- Для упрощения в дальнейшем вывода элементов списка перегружена операция вывода в поток.

- Для структуры не требуется объявлять её дружественной.

```
ostream & operator<<(ostream & out, const node& X) {
  out << X.data;
  return out;
}
```

Предварительное объявление класса итератора

- Чтобы в определении класса List операции begin() и end() могли возвращать объекты типа итератор списка, нужно сделать предварительное объявление класса итератора

```
class listIterator;
```

- Это связано с рекурсией в определении классов коллекции List и итератора listIterator.

Определение класса List

```
class List{
public:
  class Error {
  public:
    void what(){
      cout << "List is empty" << endl;
    }
  };
 List() {
    head = nullptr; tail = nullptr;
  }
  List(const List& l);
  ~List();
```

```
bool isEmpty() const;
 void inHead(int val);
 void inTail(int val);
 int getFirst()const;
 int getLast()const;
 void delFirst();
 void delLast(); listIterator begin() const;
 listIterator end() const;
 friend ostream& operator<<(ostream & os, const List& l);
 //в полной реализации могут быть еще методы
```

```
private:
  node* head;
  node* tail;
  Error err;
};
```

Определение класса итератора для списка

```
class listIterator {
public:
  class Error {
  public:
    void what(){
      cout << "Iterator error" << endl;
    }
  };
private:
  const List *collection;
  node *cur;
```

```
public:
  listIterator(const List *s, node *e) :collection(s), cur(e){}
   int operator *(){
    return cur->data;
  }
  listIterator operator++(); //префиксный ++
  listIterator operator--(); //префиксный --
  int operator == (const listIterator &ri) const;
  int operator != (const listIterator &ri) const;
};
```

Реализация класса list для двусвязного списка

```
List::List(const List& l) {
  head = 0; tail = 0;   node* q = l.head;
  while (q)   {
    inTail(q->data);
    q = q->next;
  }
}

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

```
bool List::isEmpty() const {
  return (head == nullptr);
}
void List::inHead(int val){
  node* t = new node(val,head,nullptr);
  if (!head)    tail = t;
     else     head->prev = t;
  head = t;
}
void List::inTail(int val){
  node* t = new node(val, nullptr,tail);
  if (head){
    tail->next = t;     tail = t;
  }
  else   head = tail = t;
}
```

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

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

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

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

Реализация класса listIterator для итератора двусвязного списка

```
listIterator listIterator:: operator++() {
  if (cur) {
    cur = cur->next;
    return *this;
  }
  else throw Error();
}
```

```
listIterator listIterator:: operator--() {
  if (cur) {
    cur = cur->prev;
    return *this;
  }
  else throw Error();
}
```

```
int listIterator:: operator==(const listIterator &ri) const {
  return ((collection == ri.collection) && (cur == ri.cur));
}
int listIterator:: operator!=(const listIterator &ri) const {
  return !(*this == ri);
```

```
listIterator List::begin() const {
  return listIterator(this, head);
}
listIterator List::end() const {
  listIterator iter(this, nullptr);
  return iter;
}
```

Метод end()

- 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();

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

Нахождение суммы элементов списка, расположенных между двумя итераторами

```
int sum(listIterator b, listIterator e) {
  int sum = 0;
  while (b != e){
    sum+= *b;
    ++b;
  }
  return sum;
}
```

Нахождение итератора, ссылающегося на максимальный элемент списка

```
listIterator max(listIterator b, listIterator e) {
  listIterator maxx = b;
  while (b != e){
    if (*b> *maxx)
      maxx = b;
    ++b;
  }
  return maxx;
}
```

Вывод в обратном порядке элементов списка, расположенных между двумя итераторами

```
void reverseprint(listIterator be, listIterator b){
  while (be!=b) {
    cout << *be << ' ';
    --be;
  }
  cout  << endl;
}
```

```
int main(){
  List l1;
  for (int i = 0; i < 5;++i)
    l1.inHead(i);
  for (int i = 5; i < 10; ++i)
    l1.inTail(i);
  List l2(l1);
  cout << " list \n" << l2 << endl;
  cout << sum(l2.begin(), l2.end())<<endl;
  listIterator mt = max(l2.begin(), l2.end());
  cout << *mt << endl;
  reverseprint(mt,l2.begin());
return 0;
}
```