Программирование на C++. Часть 2
Лекция 14

ПМИ Семестр 2

Демяненко Я.М.

2026

Реализация итератора для двусвязного списка

Задача.
Реализовать линейный двусвязный список и итератор для него с возможностью перемещения в двух направлениях.

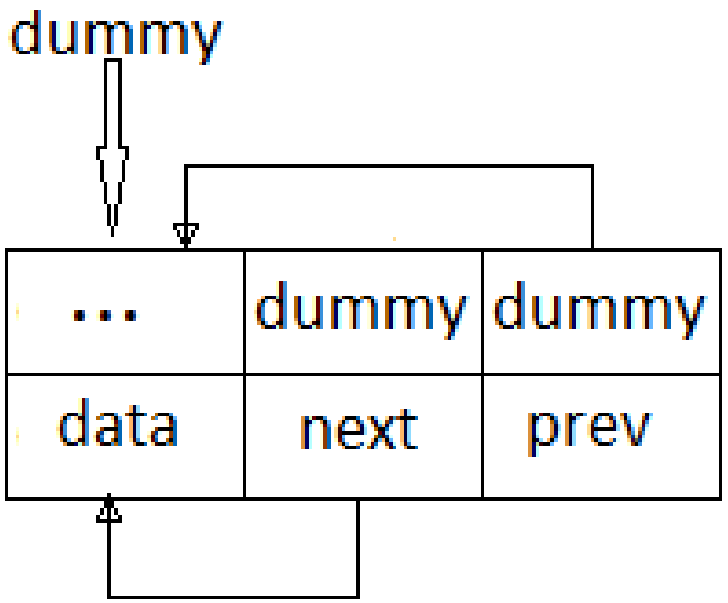
Для корректной реализации потребуется пересмотреть организацию списка и откорректировать реализацию итератора.

Структура узла двусвязного списка

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

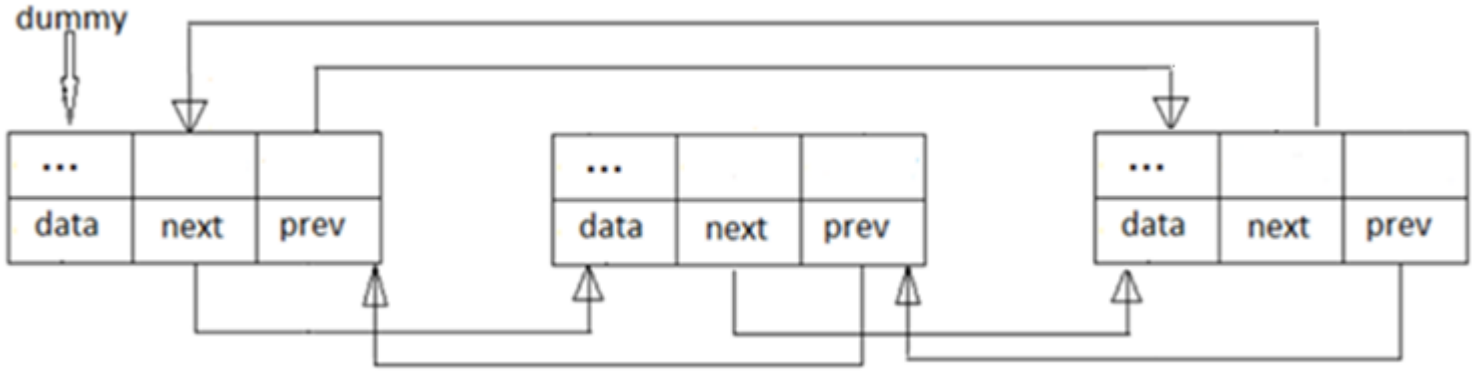
```
ostream& operator<<(ostream& out, const node& X) {
    out << X.data;
    return out;
}
```

Фиктивный элемент в случае пустого списка (dummy head node)



Он всегда существует, даже если список пуст.

В этом случае первый элемент в действительности является вторым.



В таком списке поля указателей не могут принимать значения nullptr.

Ссылка prev первого элемента списка указывает на заглавный узел dummy, а ссылка next последнего элемента тоже ссылается на заглавный узел dummy.

Признаком последнего элемента является тот факт, что next элемента равен dummy, а dummy->prev указывает на последний элемент списка.

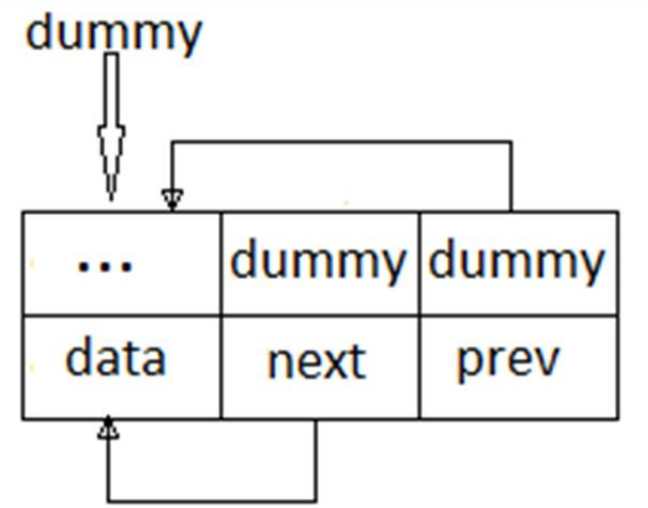
Следовательно мы можем корректно определить операцию operator – –().

Изменения в организации списка List

```
class listIterator;
class List {
public:
   class Error {
   public:
      void what() {
         cout << "List is empty" << endl;
      }
   };
 List();
 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 *dummy;
   Error err;
   friend class listIterator;
};
```

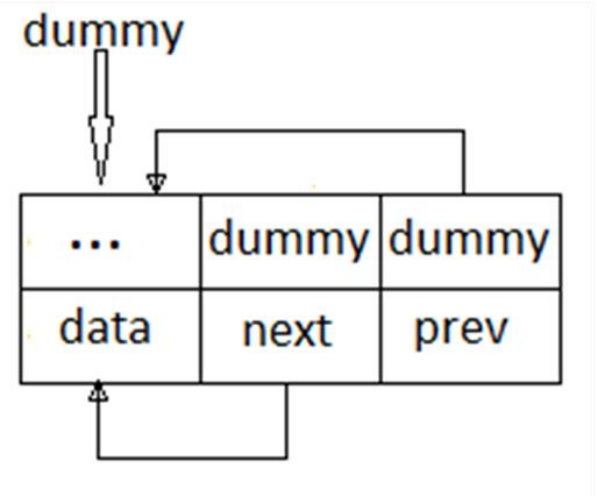
Конструктор без параметров

```
List::List() {
    dummy = new node(0, nullptr, nullptr);
    dummy->next = dummy;
    dummy->prev = dummy;
}
```



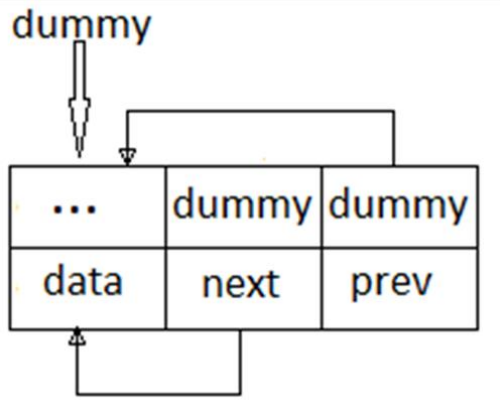
Проверка на пустоту

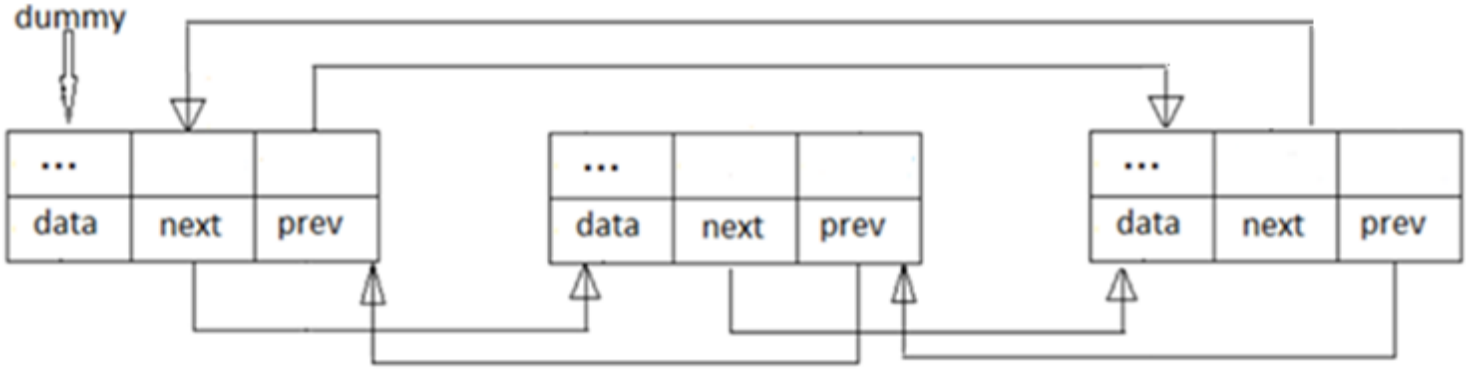
```
bool List::isEmpty() const {
    return (dummy->next == dummy);
}
```



Конструктор копии

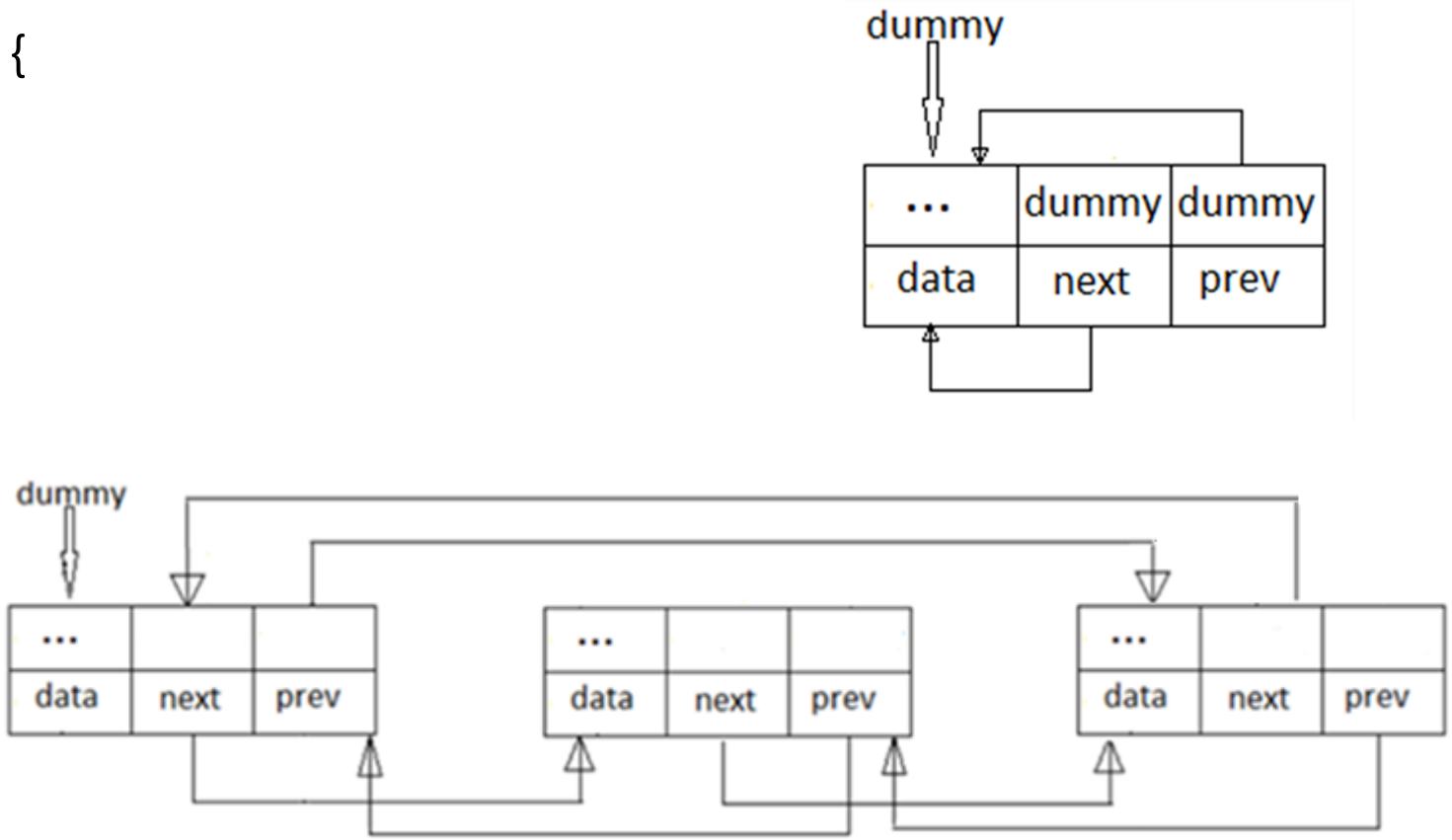
```
List::List(const List& l): List() {
    dummy = new node(0, nullptr, nullptr);
    dummy->next = dummy;
    dummy->prev = dummy;
    if (l.dummy->next!=l.dummy) {
        node* q = l.dummy->next;
        while (q != l.dummy) {
            inTail(q->data);
            q = q->next;
        }
    }
}
```





Деструктор

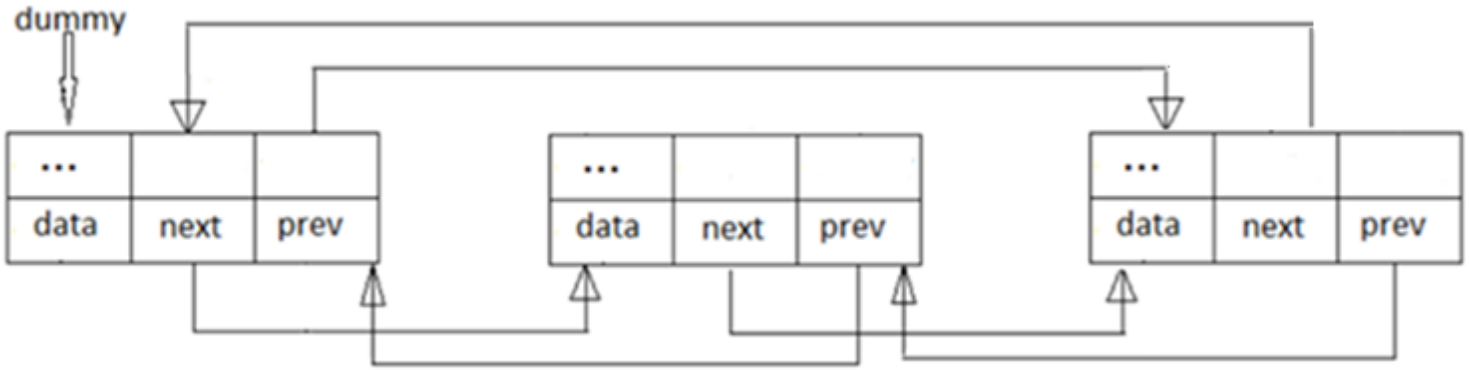
```
List::~List() {
   if (dummy->next!=dummy) {
      node* t = dummy->next;
      while (t!=dummy) {
         node* cur = t;
         t = t->next;
         delete cur;
      }
   }
   delete dummy;
}
```



Алгоритмы добавления/удаления в позициях начала и конца списка становятся сложнее не только из-за наличия второго указателя, но и из-за необходимости учёта заглавного узла.

Добавление в позиции начала списка

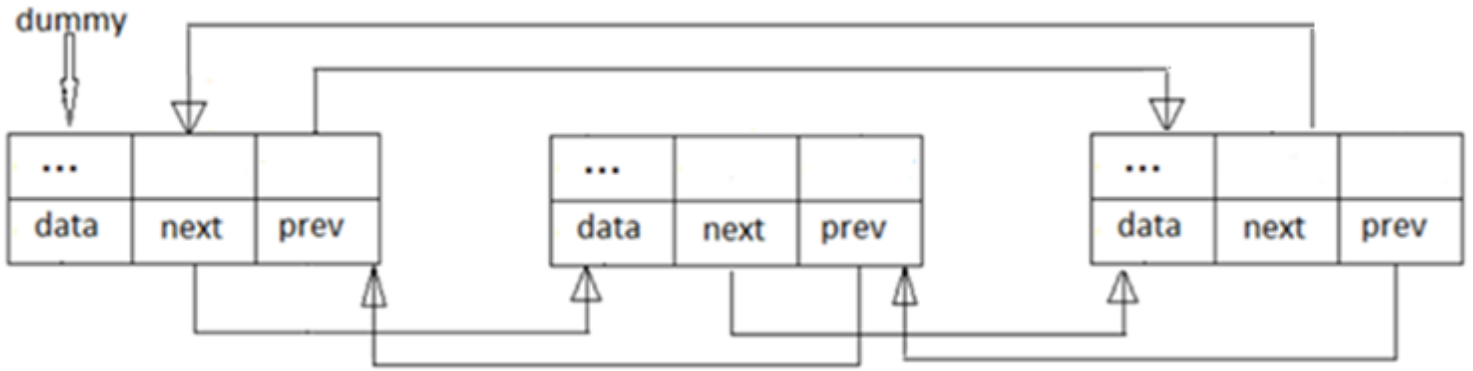
```
void List::inHead(int val) {
    node* t = new node(val, dummy->next, dummy);
    if (dummy->next==dummy) {
        dummy->prev = t;
        t->next = dummy;
    }
    else {
        t->next->prev = t;
    }
    dummy->next = t;
}
```



Добавление в позиции конца списка

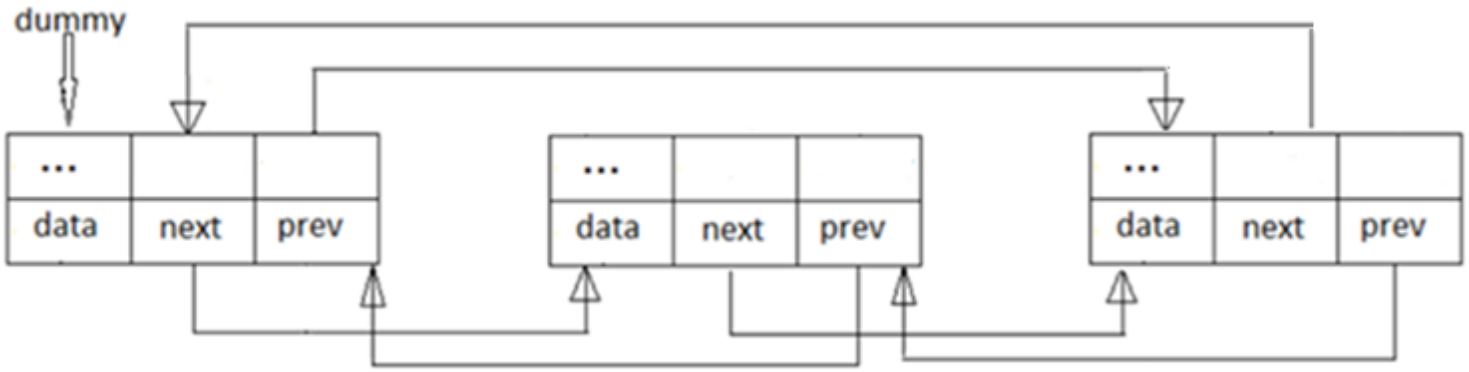
```
void List::inTail(int val) {
    if (dummy->next == dummy)
        inHead(val);
    else {
        node* t = new node(val, dummy, dummy->prev);
        t->prev->next = t;
        dummy -> prev = t;
    }

}
```



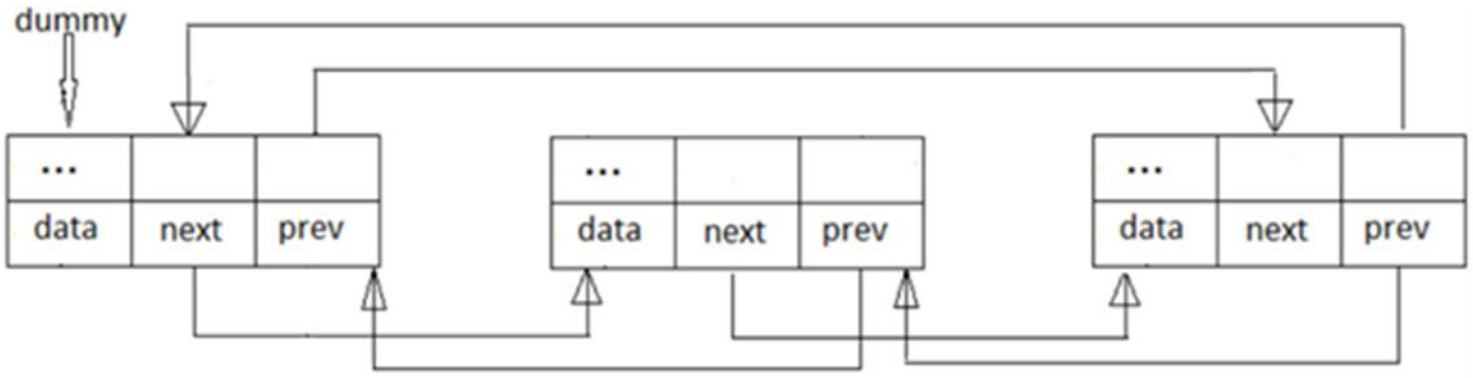
Удаление в позиции начала списка

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



Удаление в позиции конца списка

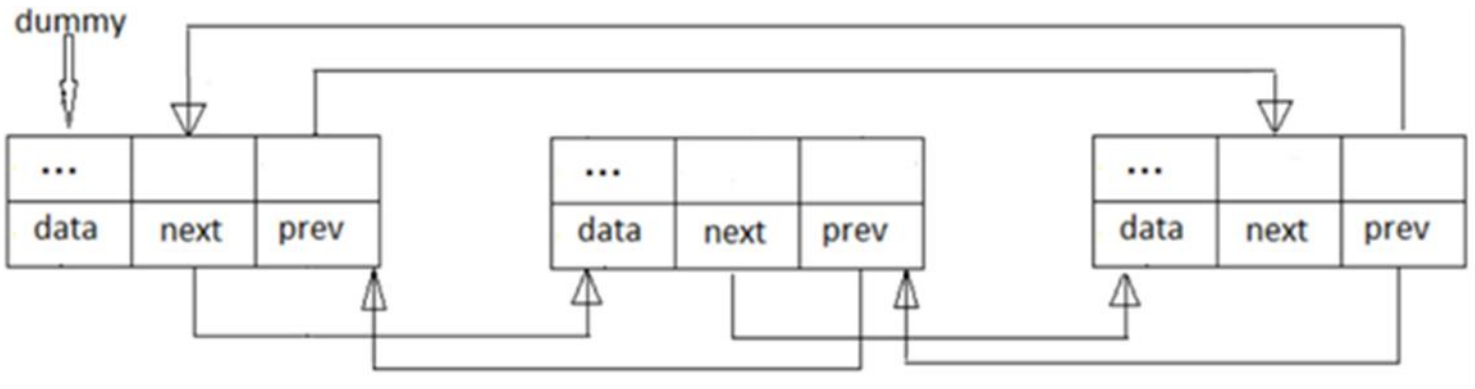
```
void List::delLast() {
   if (dummy->next!=dummy) {
      node* t = dummy->prev;
      if (t->prev!=dummy) {
         dummy->prev = t->prev;
         t->prev->next = dummy;
      }
      else{
       dummy->next = dummy;
       dummy->prev = dummy;
      }
   delete t;
   }
   else
      throw err;
}
```



Доступ к значениям первого и последнего элементов

```
int List::getFirst() const {
    if (dummy->next!=dummy)
        return dummy->next->data;
    else
        throw err;
}
```

```
int List::getLast()const {
    if (dummy->next!=dummy)
        return dummy->prev->data;
    else
        throw err;
}
```



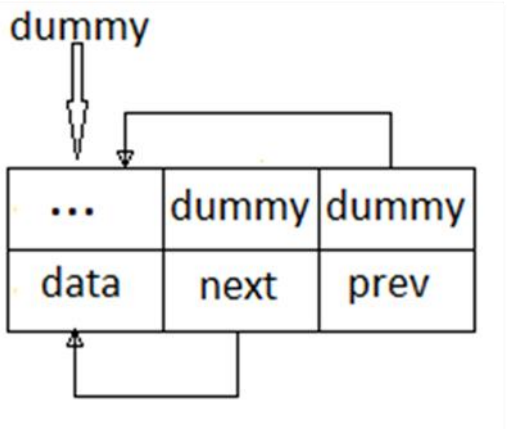
Вывод в поток

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

Реализация списка с использованием заглавного элемента не должна нарушать соглашение:

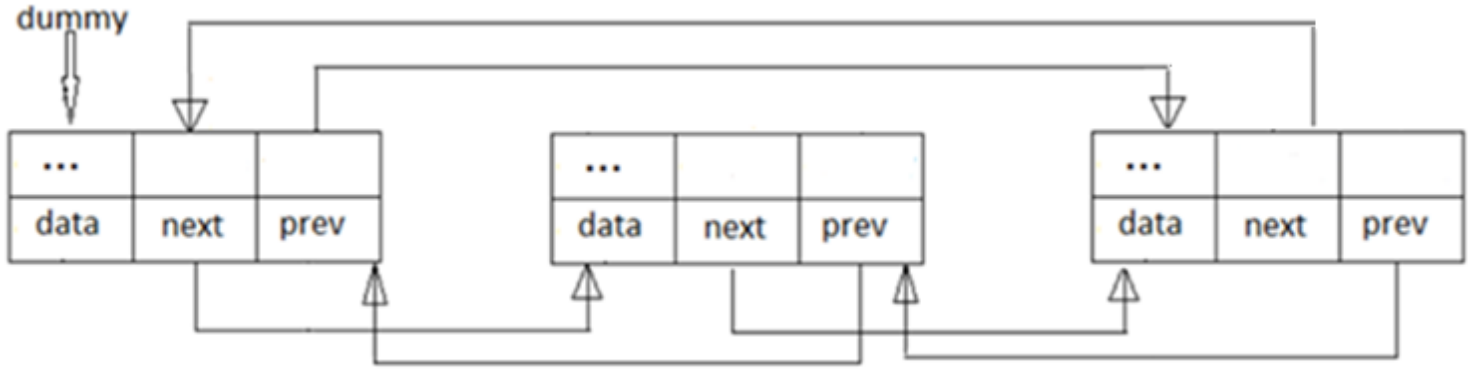
для пустого списка begin() == end()

Именно поэтому next и prev заглавного элемента равны dummy.



Методы begin() и end()

```
listIterator List::begin() const {
    return listIterator(this, dummy->next);
}
listIterator List::end() const {
    listIterator iter(this, dummy);
    return iter;
}
```



class listIterator

```
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) {}
   const int operator *() {
      if (cur == collection->dummy) throw Error();
      return cur->data;
   }
   listIterator operator++(); //префиксный ++
   listIterator operator--(); //префиксный --
   int operator == (const listIterator& ri) const;
   int operator != (const listIterator& ri) const;
};
```

```
listIterator listIterator:: operator++() {
    if (cur!=collection->dummy) {
        cur = cur->next;
        return *this;
    }
    else throw Error();
// или collection.isEmpty() или cur достиг end()
}
```

```
listIterator listIterator:: operator--() {
    if (cur->prev != collection->dummy) {
        cur = cur->prev;
        return *this;
    }
    else throw Error();
    // или collection.isEmpty() или cur достиг begin()
}
```

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

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

Ищем сумму

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

Ищем максимум

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

Выводим в обратном порядке

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

```
int main() {
    List l1;
    l1.inHead(100);
    for (int i = 0; i < 5; ++i)  l1.inHead(i);
    for (int i = 5; i < 10; ++i)  l1.inTail(i);
    cout << " list \n" << l1 << endl;

    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(l2.begin(), mt);

    reverseprint(mt, l2.end());

    return 0;
}
```