

Простейшие структуры данных

М. А. Ложников

31 октября 2019 г.

Ревизия: 0

Это предварительная невыверенная версия. Читайте на свой страх и риск.

1 Простейший односторонний список

Рассмотрим простейший односторонний список, элементами которого являются целые числа типа `int`.

```
1 // Класс, описывающий ячейку одностороннего списка
2 struct ListNode {
3     int value;          // Число, которое лежит в текущей ячейке списка
4     ListNode* next;    // Указатель на следующую ячейку
5
6     ListNode(int value) :
7         value(value),   // Записываем значение в ячейку
8         next(NULL)      // По умолчанию считаем, что это последняя ячейка
9     {};
10};
```

Этот список называется односторонним, потому что по нему можно проходить только в одном направлении: из начала в конец. Если поле `next` класса `ListNode` равно `NULL`, то данная ячейка является последней в списке.

Задача 1. Даны два непустых связных (односторонних) списка, представляющих два неотрицательных целых числа. Цифры хранятся в обратном порядке, каждая ячейка списка содержит по одной цифре. Требуется написать функцию, складывающую эти два числа и возвращающую результат в виде одностороннего списка. Можно считать, что числа не начинаются с нуля кроме самого числа 0. Источник: <https://leetcode.com/problems/add-two-numbers/>.

Решение. Рассмотрим следующую функцию:

```
1 ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
2     int value = l1->value + l2->value; // Складываем две последние цифры младших разрядов
3     // Создаём список с результатом их сложения. В переменной result будет храниться ответ.
4     ListNode* result = new ListNode(value % 10);
5     // Нам понадобится отдельная переменная, чтобы итерироваться по списку
6     ListNode* ptr = result;
7     l1 = l1->next; // Переходим к следующему разряду
8     l2 = l2->next;
```

```
9 // Делаем поразрядное сложение
10 // Указатель ptr в цикле всегда указывает на последнюю ячейку списка
11 while (l1 != NULL || l2 != NULL) { // Пока не дошли до конца обоих чисел
12     value = (value > 9); // 1, если нужно перенести единицу в текущий разряд
13     if (l1) { // Если в первом числе остались разряды
14         value += l1->value; // Обновляем результат
15         l1 = l1->next; // Итерируемся вперёд по списку
16     }
17     if (l2) { // Если во втором числе остались цифры
18         value += l2->value;
19         l2 = l2->next;
20     }
21 }
22
23 // Добавляем результат сложения в текущем разряде в конец списка result
24 ptr->next = new ListNode(value % 10);
25 ptr = ptr->next; // ptr должен указывать на последнюю ячейку списка result
26 }
27
28 if (value > 9) // Переносим остаток (если есть) в новый разряд
29     ptr->next = new ListNode(1);
30
31 return result;
32 }
```

Замечание 1.1. В стандартной библиотеке C++ есть шаблон `forward_list`, реализующий односторонний список. Для работы с ним необходимо подключить `<forward_list>`.

2 Стек

В данном разделе разберём такую структуру данных как стек, а также приведём пример работы с ним.

2.1 Описание стека

Стеком назовём структуру данных, которая может хранить набор одинаковых элементов, и для которой реализованы операции добавления нового элемента в конец стека, просмотр последнего добавленного элемента, а также его удаление. В стандартной библиотеке языка C++ стек реализуется шаблоном `std::stack`, для работы которого нужно подключить `<stack>`.

Напишем упрощённую реализацию. Ячейка стека, хранящего элементы типа `int` устроена следующим образом:

```
1 // Класс, описывающий ячейку стека
2 struct StackNode {
3     int value;          // Число, которое лежит в текущей ячейке стека
4     StackNode* prev;   // Указатель на предыдущую ячейку
5
6     // Конструктор ячейки. По умолчанию считаем, что это первая ячейка
7     StackNode(int value, StackNode* prev = NULL) :
8         value(value), // Записываем значение в ячейку
```

```
9     prev(prev)
10    { }
11 };
```

Напишем класс-обёртку для элементов стека.

```
1 #include <stdexcept> // Файл, необходимый для работы со стандартными исключениями
2
3 class Stack {
4 public:
5     Stack() :
6         topNode(NULL), // Это верхний элемент стека. По умолчанию стек пустой.
7         size(0)        // Иногда бывает полезно запоминать размер стека. Это необязательно.
8     { }
9     ~Stack() {
10        while (topNode != NULL) { // Итерируемся по стеку и освобождаем память
11            StackNode* tmp = topNode;
12
13            topNode = topNode->prev; // Переходим на предыдущую ячейку
14            delete tmp;             // Удаляем ячейку, с которой только что перешли.
15        }
16    }
17
18    // Кладём элемент в стек
19    void Push(int value) {
20        // Создаём новую ячейку стека, хранящую значение value, говорим, что
21        // весь накопленный стек хранится перед ней, и записываем указатель на неё в topNode.
22        // Таким образом, topNode указывает на верхушку стека.
23        topNode = new StackNode(value, topNode);
24        size++;
25    }
26
27    // Удаляем элемент из стека
28    void Pop() {
29        if (topNode == NULL) // Если стек пустой, кидаем исключение.
30            throw std::out_of_range("Stack is empty!");
31
32        StackNode* tmp = topNode;
33
34        topNode = topNode->prev; // Переходим на предыдущую ячейку
35        delete tmp;             // Удаляем ячейку, с которой только что перешли.
36        size--;
37    }
38
39    // Просматриваем последний добавленный элемент
40    int Top() const {
41        if (topNode == NULL) // Если стек пустой, кидаем исключение.
42            throw std::out_of_range("Stack is empty!");
43
44        return topNode->value;
45    }
46}
```

```
45 }
46
47 // Возвращаем размер стека
48 std::size_t Size() const { return size; }
49
50 // Является ли стек пустым?
51 bool Empty() const { return topNode == NULL; }
52
53 private:
54 StackNode* topNode; // Указатель на верхнюю ячейку стека
55 std::size_t size; // Размер стека
56 };
```

Замечание 2.1. Обратите внимание на то, что класс `Stack` нельзя копировать. Дело в том, что в классе не переопределена операция присваивания, а также не реализован copy-конструктор. Стандартные реализации, которые за нас сделает компилятор, просто скопируют указатель `topNode` из-за чего в деструкторе может произойти двойное освобождение памяти.

Замечание 2.2. В C++11 появился удобный синтаксис для того, чтобы сказать компилятору о том, что не нужно создавать метод по умолчанию. Для этого нужно обьявить метод в классе и в конце обьявления написать `= delete`. Таким образом, для того, чтобы запретить копирование класса `Stack` достаточно добавить в обьявление класса две строчки:

```
1 class Stack {
2 /*
3   Все поля и методы, написанные в прошлый раз
4 */
5
6 // Отличия:
7 public:
8 // Удаляем стандартный copy-конструктор
9 Stack(const Stack& other) = delete;
10 // Удаляем стандартную операцию присваивания
11 Stack& operator=(const Stack& other) = delete;
12 };
```

2.2 Пример использования класса `Stack`

Приведём решение следующей задачи:

Задача 2. Считайте с клавиатуры в стек последовательность и выведите её в обратном порядке.

```
1 #include <iostream>
2 // Наш файл с определением классов Stack и StackNode из предыдущего подраздела
3 #include "stack.hpp"
4
5 int main() {
6     Stack stack;
7     int elem;
```

```
9 // Операция >> для потока ввода возвращает ссылку на этот поток.
10 // А для самого потока переопределена операция приведения к типу
11 // 1) void* до стандарта C++11;
12 // 2) bool начиная со стандарта C++11,
13 // возвращающая NULL (или false), если произошла ошибка.
14 // Таким образом, следующая конструкция будет считывать "пока считывается".
15 while (std::cin >> elem) {
16     stack.Push(elem);
17 }
18
19 std::cout << "Result:" << std::endl;
20 // А теперь выводим эти элементы
21 while (!stack.Empty()) { // Пока стек не пуст
22     std::cout << stack.Top() << std::endl; // Выводим верхний элемент
23     stack.Pop(); // И вытаскиваем его из стека
24 }
25 return 0;
26 }
```

3 Простейшие итераторы

В данном разделе напишем простейший итератор для стека, а также приведём пример работы с ним. Отметим, что шаблон стандартной библиотеки C++ `std::stack` итераторов не содержит.

3.1 Пример реализации

Дополним наш класс `Stack` итераторами. То есть определим внутри него некоторые вложенные классы, например, `Iterator` и `ConstIterator`, которые работают как обычные указатели в Си, то есть операция `*` будет возвращать ссылку на элемент стека, и операция `++` будет менять итератор так, чтобы он “указывал” на следующий элемент стека, а также должны быть определены операции сравнения для итераторов. Кроме того, в основном классе необходимо реализовать некоторые методы, скажем, `Begin()` и `End()`, возвращающие итератор на начало стека и невалидный итератор на конец стека (указывает на элемент “после последнего”) соответственно. Класс `ConstIterator` отличается от класса `Iterator` тем, что применённая к нему операция `*` возвращает либо константную ссылку либо значение, то есть с его помощью нельзя менять элементы нашего контейнера.

```
1 class StackWithIterator {
2     // Вначале идут все те же поля и методы, что и в классе Stack.
3     // Для краткости они здесь опущены.
4     /*
5         Поля и методы, аналогичные полям и методам Stack...
6     */
7
8     // А теперь начинаются отличия
9 public:
10    // Сначала определим так называемый константный итератор, который
11    // не может менять элементы, на которые он указывает.
12    class ConstIterator {
13        public:
```

```
14 ConstIterator(const StackNode* currentNode = NULL) :
15     currentNode(currentNode)
16 { }
17
18 int operator*() const { // Разыменовываем оператор
19     return currentNode->value;
20 }
21 ConstIterator& operator++() { // Переходим к следующей ячейке
22     currentNode = currentNode->prev;
23     return *this;
24 }
25
26 // Для сравнения итераторов нам необходимо определить операции сравнения
27 bool operator!=(const ConstIterator& other) const {
28     return currentNode != other.currentNode;
29 }
30
31 bool operator==(const ConstIterator& other) const {
32     return currentNode == other.currentNode;
33 }
34 private:
35     // Это указатель на константу, сам указатель менять можно,
36     // а то, на что он указывает -- нет.
37     const StackNode* currentNode;
38 };
39
40 // Теперь определим итератор, который может менять элементы,
41 // на которые он указывает.
42 class Iterator {
43 public:
44     Iterator(StackNode* currentNode = NULL) :
45         currentNode(currentNode)
46     { }
47
48     // Разыменовываем оператор. Метод возвращает ссылку на элемент.
49     int& operator*() const {
50         return currentNode->value;
51     }
52     Iterator& operator++() { // Переходим к следующей ячейке
53         currentNode = currentNode->prev;
54         return *this;
55     }
56
57     // Для сравнения итераторов нам необходимо определить операцию сравнения
58     bool operator!=(const Iterator& other) const {
59         return currentNode != other.currentNode;
60     }
61
62     bool operator==(const Iterator& other) const {
63         return currentNode == other.currentNode;
64     }
```

```
65 private:
66     StackNode* currentNode; // Указатель на текущую ячейку
67 };
68
69 // Константный итератор на начало промежутка
70 ConstIterator Begin() const { return ConstIterator(topNode); }
71 // Константный итератор на конец промежутка (указывает на элемент "после последнего")
72 ConstIterator End() const { return ConstIterator(); }
73 // Итератор на начало промежутка
74 Iterator Begin() { return Iterator(topNode); }
75 // Итератор на конец промежутка (указывает на элемент "после последнего")
76 Iterator End() { return Iterator(); }
77 };
```

Замечание 3.1. В примере выше метод `End()` возвращает невалидный итератор, указывающий на элемент “после последнего” таким образом, элементы стека лежат в промежутке `[Begin(), End()]`.

Замечание 3.2. Аналогично нашему классу `Stack` (см. замечание 2.1), класс `StackWithIterator` нельзя копировать.

Замечание 3.3. Стоит отметить, что для итераторов обычно используют префиксную операцию инкремента (`++`), поскольку она не создаёт новый объект в отличие от постфиксной. Именно поэтому мы привели реализацию префиксной операции.

3.2 Пример использования

Задача 3. Напишем программу, прибавляющую к каждому элементу массива следующий элемент.

Решение:

```
1 #include <iostream>
2 // Файл с определением нашего класса StackWithIterator
3 #include "stack_with_iterator.hpp"
4
5 void ReadStack(StackWithIterator& stack) {
6     int elem;
7     while (std::cin >> elem) {
8         stack.Push(elem);
9     }
10 }
11
12 void PrintStack(const StackWithIterator& stack) {
13     // Итерируемся по циклу с помощью константного итератора ввиду того, что
14     // в функцию передана ссылка на const StackWithIterator
15     // Инициализируем итератором на начало
16     for (StackWithIterator::ConstIterator it = stack.Begin();
17          it != stack.End(); // Проверяем, дошли ли до конца
18          ++it) { // Переходим к следующему итератору
19         // *it возвращает элемент стека типа int
20         std::cout << *it << std::endl;
21     }
22 }
```

```
21     }
22 }
23
24 void Process(StackWithIterator& stack) {
25     // Модифицируем элементы, поэтому оператор неконстантный
26     StackWithIterator::Iterator it = stack.Begin();
27     int prevValue = *it;
28     ++it; // Переходим к следующему элементу
29
30     for (; it != stack.End(); ++it) {
31         int tmp = *it;
32         *it += prevValue;
33         prevValue = tmp;
34     }
35 }
36
37 int main() {
38     StackWithIterator stack;
39
40     ReadStack(stack);
41
42     Process(stack);
43
44     PrintStack(stack);
45
46     return 0;
47 }
```

4 Двунаправленный список

В стандартной библиотеке C++ есть шаблон `list`, определённый в `<list>` и реализующий двунаправленный список.

4.1 Ячейка двунаправленного списка

В отличие от стека и однонаправленного списка, ячейки двунаправленных списков содержат указатели на предыдущий и на следующий элементы. Таким образом, в списки можно добавлять элементы в произвольную позицию, удалять произвольные элементы, а также итерироваться в обоих направлениях. Реализуем класс `ListNode`, описывающий ячейку однонаправленного списка, хранящую элементы типа `int`.

```
1 struct ListNode {
2     int value;          // Значение в ячейке
3     ListNode* prev;   // Указатель на предыдущий элемент
4     ListNode* next;   // Указатель на следующий элемент
5
6     ListNode(int value, ListNode* prev = NULL, ListNode* next = NULL) :
7         value(value),
8         prev(prev),
```

```
9     next(next)
10    { }
11 };
```

4.2 Реализация двунаправленного списка

Разобьём реализацию списка на три части: базовые методы работы со списком, итераторы и методы вставки и удаления элемента. Таким образом, список будет выглядеть так:

```
1 #include <stdexcept>
2
3 class List {
4     /*
5      Базовые поля и методы класса
6     */
7
8     /*
9      Итераторы: прямые и обратные, а также методы для работы с ними
10     */
11
12     /*
13      Методы вставки и удаления элемента
14     */
15};
```

Приведём реализацию класса по частям.

4.2.1 Базовые члены класса

```
1 class List {
2 public:
3     List() : // Конструктор по умолчанию
4         frontNode(NULL),
5         backNode(NULL)
6     { }
7     // Сору-конструктор
8     List(const List& other) :
9         frontNode(NULL),
10        backNode(NULL)
11    {
12        if (!other.frontNode)
13            return; // Список пустой, делать нечего.
14
15        // Оператор new выбрасывает исключение std::bad_alloc в том случае, если ему
16        // не удается выделить память. Нам нужно отловить это исключение, освободить
17        // всю выделенную память и передать исключение дальше по цепочке
18        // (следующему обработчику, если он есть), чтобы показать, что наш список
19        // не удалось скопировать.
20        try {
```

```
21 // Создаём первую ячейку посредством стандартного сору-конструктора
22 // Этот конструктор неправильно инициализирует соседей (он просто скопирует
23 // указатели на соответствующие ячейки старого списка). Будем это исправлять.
24 frontNode = new ListNode(*other.frontNode);
25 frontNode->prev = NULL; // Это первая ячейка, у неё нет предыдущей
26 frontNode->next = NULL; // Инициализируем по умолчанию
27
28 // Определим переменные для итерирования по ячейкам списка
29 const ListNode* otherNode = other.frontNode; // Для итерирования по старому списку
30 ListNode* node = frontNode; // Для итерирования по новому списку
31
32 // Итерируемся по всем ячейкам и копируем данные
33 while (otherNode->next != NULL) { // Пока не дойдём до последней ячейки
34     // Создаём новую ячейку копированием из соответствующей старой
35     node->next = new ListNode(*otherNode->next);
36     node->next->prev = node; // Перед созданной ячейкой находится текущая
37     node->next->next = NULL; // Инициализируем по умолчанию
38
39     // Переходим к следующей ячейке
40     otherNode = otherNode->next;
41     node = node->next;
42 }
43
44 backNode = node; // Инициализируем последнюю ячейку
45 }
46 catch(std::bad_alloc& e) {
47     // Оператор new сгенерировал исключение, не удалось выделить память.
48     // Для того, чтобы избежать утечек, нам нужно освободить всю память,
49     // которую выделили.
50     Clear();
51
52     // Выбрасываем то же самое исключение, чтобы его поймал следующий
53     // обработчик (если он есть). Тем самым мы говорим, что список
54     // не удалось скопировать.
55     throw;
56 }
57 }
58
59 ~List() {
60     Clear(); // Очищаем всю выделенную память
61 }
62
63 List& operator=(const List& other) {
64     Clear(); // Сначала нужно подчистить старый список
65
66     if (!other.frontNode)
67         return *this; // Список пуст, делать нечего.
68
69     // Аналогично сору-конструктору, проверяем, выделилась ли память.
70     try {
71         // Создаём первую ячейку посредством сору-конструктора
```

```
72     frontNode = new ListNode(*other.frontNode);
73     frontNode->prev = NULL; // Это первая ячейка, у неё нет предыдущей
74     frontNode->next = NULL; // Инициализируем по умолчанию
75
76     // Определим переменные для итерирования по ячейкам списка
77     const ListNode* otherNode = other.frontNode; // Для итерирования по старому списку
78     ListNode* node = frontNode; // Для итерирования по новому списку
79
80     // Итерируемся по всем ячейкам и копируем данные
81     while (otherNode->next != NULL) {
82         // Создаём новую ячейку копированием из соответствующей старой
83         node->next = new ListNode(*otherNode->next);
84         node->next->prev = node; // Перед созданной ячейкой находится текущая
85         node->next->next = NULL; // Инициализируем по умолчанию
86
87         // Переходим к следующей ячейке
88         otherNode = otherNode->next;
89         node = node->next;
90     }
91
92     backNode = node; // Инициализируем последнюю ячейку
93 }
94 catch (std::bad_alloc& e) {
95     // Оператор new выбросил исключение std::bad_alloc.
96     // Не удалось выделить память. Очищаем всё, что выделили.
97     Clear();
98     // Пробрасываем исключение дальше.
99     throw;
100 }
101 return *this;
102 }
103
104 void Clear() {
105     // Итерируемся по циклу от первой ячейки до последней
106     while (frontNode != NULL) {
107         ListNode* tmp = frontNode;
108
109         frontNode = frontNode->next; // Переходим к следующей ячейке
110         delete tmp; // Удаляем ту, из которой ушли
111     }
112     frontNode = NULL; // Обнулим указатели, чтобы показать, что список пуст.
113     backNode = NULL;
114 }
115 // Добавляем элемент в начало списка
116 void PushFront(int value) {
117     if (frontNode) {
118         // Создаём ячейку в начале списка (перед frontNode)
119         frontNode->prev = new ListNode(value, NULL, frontNode);
120         frontNode = frontNode->prev; // Обновляем frontNode
121     } else {
122         frontNode = backNode = new ListNode(value); // Список был пуст
```

```
123     }
124 }
125
126 void PushBack(int value) {
127     if (backNode) {
128         // Создаём ячейку в конце списка (после backNode)
129         backNode->next = new ListNode(value, backNode);
130         backNode = backNode->next;
131     } else {
132         frontNode = backNode = new ListNode(value); // Список был пуст
133     }
134 }
135
136 void PopFront() {
137     if (!frontNode) // Если список пуст, кидаем исключение
138         throw std::out_of_range("List is empty!");
139
140     ListNode* tmp = frontNode; // Сохраняем старое значение
141     frontNode = frontNode->next; // Обновляем указатель на первую ячейку
142
143     if (frontNode == NULL)
144         backNode = NULL; // Список состоял из одного элемента
145     else
146         frontNode->prev = NULL; // Эта ячейка теперь первая
147
148     delete tmp; // Освобождаем память удалённой ячейки
149 }
150
151 void PopBack() {
152     if (!backNode) // Если список пуст, кидаем исключение
153         throw std::out_of_range("List is empty!");
154
155     ListNode* tmp = backNode; // Сохраняем старое значение
156     backNode = backNode->prev; // Обновляем указатель на последнюю ячейку
157
158     if (backNode == NULL)
159         frontNode = NULL; // Список состоял из одного элемента
160     else
161         backNode->next = NULL; // Эта ячейка теперь последняя
162
163     delete tmp;
164 }
165
166 // В следующих двух функциях проверок не делается.
167 // Функция для получения первого элемента списка.
168 int Front() const { return frontNode->value; }
169 // Функция для получения последнего элемента списка.
170 int Back() const { return backNode->value; }
171
172 // Является ли список пустым?
173 bool Empty() const {
```

```
174     return frontNode == NULL;
175 }
176
177 private:
178     ListNode* frontNode; // Указатель на первую ячейку списка
179     ListNode* backNode; // Указатель на последнюю ячейку списка
180
181 /**
182     Итераторы: прямые и обратные
183 */
184
185 /**
186     Методы вставки и удаления элементов
187 */
188 };
```

4.2.2 Итераторы

Отличие от итераторов стека в том, что для этих итераторов определена ещё и операция --.

Прямые итераторы. Реализуем итераторы, которые проходят по списку из начала в конец.

```
1 class List {
2     /*
3         Базовые поля и методы класса
4     */
5
6     // Реализация прямых итераторов
7     public:
8     // Сначала константный итератор, который не может менять элементы списка
9     class ConstIterator {
10         public:
11             ConstIterator(ListNode* currentNode = NULL) :
12                 currentNode(currentNode)
13             {}
14
15             int operator*() const { // Разыменовываем итератор
16                 return currentNode->value;
17             }
18             ConstIterator& operator++() { // Переходим к следующей ячейке
19                 currentNode = currentNode->next;
20                 return *this;
21             }
22             // Переходим к предыдущей ячейке. Итератор стека этого не умел.
23             ConstIterator& operator--() {
24                 currentNode = currentNode->prev;
25                 return *this;
26             }
27
28             // Для сравнения итераторов нам необходимо определить операции сравнения
29             bool operator==(const ConstIterator& other) const {
```

```
30     return currentNode != other.currentNode;
31 }
32
33 bool operator==(const ConstIterator& other) const {
34     return currentNode == other.currentNode;
35 }
36 private:
37     ListNode* currentNode;
38
39 // Класс List должен иметь доступ к приватным полям для реализации методов
40 // вставки и удаления элементов
41 friend class List;
42 };
43
44 // Теперь определим оператор, который может менять элементы,
45 // на которые он указывает
46 class Iterator {
47 public:
48     Iterator(ListNode* currentNode = NULL) :
49         currentNode(currentNode)
50     { }
51
52 // Разыменовываем оператор. Метод возвращает ссылку на элемент.
53 int& operator*() const {
54     return currentNode->value;
55 }
56 Iterator& operator++() { // Переходим к следующей ячейке
57     currentNode = currentNode->next;
58     return *this;
59 }
60
61 // Переходим к предыдущей ячейке. Оператор стека этого не умел.
62 Iterator& operator--() {
63     currentNode = currentNode->prev;
64     return *this;
65 }
66
67 // Для сравнения операторов нам необходимо определить операцию сравнения
68 bool operator!=(const Iterator& other) const {
69     return currentNode != other.currentNode;
70 }
71
72 bool operator==(const Iterator& other) const {
73     return currentNode == other.currentNode;
74 }
75 private:
76     ListNode* currentNode; // Указатель на текущую ячейку
77
78 // Класс List должен иметь доступ к приватным полям для реализации методов
79 // вставки и удаления элементов.
80 friend class List;
```

```
81 };  
82  
83 // Константный итератор на начало промежутка  
84 ConstIterator Begin() const { return ConstIterator(frontNode); }  
85 // Константный итератор на конец промежутка (указывает на элемент "после последнего")  
86 ConstIterator End() const { return ConstIterator(); }  
87 // Итератор на начало промежутка  
88 Iterator Begin() { return Iterator(frontNode); }  
89 // Итератор на конец промежутка (указывает на элемент "после последнего")  
90 Iterator End() { return Iterator(); }  
91  
92 /*  
93     Обратные итераторы  
94 */  
95  
96 /*  
97     Методы вставки и удаления элемента  
98 */  
99 };
```

Обратные итераторы. Они служат для прохода из конца списка в начало. У них инвертированы операции `++` и `--`. Для работы с ними нам потребуется определить в основном классе методы `RBegin()` и `REnd()`, возвращающие итератор на последний элемент списка и невалидный итератор на начало списка (указывает на элемент “перед первым”) соответственно.

```
1 class List {  
2     /*  
3         Базовые поля и методы класса  
4     */  
5  
6     /*  
7         Прямые итераторы  
8     */  
9  
10    // Реализация обратных итераторов  
11    public:  
12        // Сначала константный итератор, который не может менять элементы списка  
13        class ConstReverseIterator {  
14            public:  
15                ConstReverseIterator(ListNode* currentNode = NULL) :  
16                    currentNode(currentNode)  
17                {}  
18  
19                int operator*() const { // Разыменовываем итератор  
20                    return currentNode->value;  
21                }  
22                // Для обратных итераторов эти операции инвертированы  
23                // Переходим к следующей (то есть предыдущей) ячейке  
24                ConstReverseIterator& operator++()  
25                { currentNode = currentNode->prev;
```

```
26     return *this;
27 }
28
29 // Переходим к предыдущей (то есть следующей) ячейке
30 ConstReverseIterator& operator--() {
31     currentNode = currentNode->next;
32     return *this;
33 }
34
35 // Для сравнения операторов нам необходимо определить операции сравнения
36 bool operator!=(const ConstReverseIterator& other) const {
37     return currentNode != other.currentNode;
38 }
39
40 bool operator==(const ConstReverseIterator& other) const {
41     return currentNode == other.currentNode;
42 }
43 private:
44     ListNode* currentNode;
45
46 // Класс List должен иметь доступ к приватным полям для реализации методов
47 // вставки и удаления элементов.
48 friend class List;
49 };
50
51 // Теперь определим оператор, который может менять элементы,
52 // на которые он указывает
53 class ReverseIterator {
54 public:
55     ReverseIterator(ListNode* currentNode = NULL) :
56         currentNode(currentNode)
57     { }
58
59     // Разыменовываем оператор. Метод возвращает ссылку на элемент.
60     int& operator*() const {
61         return currentNode->value;
62     }
63
64     // Для обратных операторов эти операции инвертированы
65     // Переходим к следующей (то есть предыдущей) ячейке
66     ReverseIterator& operator++() {
67         currentNode = currentNode->prev;
68         return *this;
69     }
70
71     // Переходим к предыдущей (то есть следующей) ячейке
72     ReverseIterator& operator--() {
73         currentNode = currentNode->next;
74         return *this;
75     }
76
```

```
77 // Для сравнения итераторов нам необходимо определить операцию сравнения
78 bool operator!=(const ReverseIterator& other) const {
79     return currentNode != other.currentNode;
80 }
81
82 bool operator==(const ReverseIterator& other) const {
83     return currentNode == other.currentNode;
84 }
85 private:
86 ListNode* currentNode; // Указатель на текущую ячейку
87
88 // Класс List должен иметь доступ к приватным полям для реализации методов
89 // вставки и удаления элементов.
90 friend class List;
91 };
92
93 // Для обратных итераторов следующие функции тоже инвертированы
94 // Константный итератор на начало промежутка (то есть на последний элемент списка)
95 ConstReverseIterator RBegin() const { return ConstReverseIterator(backNode); }
96 // Константный итератор на конец промежутка (то есть на элемент "перед первым")
97 ConstReverseIterator REnd() const { return ConstReverseIterator(); }
98 // Итератор на начало промежутка (то есть на последний элемент списка)
99 ReverseIterator RBegin() { return ReverseIterator(backNode); }
100 // Итератор на конец промежутка (то есть на элемент "перед первым")
101 ReverseIterator REnd() { return ReverseIterator(); }
102
103 /*
104     Методы вставки и удаления элемента
105 */
106 };
```

Замечание 4.1. Эти итераторы задают промежуток (`REnd()`, `RBegin()`], где `RBegin()` – итератор на последний элемент списка, а `REnd()` – невалидный итератор на конец промежутка (на элемент “перед первым” в списке).

Замечание 4.2. Обратим внимание, что в данной реализации операция декремента -- некорректна по отношению к итератору на конец промежутка. Однако, в стандартной библиотеке C++ для итераторов шаблона `std::list` эта операция корректна.

4.2.3 Методы вставки и удаления элементов

Список поддерживает операцию вставки элемента в произвольную позицию, а также удаление произвольного элемента. Метод, вставляющий элемент, возвращает итератор на этот элемент. Метод, удаляющий элемент возвращает итератор на элемент, который следовал за удалённым.

```
1 class List {
2     /*
3     Базовые поля и методы класса
4     */
5
6     /*
```

```
7 Итераторы: прямые и обратные, а также методы для работы с ними
8 */
9
10 // Реализация методов вставки и удаления элементов
11 public:
12 // Вставляет элемент value на позицию pos. Возвращает итератор на вставленный элемент.
13 Iterator Insert(Iterator pos, int value) {
14     // Для следующей операции класс List объявлен дружественным для итераторов.
15     ListNode* currentNode = pos.currentNode; // Получаем указатель на текущую ячейку.
16
17     return Insert(currentNode, value);
18 }
19
20 // Можно и по константному итератору
21 // Вставляет элемент value на позицию pos. Возвращает итератор на вставленный элемент.
22 Iterator Insert(ConstIterator pos, int value) {
23     // Для следующей операции класс List объявлен дружественным для итераторов.
24     ListNode* currentNode = pos.currentNode; // Получаем указатель на текущую ячейку.
25
26     return Insert(currentNode, value);
27 }
28
29 // Удаляет элемент, на который указывает итератор pos. Возвращает итератор на элемент,
30 // который следовал за удалённым.
31 Iterator Remove(Iterator pos) {
32     // Для следующей операции класс List объявлен дружественным для итераторов.
33     ListNode* currentNode = pos.currentNode; // Получаем указатель на текущую ячейку.
34
35     return Remove(currentNode);
36 }
37
38 // Можно и для константного итератора
39 // Удаляем элемент, на который указывает итератор pos
40 Iterator Remove(ConstIterator pos) {
41     // Для следующей операции класс List объявлен дружественным для итераторов.
42     ListNode* currentNode = pos.currentNode; // Получаем указатель на текущую ячейку.
43
44     return Remove(currentNode);
45 }
46 private:
47 // Вставляет элемент value перед ячейкой currentNode.
48 // Возвращает итератор на вставленный элемент.
49 Iterator Insert(ListNode* currentNode, int value) {
50     if (currentNode == NULL) { // Если нужно вставить в конец списка
51         PushBack(value);
52         return Iterator(backNode);
53     }
54
55     if (currentNode == frontNode) { // Если нужно вставить в начало списка
56         PushFront(value);
57         return Iterator(frontNode);
```

```
58     }
59
60     // Создаём новую ячейку между предыдущей и текущей
61     ListNode* insertedNode = new ListNode(value, currentNode->prev, currentNode);
62
63     currentNode->prev->next = insertedNode; // После предыдущей идёт вставленная
64     currentNode->prev = insertedNode;        // Вставили ячейку перед текущей
65
66     return Iterator(insertedNode); // Возвращаем итератор на вставленную ячейку
67 }
68
69 // Удаляет ячейку currentNode. Возвращает итератор на элемент, который
70 // следовал за удалённым.
71 Iterator Remove(ListNode* currentNode) {
72     if (currentNode == NULL) // Список пуст, кидаем исключение
73         throw std::out_of_range("Invalid iterator!");
74
75     if (currentNode->next == NULL) { // Если удаляем из конца списка
76         PopBack();
77         return Iterator(NULL);
78     }
79
80     if (currentNode->prev == NULL) { // Если удаляем из начала списка
81         PopFront();
82         return Iterator(frontNode);
83     }
84     // Объявим для наглядности указатели на предыдущую и следующую ячейки
85     ListNode* nextNode = currentNode->next;
86     ListNode* prevNode = currentNode->prev;
87
88     delete currentNode; // Удаляем текущую ячейку
89
90     nextNode->prev = prevNode; // Перед следующей теперь стоит предыдущая.
91     prevNode->next = nextNode; // А после предыдущей следующая.
92
93     return Iterator(nextNode); // Возвращаем итератор на следующую ячейку.
94 }
95 };
```

4.3 Пример работы с двунаправленными списками

Задача 4. Реализуйте сортировку массива вставками с последовательным поиском, затем выведите элементы массива в прямом и обратном порядках.

Решение.

```
1 #include <iostream>
2 // Реализация нашего класса List
3 #include "list.hpp"
4
```

```
5 // Печатаем список. Соединяем с функцией для стека.
6 void PrintList(const List& list) {
7     std::cout << "List:" << std::endl;
8     for (List::ConstIterator it = list.Begin();
9          it != list.End();
10        ++it) {
11         std::cout << *it << std::endl;
12     }
13 }
14
15 // Печатаем список в обратном порядке, используя обратные итераторы.
16 void PrintListReversed(const List& list) {
17     std::cout << "Reversed List:" << std::endl;
18     for (List::ConstReverseIterator it = list.RBegin(); // Это последний элемент
19          it != list.REnd(); // Пока не дошли до начала
20        ++it) { // Оператор ++ инвертирован
21         std::cout << *it << std::endl;
22     }
23 }
24
25 // Возвращаем итератор на первый элемент промежутка, который больше или равен elem
26 // Промежуток определяется [listBegin, listEnd)
27 List::Iterator LowerBound(List::Iterator listBegin, List::Iterator listEnd, int elem) {
28     for (List::Iterator it = listBegin; it != listEnd; ++it) {
29         if (*it >= elem)
30             return it;
31     }
32     return listEnd; // Такого элемента в списке нет.
33 }
34
35 void ReadListAndSort(List& list) {
36     int elem;
37
38     while (std::cin >> elem) { // Считываем элемент
39         // Вычислим итератор на то место в списке, куда этот элемент нужно вставить
40         List::Iterator pos = LowerBound(list.Begin(), list.End(), elem);
41         // А теперь вставим его туда
42         list.Insert(pos, elem);
43     }
44 }
45
46 int main() {
47     List list;
48
49     ReadListAndSort(list);
50
51     PrintList(list);
52
53     PrintListReversed(list);
54
55     return 0;
}
```

