

# Простейшие структуры данных – 2

М. А. Ложников

14 ноября 2019 г.

Ревизия: 0

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

## 1 Циклический буфер

Отметим достоинства шаблона `std::vector` или динамического массива как контейнера для хранения данных

- Вектор позволяет легко добавлять элементы в конец (амортизированная сложность  $O(1)$ ).
- Вектор предоставляет доступ к произвольному элементу по индексу (сложность  $O(1)$ ).
- Итераторы предоставляют доступ к произвольному элементу по его номеру за  $O(1)$ .

Однако, у вектора есть и недостатки:

- Добавление (удаление) элемента в начало или середину имеет линейную сложность  $O(n)$ .
- Метод `push_back` периодически выделяет память и копирует все накопленные данные.

Рассмотрим структуру данных, которая будет обладать всеми перечисленными достоинствами вектора, а также она будет позволять легко за  $O(1)$  операций добавлять элемент в начало контейнера, а также легко удалять его оттуда. Для упрощения реализации будем считать, что размер нашего контейнера фиксирован и задаётся в конструкторе. Такой контейнер называется циклическим буфером. Напишем реализацию, предназначенную для хранения целых чисел типа `int`.

### 1.1 Арифметика в кольце по модулю

Сначала для упрощения вычислений рассмотрим класс `Ring`, реализующий операции сложения и вычитания в кольце чисел по некоторому модулю.

---

```
1 #include <stdexcept>
2 #include <cassert> // нужно для макроса assert()
3
4 // Класс, реализующий операции сложения и вычитания в кольце целых чисел
5 // по модулю size. Если size = 0, то операции сложения и вычитания
6 // работают как с обычными числами.
7 class Ring {
8 public:
9     // Создаём класс Ring из числа. Если size = 0, то это обычное число,
10    // а не число из кольца. Тип данных std::ptrdiff_t - это знаковый тип данных,
```

```
11 // совпадающий по размеру занимаемой памяти с std::size_t. То есть,
12 // он подходит для хранения разности указателей или индексов массива.
13 // Отметим, что оба аргумента операции % должны быть знаковые,
14 // если один из аргументов оказался бы беззнаковым, то второй
15 // аргумент был бы преобразован в беззнаковый перед выполнением операции.
16 Ring(std::ptrdiff_t number, std::ptrdiff_t size = 0) :
17     number(number),
18     size(size)
19 {
20     if (size < 0) // Такого быть не должно.
21         throw std::invalid_argument("Size can't be less zero!");
22     if (size > 0) // Это число в кольце. Считаем остаток от деления.
23     {
24         if (number >= 0)
25             this->number %= size;
26         else
27             this->number = size + number % size;
28
29         // Если утверждение внутри assert() не выполнено, то программа,
30         // скомпилированная без заданного макроса NDEBUG вылетит и напишет,
31         // файл и строчку assert'a, на котором она вылетела.
32         // Программы, скомпилированные с заданным макросом NDEBUG игнорируют assert().
33         // Удобно использовать для отладки.
34         assert(this->number >= 0 && this->number < size);
35     }
36 }
37
38 // Операция сложения для чисел Ring. Размер кольца каждый раз берём максимальный.
39 Ring operator+(const Ring& other) const {
40     return Ring(number + other.number, std::max(size, other.size));
41 }
42
43 // Операция вычитания для чисел Ring. Размер кольца каждый раз берём максимальный.
44 Ring operator-(const Ring& other) const {
45     return Ring(number - other.number, std::max(size, other.size));
46 }
47
48 Ring& operator+=(const Ring& other) {
49     number += other.number; // Прибавляем число.
50
51     if (size == 0 && other.size == 0)
52         return *this; // Если оба числа являются обычными, то ничего не делаем.
53
54     size = std::max(size, other.size); // Пересчитываем размер кольца.
55     if (number > 0) // Считаем остаток от деления.
56         number %= size;
57     else
58         number = size + number % size;
59     return *this;
60 }
61
```

```
62 Ring& operator--(const Ring& other) {
63     number -= other.number; // Вычитаем число.
64
65     if (size == 0 && other.size == 0)
66         return *this; // Если оба числа являются обычными, то ничего не делаем.
67
68     size = std::max(size, other.size); // Пересчитываем размер кольца.
69     if (number > 0) // Считаем остаток от деления.
70         number %= size;
71     else
72         number = size + number % size;
73     return *this;
74 }
75
76 // Префиксная операция сначала меняет число, затем возвращает его.
77 Ring& operator++() {
78     number++;
79     if (size > 0)
80         number %= size;
81     return *this;
82 }
83
84 // Постфиксная операция возвращает старое значение.
85 Ring operator++(int) {
86     Ring retVal(*this); // Стандартный сору-конструктор.
87     number++;
88     if (size > 0)
89         number %= size;
90     return retVal; // Возвращаем старое значение.
91 }
92
93 // Префиксная операция сначала меняет число, затем возвращает его.
94 Ring& operator--() {
95     number--;
96     if (size > 0 && number < 0)
97         number = size + number;
98     return *this;
99 }
100
101 // Постфиксная операция возвращает старое значение.
102 Ring operator--(int) {
103     Ring retVal(*this); // Стандартный сору-конструктор.
104     number--;
105     if (size > 0 && number < 0)
106         number = size + number;
107     return retVal; // Возвращаем старое значение.
108 }
109
110 // Это операция неявного приведения типа к Ring к std::size_t. Она позволяет
111 // использовать Ring в качестве индекса массива или вектора.
112 operator std::size_t() const {
```

```
113     return static_cast<std::size_t>(number);
114 }
115
116 // Всевозможные операции сравнения.
117 bool operator<(const Ring& other) const { return number < other.number; }
118 bool operator>(const Ring& other) const { return number > other.number; }
119 bool operator<=(const Ring& other) const { return number <= other.number; }
120 bool operator>=(const Ring& other) const { return number >= other.number; }
121 bool operator==(const Ring& other) const { return number == other.number; }
122 bool operator!=(const Ring& other) const { return number != other.number; }
123
124 // Функции для чтения приватных полей.
125 std::ptrdiff_t Number() const { return number; }
126 std::ptrdiff_t Size() const { return size; }
127
128 private:
129 // Тип std::ptrdiff_t - знаковый тип данных, пригодный для хранения разности
130 // указателей или разности индексов массива или вектора. В памяти он занимает
131 // столько же места, сколько и std::size_t.
132 std::ptrdiff_t number; // Само число.
133 // Размер кольца. Если size = 0, то арифметические операции работают
134 // как с обычными числами. Эта переменная тоже знаковая для корректной работы
135 // операции деления с остатком.
136 std::ptrdiff_t size;
137 };
```

---

## 1.2 Реализация базовых функций циклического буфера

---

```
1 #include <stdexcept>
2 #include <vector> // Это для вектора
3 #include "ring.hpp" // Это файл с определением нашего класса Ring
4
5 // Класс CircularBuffer реализует "закольцованный массив". В таком массиве
6 // после элемента с индексом size - 1 идёт элемент с индексом 0. Начало
7 // циклического буфера может располагаться в произвольной позиции в массиве.
8 // CircularBuffer поддерживает быстрое добавление элемента в конец или в начало,
9 // а также быстрое удаление элемента из конца или из начала.
10 class CircularBuffer {
11 public:
12     CircularBuffer(std::size_t size) :
13         data(size), // Создаём вектор размера size элементов типа int
14         indexBegin(0, size), // Это позиция, с которой начинаются данные в кольце.
15         size(0) // Количество элементов, которые лежат в циклическом буфере.
16     { }
17
18     // Операция чтения по номеру элемента
19     int operator[](std::size_t index) const { // Метод для чтения элементов
20         if (index >= Size()) // Кидаем исключение, если закончилось свободное место.
21             throw std::out_of_range("Invalid index!");
```

```
22
23 // Здесь выражение indexBegin + Ring(index) выполняется так:
24 // Сначала вызывается операция сложения для двух чисел типа Ring.
25 // Результатом этой операции является новое число типа Ring,
26 // которое неявным образом преобразуется в std::size_t, то есть в индекс вектора.
27 // Отметим, что конструкция indexBegin + index также скомпилируется,
28 // однако работать будет неправильно, а именно, для такой записи
29 // indexBegin сначала преобразовался бы в std::size_t, а затем
30 // сработала бы операция сложения для чисел типа std::size_t.
31 return data[indexBegin + Ring(index)];
32 }
33
34 // Операция записи по номеру элемента
35 int& operator[](std::size_t index) { // Метод для записи элементов
36     if (index >= Size()) // Кидаем исключение, если закончилось свободное место.
37         throw std::out_of_range("Invalid index!");
38
39     // Взятие элемента по индексу, вычисленному при помощи арифметики в кольце.
40     return data[indexBegin + Ring(index)];
41 }
42
43 // Кладём элемент в конец.
44 void PushBack(int elem) {
45     // Вычисляем позицию нового элемента с помощью арифметики в кольце.
46     data[indexBegin + Ring(size)] = elem;
47     size++; // Увеличиваем число элементов.
48 }
49
50 // Кладём элемент в начало.
51 void PushFront(int elem) {
52     // Вычисляем позицию нового элемента с помощью арифметики в кольце.
53     data[--indexBegin] = elem;
54     size++; // Увеличиваем число элементов.
55 }
56
57 // Удаляем элемент из конца.
58 void PopBack() {
59     size--;
60 }
61
62 // Удаляем элемент из начала.
63 void PopFront() {
64     indexBegin++;
65     size--;
66 }
67
68 // Возвращает последний элемент.
69 int Back() const {
70     // Позиция вычисляется с помощью арифметике в кольце.
71     return data[indexBegin + Ring(size - 1)];
72 }
```

```
73
74 // Возвращает первый элемент.
75 int Front() const {
76     // Объекты типа Ring неявно можно преобразовывать к std::size_t.
77     return data[indexBegin];
78 }
79
80 std::size_t Size() const { return size; } // Количество элементов в буфере.
81 std::size_t Capacity() const { return data.size(); } // Вместимость буфера.
82
83 private:
84     // Массив "закольцован". После индекса (data.size() - 1) идёт индекс 0.
85     std::vector<int> data;
86     Ring indexBegin; // С этого индекса начинается массив (это индекс первого эл-та)
87     std::size_t size; // Количество добавленных элементов
88
89     /*
90     Итераторы
91     */
92 };
```

---

**Замечание 1.1.** Отметим, что при компиляции кода класса `CircularBuffer` компилятор (например, `g++` версии 8.2.1) может выдавать предупреждение `"implicit conversion changes signedness: 'std::size_t'..."`, если при компиляции использовался флаг `-Wsign-conversion`. Для того, чтобы избежать этого предупреждения, следует использовать явное приведение типа `static_cast`. Поскольку данный флаг компиляции используется редко, то явное приведение типа было опущено.

**Замечание 1.2.** Для класса `CircularBuffer` не нужно писать сору-конструктор и операцию присваивания. В самом деле, он не содержит указателей, а сору-конструктор и операция присваивания шаблона `std::vector` сами скопируют все данные. Таким образом, для класса `CircularBuffer` подойдёт стандартная реализация сору-конструктора и операции присваивания, сгенерированные компилятором.

### 1.3 Итераторы циклического буфера

---

```
1 class CircularBuffer {
2     /*
3     Базовые методы
4     */
5
6     // Реализация итераторов
7 public:
8     // Итератор, который может изменять элементы, на которые указывает.
9     class Iterator {
10    public:
11        // Для своей работы наш итератор требует итератор на начало вектора, позицию
12        // начала буфера в векторе, а также позицию нашего итератора относительно начала
13        // буфера. Отметим, что итератор должен вести арифметику в кольце чисел по модулю
14        // size + 1, иначе (если бы размер кольца был size) при полностью заполненном буфере
```

```
15 // итератор на начало буфера совпал бы с итератором на конец буфера.
16 Iterator(std::vector<int>::iterator itBegin, Ring indexBegin, std::size_t pos) :
17     itBegin(itBegin), // Итератор на начало вектора (не путать с началом буфера)
18     indexBegin(indexBegin), // Начало буфера в векторе (к кольце по модулю size + 1)
19     pos(pos) // Позиция итератора относительно начало буфера
20 { }
21
22 // Разыменовываем итератор. Для корректной работы нужно вернуться в кольцо
23 // чисел по модулю size, поскольку индексы находятся именно в нём.
24 // Для этого создаём переменную reducedRing.
25 int& operator*() const {
26     Ring reducedRing(indexBegin, indexBegin.Size() - 1);
27     // reducedRing + Ring(pos) - это позиция в кольце чисел по модулю size,
28     // она прибавляется к итератору на начало вектора, а затем разыменовывается.
29     return *(itBegin + static_cast<std::ptrdiff_t>(reducedRing + Ring(pos)));
30 }
31
32 Iterator& operator++() { // Операция префиксного инкремента
33     pos++; // Просто сдвигаем позицию.
34     return *this;
35 }
36
37 Iterator& operator--() { // Операция префиксного декремента.
38     pos--; // Просто сдвигаем позицию.
39     return *this;
40 }
41
42 // Операция сложения итератора и числа возвращает новый итератор.
43 Iterator operator+(std::size_t shift) const {
44     return Iterator(itBegin, indexBegin, pos + shift);
45 }
46
47 // Операция разности итератора и числа возвращает новый итератор.
48 Iterator operator-(std::size_t shift) const {
49     return Iterator(itBegin, indexBegin, pos - shift);
50 }
51
52 Iterator& operator+=(std::size_t shift) {
53     pos += shift; // Просто сдвигаем позицию
54     return *this;
55 }
56
57 Iterator& operator-=(std::size_t shift) {
58     pos -= shift; // Просто сдвигаем позицию
59     return *this;
60 }
61
62 // Операция сравнения для итераторов. Можно было ограничиться сравнением
63 // pos и other.pos.
64 bool operator==(const Iterator& other) const {
65     return pos == other.pos && indexBegin == other.indexBegin &&
```

```
66         itBegin == other.itBegin;
67     }
68
69     bool operator!=(const Iterator& other) const {
70         return pos != other.pos || indexBegin != other.indexBegin ||
71             itBegin != other.itBegin;
72     }
73
74 private:
75     // Итератор на начало вектора. Он нужен для доступа к элементам вектора.
76     std::vector<int>::iterator itBegin;
77     // Начало циклического буфера в кольце по модулю size + 1.
78     const Ring indexBegin;
79     std::size_t pos; // Текущая позиция итератора относительно начала буфера.
80 };
81
82 // Итератор, который не может изменять элементы, на которые указывает.
83 class ConstIterator {
84 public:
85     // Для своей работы наш итератор требует итератор на начало вектора, позицию
86     // начала буфера в векторе, а также позицию нашего итератора относительно начала
87     // буфера. Отметим, что итератор должен вести арифметику в кольце чисел по модулю
88     // size + 1, иначе (если бы размер кольца был size) при полностью заполненном буфере
89     // итератор на начало буфера совпал бы с итератором на конец буфера.
90     ConstIterator(std::vector<int>::const_iterator itBegin, Ring indexBegin,
91                 std::size_t pos) :
92         itBegin(itBegin), // Итератор на начало вектора (не путать с началом буфера)
93         indexBegin(indexBegin), // Начало буфера в векторе (к кольце по модулю size + 1)
94         pos(pos) // Позиция итератора относительно начало буфера
95     { }
96
97     // Разыменовываем итератор. Для корректной работы нужно вернуться в кольцо
98     // чисел по модулю size, поскольку индексы находятся именно в нём.
99     // Для этого создаём переменную reducedRing.
100    int operator*() const {
101        Ring reducedRing(indexBegin, indexBegin.Size() - 1);
102        // reducedRing + Ring(pos) - это позиция в кольце чисел по модулю size,
103        // она прибавляется к итератору на начало вектора, а затем разыменовывается.
104        return *(itBegin + static_cast<std::ptrdiff_t>(reducedRing + Ring(pos)));
105    }
106
107    ConstIterator& operator++() { // Операция префиксного инкремента
108        pos++; // Просто сдвигаем позицию.
109        return *this;
110    }
111
112    ConstIterator& operator--() { // Операция префиксного декремента.
113        pos--; // Просто сдвигаем позицию.
114        return *this;
115    }
116
```



```
117 // Операция сложения итератора и числа возвращает новый итератор.
118 ConstIterator operator+(std::size_t shift) const {
119     return ConstIterator(itBegin, indexBegin, pos + shift);
120 }
121
122 // Операция разности итератора и числа возвращает новый итератор.
123 ConstIterator operator-(std::size_t shift) const {
124     return ConstIterator(itBegin, indexBegin, pos - shift);
125 }
126
127 ConstIterator& operator+=(std::size_t shift) {
128     pos += shift; // Просто сдвигаем позицию
129     return *this;
130 }
131
132 ConstIterator& operator--=(std::size_t shift) {
133     pos -= shift; // Просто сдвигаем позицию
134     return *this;
135 }
136
137 // Операция сравнения для итераторов. Можно было ограничиться сравнением
138 // pos и other.pos.
139 bool operator==(const ConstIterator& other) const {
140     return pos == other.pos && indexBegin == other.indexBegin &&
141         itBegin == other.itBegin;
142 }
143
144 bool operator!=(const ConstIterator& other) const {
145     return pos != other.pos || indexBegin != other.indexBegin ||
146         itBegin != other.itBegin;
147 }
148 private:
149 // Итератор на начало вектора. Он нужен для доступа к элементам вектора.
150 std::vector<int>::const_iterator itBegin;
151 // Начало циклического буфера в кольце по модулю size + 1.
152 const Ring indexBegin;
153 std::size_t pos; // Текущая позиция итератора относительно начала буфера.
154 };
155
156 // Следующие функции возвращают итераторы на начало и конец буфера.
157 // Напомним, что итераторы используют арифметику в кольце по модулю size + 1
158 // для того, чтобы в полностью заполненном буфере итератор на начало отличался
159 // от итератора на конец.
160 Iterator Begin() { // Итератор на начало
161     return Iterator(data.begin(), Ring(indexBegin, indexBegin.Size() + 1), 0);
162 }
163 Iterator End() { // Невалидный итератор на конец
164     return Iterator(data.begin(), Ring(indexBegin, indexBegin.Size() + 1), size);
165 }
166 ConstIterator Begin() const { // Константный итератор на начало
167     return ConstIterator(data.begin(), Ring(indexBegin, indexBegin.Size() + 1), 0);
```

```
168 }
169 ConstIterator End() const { // Невалидный константный итератор на конец
170     return ConstIterator(data.begin(), Ring(indexBegin, indexBegin.Size() + 1), size);
171 }
172 };
```

---

## 1.4 Пример использования

Приведём пример решения задачи

**Задача 1.** *Сдвинуть элементы массива на  $k$  позиций вправо.*

---

```
1 #include <iostream>
2 #include "circular_buffer.hpp" // Файл с определением класса CircularBuffer
3
4 // Функция, которая выполняет всю работу.
5 void Process(CircularBuffer& buffer, std::size_t k) {
6     for (std::size_t i = 0; i < k; i++) {
7         int elem = buffer.Back();
8         buffer.PopBack(); // Удаляем элемент из конца.
9         buffer.PushFront(elem); // Добавляем его в начало.
10    }
11 }
12
13 // Операция предназначена для использования класса CircularBuffer с потоками вывода
14 std::ostream& operator<<(std::ostream& out, const CircularBuffer& buffer) {
15     out << "[";
16     for (CircularBuffer::ConstIterator it = buffer.Begin(); it != buffer.End(); ++it) {
17         out << *it << " ";
18     }
19     out << "]";
20     return out;
21 }
22
23 int main() {
24     std::size_t size;
25     std::size_t k;
26
27     if (!(std::cin >> size >> k)) {
28         std::cout << "error" << std::endl;
29         return 1;
30     }
31
32     CircularBuffer buffer(size); // Заводим циклический буфер размера size.
33
34     for (std::size_t i = 0; i < size; i++) {
35         int elem = 0;
36
37         if (!(std::cin >> elem)) { // Считываем элемент.
38             std::cout << "error" << std::endl;
```

```
39     return 1;
40 }
41 // Добавляем считанный элемент в конец буфера.
42 buffer.PushBack(elem);
43 }
44
45 // Выводим буфер на экран с помощью определённой выше операции.
46 std::cout << "Initial buffer = " << buffer << std::endl;
47
48 Process(buffer, k); // Обрабатываем буфер.
49 // Выводим результат на экран.
50 std::cout << "Result = " << buffer << std::endl;
51 return 0;
52 }
```

---