

Разработка и реализация класса “Массив с динамически изменяемыми границами”

Содержание

1 Дек целых значений (модуль, система программирования C)	2
1.1 Спецификация структуры данных модуля	2
1.2 Спецификация операций модуля	2
1.3 Требования	4
2 Дек целых значений (класс, система программирования C++)	5
2.1 Спецификация классов	5
2.1.1 Спецификация класса IntDequeElement	5
2.1.2 Спецификация класса IntDeque	6
2.2 Требования	7
3 Дек целых значений (класс, переопределение операций)	8
3.1 Указания	10
3.2 Требования	10
4 Список целых значений (класс, наследование)	11
4.1 Указания	12
4.2 Требования	12
5 Массив целых чисел с динамически изменяемыми границами (класс, наследование)	14
5.1 Требования	16
5.1.1 Пример вывода информации на экран	16
6 Массив с динамически изменяемыми границами (факультативное задание)	17
6.1 Требования	18
6.1.1 Пример вывода информации на экран	19

Общие требования

Внимание! Запрещается предоставлять работу, выполненную не самостоятельно! В случае обнаружения плагиата, работа рассматриваться не будет!

Общие положения

В рамках данной работы массивы представляют собой одномерные изменяемые объекты, которые могут динамически сжиматься и расширяться. Каждый массив имеет нижнюю границу и последовательность элементов, пронумерованных, начиная с нижней границы. Все элементы массива относятся к одному и тому же типу [2].

Реализация массива с динамически изменяемыми границами будет базироваться на такой структуре данных, как дек или очередь с двумя концами (double-ended queue).

Дек — линейный список, в котором все включения и исключения (и обычно всякий доступ) делаются на обоих концах списка [1].

В соответствии с этим, выполнение работы состоит из нескольких этапов. Задание по каждому последующему этапу выдаётся после завершения выполнения текущего этапа.

Разработка и реализация функций и методов для каждого этапа должна выполняться **строго** в соответствии с приведёнными в задании спецификациями этих функций и методов. Реализации этапов, выполненные с отступлениями от заданных спецификаций, рассматриваться не будут.

Перед передачей выполненной реализации на проверку данная реализация должна быть протестирована разработчиком.

Для успешной сдачи работы разработчик должен понимать и уметь объяснить все приведённые в работе объявления модулей, классов и шаблонов классов и их реализации.

1 Дек целых значений (модуль, система программирования C)

Разработка, реализация и тестирование модуля для работы с деком **целых** значений. Инструмент: система программирования C [4].

1.1 Спецификация структуры данных модуля

```
/* Описание реализации элемента дека */
struct deque_element {
    int element;
    /* значение элемента */

    struct deque_element * next;
    /* указатель на следующий элемент */

    struct deque_element * prev;
    /* указатель на предыдущий элемент */
};

/* Описание реализации дека */
struct intdeque {
    struct deque_element * left;
    /* указатель на левый элемент дека */

    struct deque_element * right;
    /* указатель на правый элемент дека */

    int buffer;
    /* значение возвращаемого элемента для операций
       remove_left, remove_right */
};

typedef struct intdeque * intdeque;
```

1.2 Спецификация операций модуля

- Создание дека:

```
struct intdeque * create_deque();
```

Функция размещает в памяти новый дек. Функция возвращает указатель на созданный дек. Если дек не может быть создан, то функция возвращает значение NULL.

- Добавление элемента в дек с левого края:

```
int* add_left(struct intdeque * pdeque, int elem);
```

Функция добавляет новый элемент в дек, соответствующий структуре [pdeque], с левого края.

Значение нового элемента передаётся через параметр [elem].

В случае успешного выполнения функция помещает значение [elem] в поле [buffer] дека, соответствующего структуре [pdeque], и, в качестве результата, возвращает значение адреса области памяти, соответствующей полю [buffer].

В противном случае функция возвращает значение NULL.

- Добавление элемента в дек с правого края:

```
int* add_right(struct intdeque * pdeque, int elem);
```

Функция добавляет новый элемент в дек, соответствующий структуре [pdeque], с правого края.

Значение нового элемента передаётся через параметр [elem].

В случае успешного выполнения функция помещает значение [elem] в поле [buffer] дека, соответствующего структуре [pdeque], и, в качестве результата, возвращает значение адреса области памяти, соответствующей полю [buffer].

В противном случае функция возвращает значение NULL.

- Удаление элемента из дека с левого края:

```
int* remove_left(struct intdeque * pdeque);
```

Функция удаляет элемент из дека, соответствующего структуре [pdeque], с левого края.

В случае успешного выполнения функция помещает значение удаляемого элемента в поле [buffer] дека, соответствующего структуре [pdeque], и, в качестве результата, возвращает значение адреса области памяти, соответствующей полю [buffer].

В противном случае функция возвращает значение NULL.

- Удаление элемента из дека с правого края:

```
int* remove_right(struct intdeque * pdeque);
```

Функция удаляет элемент из дека, соответствующего структуре [pdeque], с правого края.

В случае успешного выполнения функция помещает значение удаляемого элемента в поле [buffer] дека, соответствующего структуре [pdeque], и, в качестве результата, возвращает значение адреса области памяти, соответствующей полю [buffer].

В противном случае функция возвращает значение NULL.

- Удаление дека:

```
struct intdeque * delete_deque(struct intdeque * pdeque);
```

Функция удаляет дек, соответствующий структуре [pdeque].

В качестве результата функция возвращает значение NULL.

1.3 Требования

В процессе выполнения этапа должны быть созданы и предоставлены на проверку следующие файлы:

- `intdeque.h` — файл с описанием структуры данных и функций (операций) модуля;
- `intdeque.c` — файл с исходным текстом реализации функций (операций) модуля;
- `test01.c` — файл, содержащий исходный текст на языке программирования C с реализацией функции `main()`, обеспечивающей тестирование разработанного модуля.

Тестовая программа должна обеспечить создание и работу с двумя деками с именами `pdeque01` и `pdeque02` соответственно. Тестовая программа должна обеспечить добавление не менее трёх элементов с левой стороны дека и не менее трёх элементов с правой стороны дека для каждого из деков. При каждом действии на экран должна выводиться следующая информация:

- имя вызываемой функции;
- имя дека, для которого используется вызов функции;
- передаваемые целые значения (для функций `add_left()`, `add_right()`);
- результат выполнения функции.

Формат вывода информации на экран (примеры для каждой из функций):

- функция `create_deque`:

```
pdeque01 = create_deque() == OK
— если функция возвращает значение, не равное NULL;
```

```
pdeque01 = create_deque() == NOT OK
— если функция возвращает значение, равное NULL;
```

- функции `add_left()` и `add_right()`:

```
add_left(pdeque01, 10) == 10
— если функция выполнена успешно, то необходимо после знака == вывести значение,
хранящееся в области памяти, соответствующей полю buffer структуры, реализующей
дек;
```

```
add_left(pdeque01, 10) == NULL
— если функция не может быть выполнена (например, нет памяти для добавления эле-
мента в дек);
```

- функции `remove_left()` и `remove_right()`:

```
remove_left(pdeque01) == 10
— если функция выполнена успешно, то необходимо после знака == вывести значение,
хранящееся в области памяти, соответствующей полю buffer структуры, реализующей
дек;
```

```
remove_left(pdeque01) == NULL
— если функция не может быть выполнена (например, в деке нет элементов);
```

- функция `delete_deque()`:

```
delete_deque(pdeque01).
```

В тестовой программе общее количество вызовов функций `remove_left()` и `remove_right()` должно превышать общее количество вызовов функций `add_left()` и `add_right()` для каждого из деков.

После вызова функции `delete_deque()` необходимо вызвать функции `remove_left()` и `remove_right()` для каждого из деков, в результате чего должно быть обеспечено предсказуемое поведение программы.

Тестовая программа должна быть реализована без запросов на ввод данных пользователем с экрана.

2 Дек целых значений (класс, система программирования C++)

Разработка, реализация и тестирование класса для работы с деком **целых** значений.
Инструмент: система программирования C++ [5] [3].

2.1 Спецификация классов

Разработка класса “Дек целых значений” выполняется на основе результатов этапа 1.
В соответствии с этим предлагается разработать и реализовать следующие два класса:

- **IntDequeElement** — класс, содержащий структуру данных и методы для работы с элементом дека;
- **IntDeque** — класс, содержащий структуру данных и методы для работы с деком.

2.1.1 Спецификация класса IntDequeElement

```
class IntDequeElement{
    int element;
    /* значение элемента */

    IntDequeElement * next;
    /* указатель на следующий элемент */

    IntDequeElement * prev;
    /* указатель на предыдущий элемент */

public:
    IntDequeElement();
    /* Конструктор по умолчанию для создания элемента списка.
       Обеспечивает задание следующих начальных значений элементам класса:
       element <- 0
       next <- NULL
       prev <- NULL
    */

    IntDequeElement(int _element);
    /* Конструктор для создания элемента списка, обеспечивающий
       задание начального значения [_element] элементу списка:
       element <- _element
       next <- NULL
       prev <- NULL
    */

    IntDequeElement(int _element, IntDequeElement * _prev, IntDequeElement * _next);
    /* Конструктор для создания элемента списка, обеспечивающий
```

```

        задание начального значения [_element] элементу списка,
        задание начального значения [_prev] указателю на предыдущий элемент списка
        задание начального значения [_next] указателю на следующий элемент списка */

void SetElement(int _element);
/* Присваивание значения [_element] элементу списка */

int GetElement();
/* Получение значения элемента списка */

void SetNext(IntDequeElement * _next);
/* Присваивание значения [_next] полю next */

IntDequeElement * GetNext();
/* Получение значения поля next */

void SetPrev(IntDequeElement * _prev);
/* Присваивание значения [_prev] полю prev */

IntDequeElement * GetPrev();
/* Получение значения поля prev */
};

```

2.1.2 Спецификация класса IntDeque

```

class IntDeque{
    IntDequeElement * left;
    /* указатель на крайний левый элемент списка */

    IntDequeElement * right;
    /* указатель на крайний правый элемент списка */

    int buffer;
    /* Значение добавленного/удалённого элемента списка */

public:

    IntDeque();
    /* Конструктор по умолчанию для создания дека:
        left <- NULL
        right <- NULL
        buffer <- 0
    */

    IntDeque(IntDeque& _deque);
    /* Конструктор копирования.
        Обеспечивает создание дека, являющегося копией дека _deque.
    */

    int* AddLeft(int element);
    /* Добавление элемента со значением [element] слева.
        После добавления элемента поле buffer получает значение [element].
        В случае успешного выполнения, в качестве результата
        возвращает значение указателя на поле buffer.
        Иначе, возвращает значение NULL.
    */

```

```

int* AddRight(int element);
/* Добавление элемента со значением [element] справа.
   После добавления элемента поле buffer получает значение [element].
   В случае успешного выполнения, в качестве результата
   возвращает значение указателя на поле buffer.
   Иначе, возвращает значение NULL.
*/

int* RemoveLeft();
/* Удаление слева.
   Поле buffer получает значение удалённого элемента.
   В качестве результата возвращает значение указателя на поле buffer.
   Если в списке нет элементов, то, в качестве результата,
   возвращает значение NULL.
*/

int* RemoveRight();
/* Удаление справа.
   Поле buffer получает значение удалённого элемента.
   В качестве результата возвращает значение указателя на поле buffer.
   Если в списке нет элементов, то, в качестве результата,
   возвращает значение NULL.
*/

int GetElement();
/* Получение значения последнего добавленного или удалённого элемента
   (получение значения поля buffer).
*/

int IsEmpty();
/* Проверка дека на пустоту (отсутствие элементов в деке).
   Если в деке есть элементы, то операция в качестве результата возвращает
   значение 0.
   Иначе, операция в качестве результата возвращает значение 1 (дек пуст).
*/

~IntDeque();
/* Деструктор.
   Обеспечивает удаление всего дека.
*/

};

```

2.2 Требования

В процессе выполнения этапа должны быть созданы и предоставлены на проверку следующие файлы:

- `intdeque.hpp` — файл с описанием классов;
- `intdeque.cpp` — файл с исходным текстом реализации методов классов;
- `test02.cpp` — файл, содержащий исходный текст на языке программирования C++ с реализацией функции `main()`, обеспечивающей тестирование работы с объектами разработанных классов.

Тестовая программа должна обеспечить создание и работу с тремя деками с именами `pdeque01`, `pdeque02`, `pdeque03` соответственно. Тестовая программа должна обеспечить добавление не менее трёх элементов с левой стороны дека и не менее трёх элементов с правой стороны дека для деков `pdeque01`, `pdeque02`. Дек `pdeque03` при создании должен быть проинициализирован значением одного из деков `pdeque01` или `pdeque02`, причём дек `pdeque03` должен создаваться в тот момент времени когда дек `pdeque01` или `pdeque02` соответственно не является пустым (содержит элементы). При каждом действии на экран должна выводиться следующая информация:

- имя дека и вызываемый метод;
- передаваемые целые значения (для методов `AddLeft()`, `AddRight()`);
- результат выполнения метода.

Формат вывода информации на экран (примеры для каждого из методов):

- методы `AddLeft()`, `AddRight()`, `GetElement()`:

```
pdeque01.AddLeft(10); pdeque01.GetElement() == 10
— необходимо после знака == вывести результат выполнения метода GetElement();
```

- методы `RemoveLeft()`, `RemoveRight()`, `GetElement()`:

```
pdeque01.RemoveLeft(); pdeque01.GetElement() == 10
— если метод RemoveLeft() (RemoveRight()) выполнен успешно, то необходимо после
знака == вывести результат выполнения метода GetElement();
```

```
pdeque01.RemoveLeft() == NULL
— если метод не может быть выполнен (например, в деке нет элементов);
```

- конструктор копирования `IntDeque(IntDeque& _deque)` (пример приведён для случая, когда создаваемый деку `pdeque03` инициализируется значением дека `pdeque01`):

```
IntDeque pdeque03 = pdeque01
```

В тестовой программе общее количество вызовов методов `RemoveLeft()` и `RemoveRight()` должно превышать общее количество вызовов метода `AddLeft()` и `AddRight()` для каждого из деков.

Тестовая программа должна быть реализована без запросов на ввод данных пользователем с экрана.

Для динамического выделения и удаления памяти использовать операции языка программирования C++ **new** и **delete** (см. [3]).

При выполнении этапа **ЗАПРЕЩАЕТСЯ** использовать классы стандартной библиотеки системы программирования C++.

3 Дек целых значений (класс, переопределение операций)

Дополнить класс дека целых чисел, разработанный на этапе 2, следующими операциями:

- *element* + *Deque* — добавление в дек *Deque* целочисленного элемента со значением *element* слева.

После добавления элемента поле *buffer* получает значение *element*.

В случае успешного выполнения, в качестве результата операция возвращает значение указателя на поле *buffer*.

Иначе, операция возвращает значение *NULL*.

- *Deque + element* — добавление в дек *Deque* целочисленного элемента со значением *element* справа.

После добавления элемента поле *buffer* получает значение *element*.

В случае успешного выполнения, в качестве результата операция возвращает значение указателя на поле *buffer*.

Иначе, операция возвращает значение *NULL*.

- — — *Deque* — префиксная операция “минус-минус”, удаление элемента из дека *Deque* слева.

Поле *buffer* получает значение удалённого элемента.

В качестве результата операция возвращает значение указателя на поле *buffer*.

Если в деке *Deque* нет элементов, то, в качестве результата, операция возвращает значение *NULL*.

- *Deque — —* — постфиксная операция “минус-минус”, удаление элемента из дека *Deque* справа.

Поле *buffer* получает значение удалённого элемента.

В качестве результата операция возвращает значение указателя на поле *buffer*.

Если в деке *Deque* нет элементов, то, в качестве результата, операция возвращает значение *NULL*.

- *Deque2 = Deque1* — присваивание значения объекта *Deque1* объекту *Deque2*. Например, если до выполнения операции *Deque1 = Deque2* деки *Deque1* и *Deque2* имеют следующие значения:

- *Deque1* : 10, 30, 50
- *Deque2* : 50, 70, 77, 97, 99

то в результате выполнения операции присваивания будет следующий результат:

- *Deque1* : 50, 70, 77, 97, 99
- *Deque2* : 50, 70, 77, 97, 99

Операция присваивания должна быть реализована таким образом, чтобы возможна была запись выражений, подобных следующему:

$$a = b = c;$$

где *a, b, c* — объекты класса *IntDeque*.

- *Deque1 == Deque2* — сравнение объектов *Deque1* и *Deque2* на равенство.

Операция возвращает целочисленное значение 1, если дек *Deque1* равен деку *Deque2*.

Иначе, операция возвращает значение 0.

Два дека равны между собой, если они содержат элементы с одинаковыми значениями, причём порядок, в котором эти элементы расположены в каждом из деков, также совпадает. Например, следующие два дека равны между собой:

- *Deque1* : 10, 20, 30, 40, 50
- *Deque2* : 10, 20, 30, 40, 50

Во всех остальных случаях деки не равны между собой.

Инструмент: система программирования C++ [5] [3].

3.1 Указания

Для динамического выделения и удаления памяти использовать операции языка программирования C++ **new** и **delete** (см. [3], стр. 53–55, раздел “Инициализация указателей”).

Изучить возможности перегрузки операций в языке программирования C++ (см. [3]), стр. 189–197, раздел “Перегрузка операций”).

3.2 Требования

В процессе выполнения этапа должны быть созданы и предоставлены на проверку следующие файлы:

- `intdeque.hpp` — файл с описанием классов;
- `intdeque.cpp` — файл с исходным текстом реализации методов классов;
- `test03.cpp` — файл, содержащий исходный текст на языке программирования C++ с реализацией функции `main()`, обеспечивающей тестирование работы с объектами раз-работанных классов и обработку исключительных ситуаций.

Тестовая программа должна обеспечить создание и работу с двумя деками с именами `pdeque01` и `pdeque02` соответственно. Тестовая программа должна обеспечить добавление не менее трёх элементов с левой стороны дека и не менее трёх элементов с правой стороны дека для каждого из деков. При каждом действии на экран должна выводиться следующая информация:

- имя дека и вызываемый метод;
- передаваемые целые значения (для операции `+`);
- результат выполнения метода.

Формат вывода информации на экран (примеры для каждой из операций):

- операция `+`:
`10 + pdeque01 == OK`
— в случае успешного выполнения операции;
`10 + pdeque01 == NoMemory`
— в случае возникновения ошибки;
`pdeque01 + 20 == OK`
— в случае успешного выполнения операции;
`pdeque01 + 20 == NoMemory`
— в случае возникновения ошибки.
- операция `--`:
`--pdeque01 == 10`
— в случае успешного выполнения операции;
`--pdeque01 == DequeIsEmpty`
— в случае, когда дек пуст;
`pdeque01-- == 20`
— в случае успешного выполнения операции;
`pdeque01-- == DequeIsEmpty`
— в случае, когда дек пуст;
- операция `==`:
`pdeque01 == pdeque02`
— в случае, когда деки `pdeque01` и `pdeque02` равны между собой;

`pdeque01 != pdeque02`

— в случае, когда деки `pdeque01` и `pdeque02` не равны между собой.

Требований к формату вывода на экран результата выполнения операций присваивания не предъявляется.

В тестовой программе общее количество операций — должно превышать общее количество операций +.

Тестовая программа должна быть реализована без запросов на ввод данных пользователем с экрана.

При выполнении этапа **ЗАПРЕЩАЕТСЯ** использовать классы стандартной библиотеки системы программирования C++.

4 Список целых значений (класс, наследование)

Разработать и реализовать класс *IntList*, обеспечивающий работу с двунаправленным **списком** целых значений. Данный класс должен быть классом-наследником класса *IntDeque*, который был разработан и реализован в рамках предыдущего этапа.

Поведение объектов класса *IntList* определяется набором следующих методов и операций:

- *IntList()* — конструктор по умолчанию; аналогичен конструктору *IntDeque()*, при этом указатель на текущий элемент списка должен принимать значение *NULL*;
- *IntList(IntList& plist)*; — конструктор копирования; аналогичен конструктору копирования *IntDeque& pdeque*, при этом указатель на текущий элемент должен принимать значение указателя на тот же самый по счёту элемент от первого элемента созданного списка, что и текущий элемент по счёту в исходном списке; например, если в списке *plist* указатель на текущий элемент соответствует третьему по счёту от начала списка элементу, то и во вновь созданном списке указатель на текущий элемент должен соответствовать третьему по счёту от начала списка элементу;
- *IntDequeElement * GoToLeft()* — переход к крайнему левому элементу списка; в качестве результата данный метод возвращает адрес объекта класса *IntDequeElement*, являющегося самым крайним левым элементом списка;
- *IntDequeElement * GoToRight()* — переход к крайнему правому элементу списка; в качестве результата данный метод возвращает адрес объекта класса *IntDequeElement*, являющегося самым крайним правым элементом списка;
- *IntDequeElement * GoToNext()* — переход к следующему элементу списка; если текущий элемент списка не определён (указатель на текущий элемент имеет значение *NULL*) или текущий элемент является крайним правым элементом (нет следующего элемента), то данный метод должен возвращать в качестве результата значение *NULL* и, при этом, текущий элемент не изменяется; иначе, текущим элементом становится следующий элемент и в качестве результата метод возвращает адрес этого элемента;
- *IntDequeElement * GoToPrev()* — переход к предыдущему элементу списка; если текущий элемент списка не определён (указатель на текущий элемент имеет значение *NULL*) или текущий элемент является крайним левым элементом (нет предыдущего элемента), то данный метод должен возвращать в качестве результата значение *NULL* и, при этом, текущий элемент не изменяется; иначе, текущим элементом становится предыдущий элемент и в качестве результата метод возвращает адрес этого элемента;
- *int * Fetch()* — получение значения текущего элемента списка; метод помещает значение текущего элемента в поле *buffer* соответствующего объекта класса *IntList* (поле *buffer* должно быть унаследовано из класса *IntDeque*) и в качестве результата возвращает значение адреса поля *buffer*; если текущий элемент списка не определён (указатель на текущий элемент имеет значение *NULL*), то данный метод возвращает значение *NULL*;

- $int * Store(int\ elem)$ — изменение значения текущего элемента списка на значение $elem$; значение $elem$ также присваивается полю $buffer$ соответствующего объекта класса $IntList$ (поле $buffer$ должно быть унаследовано из класса $IntDeque$) и в качестве результата возвращает значение адреса поля $buffer$; если текущий элемент списка не определён (указатель на текущий элемент имеет значение $NULL$), то данный метод возвращает значение $NULL$;
- $element + List$ — добавление в список $List$ целочисленного элемента $element$ слева; результат выполнения данной операции аналогичен результату выполнения операции $element + Deque$ класса $IntDeque$, при этом, при успешном выполнении операции, текущим элементом становится добавленный элемент; в случае ошибки текущий элемент не изменяется;
- $List + element$ — добавление в список $List$ целочисленного элемента $element$ справа; результат выполнения данной операции аналогичен результату выполнения операции $Deque + element$ класса $IntDeque$, при этом, при успешном выполнении операции, текущим элементом становится добавленный элемент; в случае ошибки текущий элемент не изменяется;
- $List2 = List1$ — присваивание значения объекта $List1$ объекту $List2$; реализуется аналогично операции $Deque2 = Deque1$ для класса $IntDeque$, при этом, значение текущего элемента в объекте $List2$ должно быть установлено так же, как это определено в описании конструктора копирования $IntList(IntList\&plist)$ выше;
- $List1 == List2$ — сравнение объектов $List1$ и $List2$ на равенство; реализуется аналогично операции $Deque1 == Deque2$; в дополнение к этому, два списка являются равными, если соблюдаются условия для операции $Deque1 == Deque2$ и указатели на текущий элемент в этих списках указывают на одну и ту же по счёту позицию;
- $-- List$ — удаление крайнего левого элемента списка; результат выполнения данной операции аналогичен результату выполнения операции $-- Deque$ класса $IntDeque$; при этом, текущим элементом списка становится элемент, следующий за удаляемым элементом; если удалён последний элемент стека, то указатель на текущий элемент принимает значение $NULL$;
- $List --$ — удаление крайнего правого элемента списка; результат выполнения данной операции аналогичен результату выполнения операции $Deque --$ класса $IntDeque$; при этом, текущим элементом списка становится предыдущий по отношению к удаляемому элемент; если удалён последний элемент стека, то указатель на текущий элемент принимает значение $NULL$.

4.1 Указания

В процессе выполнения данного этапа допускается модификация классов $IntDeque$ и $IntDequeElement$, но при условии, что внесённые в указанные классы изменения не повлекут изменений тестовой программы, реализованной на предыдущем этапе. При этом запрещается доступ к полю $buffer$ из методов и функций, не являющихся членами классов $IntDeque$ или $IntList$. Также, должен быть запрещён доступ к любым методам классов $IntDeque$ и $IntList$, не отражённых в спецификации данных классов. Рассмотреть возможность использования ключа доступа *protected* для части элементов классов $IntDeque$ и $IntList$.

Изучить возможности наследования и виртуальных функций в языке программирования C++ (см. [3]), стр. 200–207, разделы “Наследование”, “Ключи доступа”, “Простое наследование”, “Виртуальные методы”).

Инструмент: система программирования C++ [5] [3].

4.2 Требования

В процессе выполнения этапа должны быть созданы и предоставлены на проверку следующие файлы:

- *intdeque.hpp* — файл с описанием классов *IntDequeElement* и *IntDeque*;
- *intdeque.cpp* — файл с исходным текстом реализации методов классов *IntDequeElement* и *IntDeque*;
- *intlist.hpp* — файл с описанием класса *IntList*;
- *intlist.cpp* — файл с исходным текстом реализации методов класса *IntList*;
- *test04.cpp* — файл, содержащий исходный текст на языке программирования C++ с реализацией функции `main()`, обеспечивающей тестирование работы с объектами разработанных классов и обработку исключительных ситуаций.

Тестовая программа должна обеспечить создание и работу как минимум с тремя списками с именами `plist01`, `plist02` и `plist03` соответственно. Тестовая программа должна обеспечить добавление не менее трёх элементов с левой стороны списка и не менее трёх элементов с правой стороны списка для каждого из списков, а также, получение и изменение значений некоторых из элементов, не являющихся крайними элементами списка. При каждом действии на экран должна выводиться следующая информация:

- имя списка и вызываемый метод;
- передаваемые целые значения (для операции `+` и метода `Store`);
- результат выполнения метода.

Формат вывода информации на экран (примеры):

- операция `+`:

```
10 + plist01 == OK
— в случае успешного выполнения операции;
10 + plist01 == NoMemory
— в случае возникновения ошибки;
plist01 + 20 == OK
— в случае успешного выполнения операции;
plist01 + 20 == NoMemory
— в случае возникновения ошибки.
```

- операция `--`:

```
--plist01 == 10
— в случае успешного выполнения операции;
--plist01 == ListIsEmpty
— в случае, когда список пуст;
plist01-- == 20
— в случае успешного выполнения операции;
plist01-- == ListIsEmpty
— в случае, когда список пуст;
```

- методы `GoToLeft()` и `GoToRight()`:

```
plist01.GoToLeft() == OK
```

- `GoToNext()` и `GoToPrev()`:

```
plist01.GoToNext() == OK
— в случае успешного выполнения метода;
plist01.GoToNext() == NoCurrentElement
— в случае, когда указатель на текущий элемент имеет значение NULL;
```

```

plist01.GoToNext() == NoNextElement
— в случае, если указатель на следующий элемент имеет значение NULL;

plist01.GoToPrev() == NoPrevElement
— в случае, если указатель на предыдущий элемент имеет значение NULL;

```

- **Fetch():**

```

plist01.Fetch() == 10
— в случае успешного выполнения метода;

plist01.Fetch() == NoCurrentElement — когда указатель на текущий элемент
имеет значение NULL;

```

- **Store(elem):**

```

plist01.Store(20) == OK
— в случае успешного выполнения метода;

plist01.Store(20) == NoCurrentElement
— когда указатель на текущий элемент имеет значение NULL;

```

Помимо представленных выше операций, в тестовой программе в обязательном порядке должно быть проверено выполнение следующих операций (пример):

```

IntList plist03 = plist02;

plist01 = plist02 = plist03;

plist01 == plist02

```

В тестовой программе общее количество операций — должно превышать общее количество операций +.

Тестовая программа должна быть реализована без запросов на ввод данных пользователем с экрана.

Для динамического выделения и удаления памяти использовать операции языка программирования C++ **new** и **delete** (см. [3]).

При выполнении этапа **ЗАПРЕЩАЕТСЯ** использовать классы стандартной библиотеки системы программирования C++.

5 Массив целых чисел с динамически изменяемыми границами (класс, наследование)

Разработать и реализовать класс **IntArray**, обеспечивающий работу с массивом целых чисел с изменяемыми границами [2]. При выполнении работы использовать реализованные на предыдущих этапах классы **IntDequeElement**, **IntDeque**, **IntList**.

В рамках данной работы массивы представляют собой одномерные изменяемые объекты, которые могут динамически сжиматься и расширяться. Каждый массив имеет нижнюю границу и последовательность элементов, пронумерованных целыми значениями, начиная с нижней границы с шагом 1.

Для массива должны быть определены следующие операции:

- Конструкторы, обеспечивающие создание объектов класса **IntArray**. Конструкторы должны обеспечить следующие возможности создания массивов:
 - **IntArray a** — создание пустого массива с именем **a**; при добавлении первого элемента массива, индекс этого элемента должен принять значение 1;
 - **IntArray a(lb)** — создание пустого массива с именем **a**; при добавлении первого элемента массива, индекс этого элемента должен принять значение, заданное целочисленным параметром **lb**, в качестве которого также допускается использование целочисленной константы;

- `IntArray a(b)` — конструктор копирования, обеспечивающий создание массива с именем `a`, являющегося копией уже существующего массива `b` (массив `b` также является объектом класса `IntArray`), то есть, массив `a` после создания должен иметь те же самые индексы первого и последнего элементов, тот же набор, порядок и значения элементов, что и массив `b`;
- `IntArray a(lb, cnt, val)` — создание массива `a`, состоящего из элементов, количество которых соответствует значению целочисленного параметра `cnt`, причём каждый элемент имеет значение заданное целочисленным параметром `val`, а индекс первого элемента массива задаётся целочисленным параметром `lb`; любой из перечисленных параметров может быть задан целочисленной константой;
- Операции, обеспечивающие работу с массивом:
 - `a.Low()` — получение **значения** индекса первого по счёту элемента массива `a` (получение значения нижней границы массива);
 - `a.High()` — получение **значения** индекса последнего по счёту элемента массива `a` (получение значения верхней границы массива);
 - `a.Size()` — получение текущего количества элементов массива `a`;
 - `a[i]` — получение **значения** элемента массива `a` с индексом `i`;
 - `a[i] = intval` — присваивание элементу массива `a` с индексом `i` целочисленного значения, заданного выражением `intval`; целочисленная переменная и целочисленная константа являются частными случаями целочисленного выражения;
 - `a + intval` — добавление к массиву `a` элемента с целочисленным значением `intval` со стороны верхней границы; в результате выполнения этой операции количество элементов и значение верхней границы увеличиваются на единицу и, соответственно, добавленный элемент становится последним по счёту элементом массива;
 - `intval + a` — добавление к массиву `a` элемента с целочисленным значением `intval` со стороны нижней границы; в результате выполнения этой операции количество элементов массива увеличивается на единицу, а значение нижней границы уменьшается на единицу и, соответственно, добавленный элемент становится первым по счёту элементом массива;
 - `a--` — удаление элемента массива `a` со стороны верхней границы (удаление последнего по счёту элемента массива); в качестве результата данная операция возвращает **значение** удаляемого элемента массива; при удалении количество элементов в массиве и значение верхней границы уменьшаются на единицу;
 - `--a` — удаление элемента массива `a` со стороны нижней границы (удаление первого по счёту элемента); в качестве результата данная операция возвращает **значение** удаляемого элемента массива; при удалении количество элементов в массиве уменьшается на единицу, а значение нижней границы увеличивается на единицу;
 - `a = b` — присваивание массиву `a` значения массива `b`; в результате выполнения данной операции массив `a` становится копией массива `b`; если перед выполнением операции присваивания в массиве `a` есть элементы, то они удаляются.
- Деструктор, обеспечивающий удаление элементов массива.

В процессе выполнения данного этапа допускается модификация классов `IntDeque`, `IntDequeElement` и `IntList`, но при условии, что внесённые в указанные классы изменения не повлекут изменений тестовой программы, реализованной на предыдущем этапе.

Рассмотреть возможность часть методов модифицированных классов `IntDeque`, `IntDequeElement` и `IntList` реализовать как защищённые методы (вид доступа `protected`).

Инструмент: система программирования C++ [5] [3].

5.1 Требования

У любого пустого массива значение нижней границы массива должно быть на единицу больше значения верхней границы массива.

Доступ к дополнительным методам классов `IntDequeElement`, `IntDeque`, `IntList`, явно не отражённым в спецификациях этих классов, должны быть запрещён для любых функций и методов, не являющихся членами указанных классов или их классов-наследников.

В процессе выполнения этапа должны быть созданы и предоставлены на проверку следующие файлы:

- `intdeque.hpp` — файл с описанием классов `IntDequeElement` и `IntDeque`;
- `intdeque.cpp` — файл с исходным текстом реализации методов классов `IntDequeElement` и `IntDeque`;
- `intlist.hpp` — файл с описанием класса `IntList`;
- `intlist.cpp` — файл с исходным текстом реализации методов класса `IntList`;
- `intarray.hpp` — файл с описанием класса `IntArray`;
- `intarray.cpp` — файл с исходным текстом реализации методов класса `IntArray`;
- `test05.cpp` — файл, содержащий исходный текст на языке программирования C++ с реализацией функции `main()`, обеспечивающей тестирование работы с объектами разработанных классов и обработку исключительных ситуаций.

Тестовая программа должна обеспечить создание и работу как минимум с двумя массивами с именами `parray01` и `parray02` соответственно. Тестовая программа должна обеспечить добавление не менее трёх элементов со стороны нижней границы массива и не менее трёх элементов со стороны верхней границы массива для каждого из массивов, а также, получение и изменение значений некоторых из элементов, не являющихся крайними элементами массива. При каждом действии на экран должна выводиться следующая информация:

- выполняемое действие;
- результат выполнения действия;
- состояние массива после выполнения действия.

5.1.1 Пример вывода информации на экран

```
IntArray a
OK
++
!!
++
-

a.Low() == ArrayIsEmpty
ArrayIsEmpty
++
!!
++
-

a + 10
OK
+-----+
!      10!
```



```

+-----+
      1

20 + a
OK
+-----+-----+
!    20!    10!
+-----+-----+
      0      1

a[1] == 10
OK
+-----+-----+
!    20!    10!
+-----+-----+
      0      1

```

В тестовой программе общее количество операций — должно превышать общее количество операций +.

Тестовая программа должна быть реализована без запросов на ввод данных пользователем с экрана.

Для динамического выделения и удаления памяти использовать операции языка программирования C++ **new** и **delete** (см. [3]).

При выполнении этапа **ЗАПРЕЩАЕТСЯ** использовать классы стандартной библиотеки системы программирования C++.

6 Массив с динамически изменяемыми границами (факкультативное задание)

Разработать и реализовать шаблон класса **Array**, обеспечивающий работу с массивами данных различных типов с изменяемыми границами [2]. При выполнении работы реализованные на предыдущих этапах классы **IntDequeElement**, **IntDeque**, **IntList** могут быть переработаны в шаблоны классов **DequeElement**, **Deque**, **List**.

В рамках данной работы массивы представляют собой одномерные изменяемые объекты, которые могут динамически сжиматься и расширяться. Каждый массив имеет нижнюю границу и последовательность элементов, пронумерованных, начиная с нижней границы. **Все элементы конкретного массива относятся к одному и тому же типу.**

Для массива должны быть определены следующие операции:

- Конструкторы, обеспечивающие создание объектов класса **Array**. Конструкторы должны обеспечить следующие возможности создания массивов:
 - **Array a** — создание пустого массива с именем **a**; при добавлении первого элемента массива, индекс этого элемента должен принять значение 1;
 - **Array a(lb)** — создание пустого массива с именем **a**; при добавлении первого элемента массива, индекс этого элемента должен принять значение, заданное целочисленным параметром **lb**, в качестве которого также допускается использование целочисленной константы;
 - **Array a(b)** — конструктор копирования, обеспечивающий создание массива с именем **a**, являющегося копией уже существующего массива **b** (массив **b** также является объектом класса **Array**), то есть, массив **a** после создания должен иметь те же самые индексы первого и последнего элементов, тот же набор, порядок и значения элементов, что и массив **b**;

- `Array a(lb, cnt, val)` — создание массива `a`, состоящего из элементов, количество которых соответствует значению целочисленного параметра `cnt`, причём каждый элемент имеет значение заданное параметром `val`, а индекс первого элемента массива задаётся целочисленным параметром `lb`; любой из перечисленных параметров может быть задан константой соответствующего типа;
- Операции, обеспечивающие работу с массивом:
 - `a.Low()` — получение значения индекса первого по счёту элемента массива `a` (получение значения нижней границы массива);
 - `a.High()` — получение значения индекса последнего по счёту элемента массива `a` (получение значения верхней границы массива);
 - `a.Size()` — получение текущего количества элементов массива `a`;
 - `a[i]` — получение значения элемента массива `a` с индексом `i`;
 - `a[i] = val` — присваивание элементу массива `a` с индексом `i` значения, заданного выражением `val`; переменная (объект) и константа соответствующего типа (класса) являются частными случаями выражения;
 - `a + val` — добавление к массиву `a` элемента со значением `val` со стороны верхней границы; в результате выполнения этой операции количество элементов и значение верхней границы увеличиваются на единицу и, соответственно, добавленный элемент становится последним по счёту элементом массива;
 - `val + a` — добавление к массиву `a` элемента со значением `val` со стороны нижней границы; в результате выполнения этой операции количество элементов массива увеличивается на единицу, а значение нижней границы уменьшается на единицу и, соответственно, добавленный элемент становится первым по счёту элементом массива;
 - `a--` — удаление элемента массива `a` со стороны верхней границы (удаление последнего по счёту элемента массива); в качестве результата данная операция возвращает значение удаляемого элемента массива; при удалении количество элементов в массиве и значение верхней границы уменьшаются на единицу;
 - `--a` — удаление элемента массива `a` со стороны нижней границы (удаление первого по счёту элемента); в качестве результата данная операция возвращает значение удаляемого элемента массива; при удалении количество элементов в массиве уменьшается на единицу, а значение нижней границы увеличивается на единицу;
 - `a = b` — присваивание массиву `a` значения массива `b`; в результате выполнения данной операции массив `a` становится копией массива `b`; если перед выполнением операции присваивания в массиве `a` есть элементы, то они удаляются.
- Деструктор, обеспечивающий удаление элементов массива.

Инструмент: система программирования C++ [5] [3].

6.1 Требования

В процессе выполнения этапа должны быть созданы и предоставлены на проверку следующие файлы:

- `deque.hpp` — файл с описанием шаблонов классов `DequeElement` и `Deque`, включая описание реализации методов;
- `list.hpp` — файл с описанием шаблона класса `List`, включая описания реализации методов;
- `array.hpp` — файл с описанием шаблона класса `Array`, включая описания реализации методов;

- `test06.cpp` — файл, содержащий исходный текст на языке программирования C++ с реализацией функции `main()`, обеспечивающей тестирование работы с объектами разработанных классов и обработку исключительных ситуаций.

Возможно наличие и дополнительных файлов, если они потребуются, но пользователь будет оперировать только всеми или некоторыми из приведённых выше `hprp`-файлов.

Тестовая программа должна обеспечить создание и работу с двумя массивами с именами `parray01` и `parray02` соответственно, причём, массив `parray01` должен содержать элементы вещественного типа, а массив `parray02` должен содержать строки символов. Тестовая программа должна обеспечить добавление не менее трёх элементов со стороны нижней границы массива и не менее трёх элементов со стороны верхней границы массива для каждого из массивов, а также, получение и изменение значений некоторых из элементов, не являющихся крайними элементами массива. При каждом действии на экран должна выводиться следующая информация:

- выполняемое действие;
- результат выполнения действия;
- состояние массива после выполнения действия.

6.1.1 Пример вывода информации на экран

```
Array a
OK
++
!!
++
-

a.Low() == ArrayIsEmpty
ArrayIsEmpty
++
!!
++
-

a + 10.25
OK
+-----+
!    10.25!
+-----+
          1

20.25 + a
OK
+-----+-----+
!    20.25!    10.10!
+-----+-----+
          0          1

a[1] == 13.13
OK
+-----+-----+
!    20.25!    13.13!
+-----+-----+
          0          1
```

В тестовой программе общее количество операций — должно превышать общее количество операций +.

Тестовая программа должна быть реализована без запросов на ввод данных пользователем с экрана.

Для динамического выделения и удаления памяти использовать операции языка программирования C++ **new** и **delete** (см. [3]).

При выполнении этапа **ЗАПРЕЩАЕТСЯ** использовать классы стандартной библиотеки системы программирования C++.

Список литературы

- [1] Дональд Кнут, Искусство программирования для ЭВМ, том 1 “Основные алгоритмы”, М., “Мир”, 1976.
- [2] Барбара Лисков, Джон Гатэг, Использование абстракций и спецификаций при разработке программ, М., “Мир”, 1989.
- [3] Т.А. Павловская, C/C++. Программирование на языке высокого уровня, СПб., “Питер”, 2003.
- [4] Брайан Керниган, Денис Ритчи, Язык программирования C.
- [5] Бьерн Страуструп, Язык программирования C++.
- [6] Никлаус Вирт, Алгоритмы и структуры данных.