Упражнение 1. (1 балл) Реализуйте шаблонную версию класса Array. Список всех операций, которые должен поддерживать класс Array, приведен в шаблоне кода. Проверьте написанный код на типах int, double, string, char*, пользовательском типе Student (для студента храните имя и номер зачетки, а также определите необходимые методы — подумайте над конструкторами и деструкторами). Шаблон класса Array должен успешно работать с данными любого типа.

```
#include <cstddef>
using namespace std;

template <typename T>
class Array
```

```
{ private:
      T* myArray;
      size_t n;
public:
      // Список операций:
explicit Array(size_t size = 0, const T& value = T())
      // конструктор класса, который создает
      // Array размера size, заполненный значениями
      // value типа T. Считайте что у типа T есть
      // конструктор, который можно вызвать без
      // без параметров, либо он ему не нужен.
      {}
Array(const Array & mas)
      // конструктор копирования, который создает
      // копию параметра. Считайте, что для типа
      // T определен оператор присваивания.
      { }
~Array()
      // деструктор, если он вам необходим.
      {}
Array& operator=(const Array& mas)
      // оператор присваивания.
      {}
      //
      // две версии оператора доступа по индексу.
T& operator[](size_t idx)
      {}
const T& operator[](size_t idx) const
      {}
size_t size() const {}
};
```

Упражнение 2. (1 балл) В предыдущей версии предполагается, что для типа T определен оператор присваивания или он ему не нужен (например, для примитивных типов он не нужен). При создании шаблонных классов контейнеров (вроде Array и не только) разумно стараться минимизировать требования к типам шаблонных параметров. Поэтому усложним задачу, реализуйте класс Array не полагаясь на то, что для типа T определен оператор присваивания.

Подсказка: возможно, понадобится placement new и явный вызов деструктора, чтобы создавать и уничтожать объекты, аллоцировать правильно выровненную память можно с помощью `new char[N * sizeof(T)]`, где N – количество элементов массива.

```
#include <cstddef>

template <typename T>
class Array
{ private:
      T* myArray;
      size_t n;
public:
      // Список операций:
explicit Array(size_t size=0, const T& value=T())
      // конструктор класса, который создает
```

```
      // Array размера size, заполненный значениями
      // value типа T. Считайте что у типа T есть
      // конструктор, который можно вызвать без
      // без параметров, либо он ему не нужен.
      {}
Array(const Array & mas)
      // конструктор копирования, который создает
      // копию параметра. Считайте, что для типа
      // T определен оператор присваивания.
      { }
~Array()
      // деструктор, если он вам необходим.
      { }
Array& operator=(const Array& mas)
      // оператор присваивания.
      { }
      // две версии оператора доступа по индексу.
T& operator[](size_t idx)
      { }
const T& operator[](size_t idx) const
      { }
size_t size() const {}};
```

Упражнение 3. (1 балл) Шаблонные классы можно наследовать. Реализуйте шаблонную структуру ValueHolder с одним типовым параметром T, унаследованную от интерфейса ICloneable. Интерфейс ICloneable содержит всего один виртуальный метод ICloneable* clone() const, который должен вернуть указатель на копию объекта, на котором он был вызван (объект должен быть создан в куче). Шаблон ValueHolder, как говорит его название, хранит всего одно значение (назовите его data_) типа T (для типа T определен конструктор копирования). Не делайте поле data_ закрытым (поэтому в данном случае мы явно пишем, что ValueHolder должна быть структурой).

```
struct ICloneable
{
      virtual ICloneable* clone() const = 0;
      virtual ~ICloneable() { }
};

// Шаблон ValueHolder с типовым параметром T,
// должен содержать одно открытое поле data_
// типа T.

// В шаблоне ValueHolder должен быть определен
// конструктор от одного параметра типа T,
// который инициализирует поле data_.
//
// Шаблон ValueHolder должен реализовывать
// интерфейс ICloneable, и возвращать указатель
// на копию объекта созданную в куче из метода
// clone.
```

Упражнение 4. (1 балл) Реализуйте функцию копирования элементов copy_n из массива источника типа U* в целевой массив типа T*, где T и U произвольные типы, для которых определено преобразование из

U в T. На вход функция принимает два указателя и количество элементов, которые необходимо скопировать.

Пример вызова функции copy_n:

```
int ints[] = {1, 2, 3, 4};
double doubles[4] = {};
copy_n(doubles, ints, 4);
// теперь в массиве doubles содержатся
// элементы 1.0, 2.0, 3.0 и 4.0
```

```
#include <cstddef>
      // Параметры функции copy_n идут в следующем
      // порядке:
      // 1. целевой массив
      // 2. массив источник
      // 3. количество элементов, которые нужно
      // скопировать
      //
      // Вам нужно реализовать только функцию copy_n,
      // чтобы ее можно было вызвать так, как показано
      // в примере.
```

Упражение 5. (1 балл) Реализуйте шаблонную функцию minimum, которая находит минимальный элемент, который хранится в экземпляре шаблонного класса Array, при этом типовой параметр шаблона Array может быть произвольным. Чтобы сравнивать объекты произвольного типа, на вход функции также будет передаваться компаратор, в качестве компаратора может выступать функция или объект класса с перегруженным оператором «круглые скобки» "()".

Примеры вызова функции minimum:

```
bool less(int a, int b) { return a < b; }

struct Greater
{ bool operator()(int a, int b) { return b < a; } };

Array<int> ints(3);
ints[0] = 10;
ints[1] = 2;
ints[2] = 15;

int min = minimum(ints, less); // в min должно попасть 2
int max = minimum(ints, Greater()); // в max должно попасть 15
```

```
#include <cstddef>
template <typename T>
class Array
{
public:
      explicit Array(size_t size = 0, const T& value = T());
      Array(const Array& other);
      ~Array();
```

```
        Array& operator=(Array other);
        void swap(Array &other);
        size_t size() const;
        T& operator[](size_t idx);
        const T& operator[](size_t idx) const;
private:
        size_t size_;
        T *data_;
};
// Ваш код
```

Упражнение 6. (2 балла) Шаблонный класс Array может хранить объекты любого типа, для которого определён конструктор копирования, в том числе и другой Array, например, Array< Array<int> >. Глубина вложенности может быть произвольной. Напишите шаблонную функцию (или несколько) flatten, которая принимает на вход такой "многомерный" Array неизвестной заранее глубины вложенности и выводит в поток out через пробел все элементы, хранящиеся на самом нижнем уровне.

Примеры работы функции flatten:

```
Array<int> ints(2, 0);
ints[0] = 10;
ints[1] = 20;
flatten(ints, std::cout); // выводит на экран строку "10 20"
Array< Array<int> > array_of_ints(2, ints);
flatten(array_of_ints, std::cout);
// выводит на экран строку "10 20 10 20"
Array<double> doubles(10, 0.0);
flatten(doubles, std::cout);
// работать должно не только для типа int
```

Примечание: лидирующие и завершающие пробельные символы будут игнорироваться, т. е. там где ожидается "10 20" будет так же принят, например, вариант " 10 20 ", но не вывод "1020".

Подсказка: шаблонные функции тоже можно перегружать, из нескольких шаблонных функций будет выбрана наиболее специфичная.

```
#include <iostream>
// Весь вывод должен осущствляться в поток out,
// переданный в качестве параметра.
//
// Можно заводить любые вспомогаетльные функции,
// структуры или даже изменять сигнатуру flatten,
// но при этом все примеры вызова из задания должны
// компилироваться и работать.
```

Упражнение 7. (1 балл) Выше вы реализовали простой шаблон ValueHolder, в этом задании мы используем его чтобы написать класс Any (интересно, что не шаблонный), который позволяет хранить значения любого типа! Например, вы сможете создать массив объектов типа Any, и сохранять в них int-ы, double-ы или даже объекты Array. Подробности в шаблоне кода.

Подсказка: в нешаблонном классе Any могут быть шаблонные методы, например, шаблонный конструктор.

```
struct ICloneable;
// Поле data_ типа T в классе ValueHolder
// открыто, к нему можно обращаться

template <typename T>
struct ValueHolder;
// Это класс, который вам нужно реализовать

class Any
{
public:
// В классе Any должен быть конструктор, который можно вызвать
// без параметров, чтобы работал следующий код:
// Any empty; // empty ничего не хранит

// В классе Any должен быть шаблонный конструктор от одного
// параметра, чтобы можно было создавать объекты типа Any,
// например, следующим образом:
// Any i(10); // i хранит значение 10

// Деструктор: выделенные ресурсы нужно освободить.

// В классе Any также должен быть конструктор копирования (вам
// поможет метод clone интерфейса ICloneable)

// В классе должен быть оператор присваивания и/или шаблонный
// оператор присваивания, чтобы работал следующий код:
// Any copy(i); // copy хранит 10, как и i
// empty = copy; // empty хранит 10, как и copy
// empty = 0; // а теперь empty хранит 0

// Чтобы получать хранимое значение, определите в классе Any
// шаблонный метод cast, который возвращает указатель на
// хранимое значение, или нулевой указатель в случае
// несоответствия типов или если объект Any ничего не хранит:
// int *iptr = i.cast<int>(); // *iptr == 10
// char *cptr = i.cast<char>(); // cptr == 0,
// // потому что i хранит int, а не char
// Any empty2;
// int *p = empty2.cast<int>(); // p == 0
// При реализации используйте dynamic_cast.
};
```

Упражнение 8. (1 балл) В качестве упражнения на частичную специализацию шаблонов классов вам предлагается реализовать простой шаблон SameType. Этот шаблон не содержит никаких методов, а только одно **статическое константное поле типа bool** с именем value. Шаблон принимает два типовых параметра, и если два типовых параметра шаблона являются одним и тем же типом, то статическое поле value должно хранить значение true, в противном случае – значение false.

Примеры:

```
struct Dummy { };
typedef int type;
std::cout << SameType<int, int>::value << std::endl;
// выведет 1, т. е. true
std::cout << SameType<int, type>::value << std::endl;
// 1, type == int
std::cout << SameType<int, int&>::value << std::endl;
// 0, int и ссылка на int - различные типы
std::cout << SameType<Dummy, Dummy>::value << std::endl; // 1
std::cout << SameType<int, const int>::value << std::endl;
// 0, const - часть типа

// Определите шаблон SameType с двумя типовыми параметрами.
// В шаблоне должна быть определена одна статическая константа // типа bool с именем
value
```

Упражнение 9. (1 балл) Реализуйте функцию array_size, которая возвращает размер массива, переданного в качестве параметра. Функция должна работать только для массивов! Т. е. если функции передать указатель, должна произойти ошибка компиляции.

Примеры:

```
int ints[] = {1, 2, 3, 4};
int *iptr = ints;
double doubles[] = {3.14};
array_size(ints); // вернет 4
array_size(doubles); // вернет 1
array_size(iptr); // тут должна произойти ошибка компиляции
```

Подсказка: можно организовать передачу в функцию массивов только заданного размера (передача массива по ссылке), совместите его с вашими знаниями о шаблонах.