ЛАБОРАТОРНАЯ РАБОТА № 26
КЛАСС-МАССИВ

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Работа с массивами в языке **C++** требует особой тщательности, так как, компилятор не проверяет, вышли или нет индексы, которые используются при выборе элемента массива, из допустимых пределов. Следовательно, программист сам, дополнительными программными средствами, должен отслеживать правильность этих индексов. Есть и другие ограничения: индексация в массиве всегда начинается с 0, нельзя сравнить два массива с помощью операций проверки на равенство или операций отношения, при передаче массива функции в качестве аргумента размер массива должен передаваться как дополнительный аргумент, один массив не может быть присвоен другому с помощью операции присваивания. Это, казалось бы, ограничивает возможности **C++,** но механизмы классов и перегрузки операций позволяют достаточно просто и эффективно реализовывать эти и многие другие требования.

Пример разработки класса-массива переменных типа float

Разберем возможности **C++** на примере разработки класса массива вещественных чисел, в методах которого будет выполняться проверка диапазона, чтобы гарантировать, что индексы остаются в пределах границ массива.

Назовем наш класс **FArray** и сначала определим его члены-данные. Класс должен хранить массив переменных типа **float**, поэтому в закрытой части класса объявим указатель на будущий массив:

```
float* Arr;
```

Кроме того, нам необходимо знать размер массива, поэтому введем еще один член класса – **Len**. Так как размер массива не может быть отрицательным, объявим тип этой переменной **unsigned int**. Чтобы не писать каждый раз **unsigned int**, зададим новое имя с помощью оператора **typedef**:

```
typedef unsigned int UINT;
```

Тогда переменную, которая будет хранить размер массива, объявим так:

```
UINT Len;
```

В открытой части нашего класса объявим конструкторы и деструктор. Первый конструктор – конструктор без параметров, в котором будет создаваться пустой массив.

```
FArray();
```

Второй конструктор нужен, чтобы создавать массивы заданного размера, следовательно, он должен иметь один параметр типа **UINT**.

```
FArray(UINT k);
```

Деструктор необходим, так как память под массив будет выделяться динамически и ее нужно обязательно освобождать при уничтожении объекта.

```
~FArray();
```

Еще желательно создать конструктор копий, который необходим при использовании объектов данного класса в качестве аргументов функции или возвращаемого значения, но о нем поговорим позже.

В первом приближении объявление класса будет выглядеть так:

```
class FArray
{
private:
  float* Arr;   // Указатель на будущий массив
  UINT Len;     // Размер массива
public:
```

```
  FArray ();        // Конструктор по умолчанию (размер массива 0)
  FArray (UINT k); // Конструктор, для массива размера k
  ~FArray ();       // Деструктор
};
```

Чтобы создать объект нашего класса, например, в функции **main**, можно написать так:

```
int main()
{
. . . .
  FArray A1; // Объявление объекта-массива размером 0
  FArray A2(10); // Объявление объекта-массива размером 10
. . . .
}
```

Сами конструкторы и деструктор напишем позднее, сначала объявим другие функции, необходимые в классе. Во-первых, это функция, позволяющая задать размер массива и выделить под него память, а также функция, возвращающая размер массива.

```
void SetSize(UINT k); // Задание/изменение размера массива
UINT GetSize() {return Len;} // Размер массива
```

Чтобы обращаться к элементу массива традиционным образом с помощью операции индексирования `[]`, перегрузим в нашем классе соответствующую операцию. Применение операции `[]` должно выглядеть так:

```
f = A[i]; // Чтение из массива A значения элемента с индексом i
A[i] = 3.14; // Запись в i-тый элемент массива A значения 3.14
```

В первом случае операция `[]` возвращает ***i***-тый элемент, хранящийся в массиве, т.е. тип возвращаемого значения операции должен быть **float** (тип хранимых данных). Во втором случае мы пишем новое значение в ***i***-тый элемент массива, т.е. результат перегруженной операции `[]` будет стоять слева от знака присваивания. Чтобы записать новое значение, операция `[]` должна возвращать ссылку на ***i***-тый элемент, т.е. тип возвращаемого значения должен быть не **float**, а ссылкой на **float**.

Перегружаемая операция объявляется так:

```
float& operator[](UINT i); // Обращение к элементу массива
```

Объявление класса с учетом новых функций:

```
class FArray
{
private:
  float* Arr;  // Указатель на будущий массив
  UINT Len;    // Размер массива
public:
  FArray ();        // Конструктор по умолчанию (размер массива 0)
  FArray (UINT k); // Конструктор, для массива размера k
  ~FArray ();       // Деструктор

  void SetSize(UINT k); // Задание/изменение размера массива
  UINT GetSize() {return Len;} // Размер массива
  float& operator[](UINT i); // Обращение к элементу массива
};
```

Теперь напишем функции, объявленные в классе-массиве.

Конструктор по умолчанию не создает массив и не выделяет под него память, поэтому в нем нужно просто задать размер – **0** и указателю на будущий массив тоже присвоить значение **0**.

```
FArray::FArray()  // Конструктор по умолчанию (размер массива 0)
{
  Len = 0;  // Размер массива - 0
```

```
    Arr = 0;  // Указателю задан «безопасный» адрес - 0
  }
```

Во втором конструкторе нужно выделять память под массив заданного размера, поэтому можно воспользоваться функцией **SetSize**, которую мы уже объявили в классе и напишем сейчас.

Функция **SetSize** предназначена для того, чтобы задать или изменить размер массива, поэтому в ней сначала необходимо освободить память, если она уже выделялась, а затем уже выделять новую.

```
void FArray::SetSize(UINT k) // Задать/изменить размер массива
{
  if (Arr)          // Если память выделялась ранее,
  {
    delete[]Arr;        // освобождаем ее
    Arr = 0;
  }
  Len = k;               // Устанавливаем новый размер
  Arr = new float[Len]; // и выделяем новую память
}
```

Теперь конструктор с параметром выглядит совсем просто:

```
FArray::FArray(UINT k)
{
  Len = 0;  // Размер массива - 0
  Arr = 0;  // Указателю задан «безопасный» адрес - 0
  SetSize(k);  // А память выделяем с помощью функции SetSize
}
```

В деструкторе необходимо освободить память, но только если она выделялась. Для этого проверяем указатель Arr: если он равен **0**, то память не выделялась и ее не нужно освобождать.

```
FArray::~ FArray()
{
  if (Arr)          // Если память выделялась ранее,
    delete[]Arr;        // освобождаем ее
}
```

Перегружаем операцию индексирования, в которой сначала убеждаемся, что индекс не выходит за пределы массива, и только затем возвращаем ссылку на нужный элемент:

```
if (i < Len)       // Если индекс не вышел за пределы массива,
  return Arr[i];  // возвращаем ссылку на этот элемент,
```

Что делать, если индекс вышел за пределы массива, – решается по-разному в зависимости от требований к программе.

Один вариант – возвращать ссылку на первый или последний элемент массива:

```
float& FArray::operator[](UINT i) // Обращение к i-тому
{                                 // элементу массива
  if (i < Len)          // Если индекс не вышел за пределы массива,
    return Arr[i];      // возвращаем ссылку на этот элемент,
  return Arr[Len - 1]; // иначе - ссылку на последний элемент
}
```

Это не очень хорошо, т.к. в случае ошибочного индекса мы изменим не тот элемент массива. Еще один вариант – выделять заранее память на один элемент больше

```
  Arr = new float[Len + 1];
```

и возвращать ссылку на этот дополнительный элемент массива.

```
float& FArray::operator[](UINT i) // Обращение к i-тому
{                                 // элементу массива
  if (i < Len)          // Если индекс не вышел за пределы массива,
```

```
    return Arr[i];      // возвращаем ссылку на этот элемент,
  return Arr[Len];      // иначе - ссылку на дополнительный элемент
}
```

Это тоже не совсем правильно, т.к. в случае ошибочного индекса мы никак не узнаем про это. Лучший вариант – вызывать так называемую исключительную ситуацию, которую затем при использовании массива можно будет перехватывать и правильно обрабатывать.

```
float& FArray::operator[](UINT i) // Обращение к i-тому
{                                 // элементу массива
  if (i < Len)          // Если индекс не вышел за пределы массива,
    return Arr[i];      // возвращаем ссылку на этот элемент,
  throw 5;    // иначе - вызываем исключительную ситуацию
}
```

Теперь наш класс-массив будет выглядеть так:

```
//---------------------------------------
typedef unsigned int UINT;
//---------------------------------------
class FArray
{
private:
  float* Arr;  // Указатель на будущий массив
  UINT Len;    // Размер массива
public:
  FArray () {Len = 0; Arr = 0;}
  FArray (UINT k) {Len = 0; Arr = 0; SetSize(k);}
  ~FArray () {if (Arr) delete[]Arr;}

  void SetSize(UINT k); // Задание/изменение размера массива
  UINT GetSize() {return Len;} // Размер массива
  float& operator[](UINT i); // Обращение к элементу массива
};
//---------------------------------------
void FArray::SetSize(UINT k) // Задать/изменить размер массива
{
  if (Arr)        // Если память выделялась ранее,
  {
    delete[]Arr;       // освобождаем ее
    Arr = 0;
  }
  Len = k;              // Устанавливаем новый размер
  Arr = new float[Len]; // и выделяем новую память
}
//---------------------------------------
float& FArray::operator[](UINT i) // Обращение к i-тому
{                                 // элементу массива
  if (i < Len)          // Если индекс не вышел за пределы массива,
    return Arr[i];      // возвращаем ссылку на этот элемент,
  throw 5;    // иначе - вызываем исключительную ситуацию
}
```

Пример использования класса-массива:

```
int main()
{
. . . .
  FArray A(30); // Объявление объекта-массива размером 30
```

```
    for(UINT i = 0; i < A.GetSize();i++)
      A[i] = i + 1;  // Заполнение массива значениями
  . . . .
    for(UINT i = 0; i < A.GetSize();i++) // Вывод на дисплей
      cout << " A[" << i << "] = "A[i] << endl;
  . . . .
    A.SetSize(25); // Изменение размера массива  . . . .
    for(UINT i = 0; i < A.GetSize();i++) // Вывод на дисплей
      cout << " A[" << i << "] = "A[i] << endl;
  }
```

Пример разработки класса-массива объектов типа CTime

Аналогично можно создать класс-массив не только переменных стандартных типов, но и объектов классов. Рассмотрим процесс создания класса-массива на следующем примере:

Задание:

Разработать класс **Массив объектов класса CTime** и программу, иллюстрирующую возможности данного класса.

Класс должен хранить массив объектов класса **CTime**. Класс должен включать методы позволяющие:

- задавать (изменять) размер массива;
- обращаться к элементам массива (перегрузить **операцию []**);

Все проверки корректности ввода должны проводиться в методах класса.

Нам необходимо создать массив объектов класса **CTime** (класс **CTime** был разработан в работе 23). Процесс создания этого класса-массива похож на то, что мы уже проделали выше в классе **FArray**. Отличие заключается в том, что вместо типа **float** мы везде подставляем **CTime**.

Назовем наш класс **TimesArray** и сначала определим его члены-данные. Класс должен хранить массив времен, поэтому в закрытой части класса объявим указатель на будущий массив:

```
CTime* Arr;
```

Кроме того, нам необходимо знать размер массива, поэтому введем еще один член класса – **Len**. Так как размер массива не может быть отрицательным, объявим тип этой переменной **unsigned int**. Чтобы не писать каждый раз **unsigned int**, зададим новое имя с помощью оператора **typedef**:

```
typedef unsigned int UINT;
```

Тогда переменную, которая будет хранить размер массива, объявим так:

```
UINT Len;
```

В открытой части объявим конструкторы и деструктор. Кроме конструктора без параметров, в котором будет создаваться пустой массив, добавим конструктор с параметром, чтобы создавать массивы заданного размера. Деструктор необходим, так как память под массив будет выделяться динамически и ее нужно освобождать при уничтожении объекта.

```
class TimesArray
{
private:
  CTime* Arr;
  UINT Len;
public:
  TimesArray();  // Конструктор по умолчанию (размер массива 0)
  TimesArray(UINT k); // Конструктор, для массива размера k
  ~TimesArray();      // Деструктор
};
```

Сами конструкторы и деструктор напишем позднее, сначала объявим другие функции, необходимые в классе. Во-первых, это функция, позволяющая задать размер массива и выделить под него память, а также функция, возвращающая размер массива.

```
void SetSize(UINT k); // Задание/изменение размера массива
UINT GetSize() {return Len;} // Размер массива
```

Чтобы обращаться к элементу массива традиционным образом с помощью операции индексирования `[]`, перегрузим соответствующую операцию. У нас обычный массив, поэтому индекс должен быть целого типа, а применение операции должно выглядеть так:

```
TimesArray A1(10);
CTime T1 = A1[j];
A1[i] = CTime("12:00:00");
```

где **A1** – это объект нашего класса-массива. В первом случае операция `[]` возвращает *i*-тый элемент, хранящийся в массиве, т.е. тип возвращаемого значения операции должен быть **CTime** (тип хранимых данных). Но чтобы использовать операцию `[]` слева от знака присваивания, возвращать значение нужно по ссылке, т.е. тип возвращаемого значения должен быть ссылкой на **CTime**.

```
CTime& operator[](UINT i); // Обращение к элементу массива
```

Окончательно получаем следующее объявление класса-массива:

```
class TimesArray
{
private:
  CTime* Arr; // Указатель на область памяти для массива
  UINT Len;   // Размер массива
public:
  TimesArray();   // Конструктор по умолчанию (размер массива 0)
  TimesArray(UINT k); // Конструктор, для массива размера k
  ~TimesArray();      // Деструктор

  void SetSize(UINT k); // Задание/изменение размера массива
  unsigned int GetSize() {return Len;} // Размер массива
  CTime& operator[](UINT i);// Обращение к i-тому элементу массива
};
```

В деструкторе необходимо освободить память, если она выделялась.

```
TimesArray::~TimesArray()
{
  if (Arr)
    delete[] Arr;
}
```

Чтобы задать или изменить размер массива, сначала необходимо освободить память, если она уже выделялась, а затем выделить новую.

```
void TimesArray::SetSize(UINT k) // Задать/изменить размер массива
{
  if (Arr)        // Если память выделялась ранее,
  {
    delete[]Arr;       // освобождаем ее
    Arr = 0;
  }
  Len = k;              // Устанавливаем новый размер
  Arr = new CTime[Len]; // и выделяем память
}
```

Перегружаем операцию индексирования, в которой сначала убеждаемся, что индекс не выходит за пределы массива, и только затем возвращаем ссылку на нужный

элемент. Если индекс вышел за пределы массива, будем вызывать исключительную ситуацию, которую затем при использовании массива можно будет перехватывать и правильно обрабатывать.

```
CTime& TimesArray::operator[](UINT i) // Обращение к i-тому
{                                     // элементу массива
  if (i < Len)          // Если индекс не вышел за пределы массива,
    return Arr[i];      // возвращаем ссылку на этот элемент,
  throw 5;    // иначе - вызываем исключительную ситуацию
}
```

Окончательно получаем такой класс-массив:

```
class TimesArray
{
private:
  CTime* Arr;         // Указатель на область памяти для массива
  unsigned int Len;   // Размер массива
public:
  TimesArray() {Arr = 0; Len = 0;} // Конструктор по умолч.
  TimesArray(UINT k) {Arr = 0; Len = 0; SetSize(k);}
  ~TimesArray() {if (Arr) delete[] Arr;} // Деструктор

  void SetSize(UINT k); // Задание/изменение размера массива
  unsigned int GetSize() {return Len;} // Размер массива
  CTime& operator[](UINT i); // Обращение к элементу массива
};
```

Несомненным достоинством этого класса является то, что объекты, созданные на его основе, сами заботятся о выделении и освобождении памяти, следят за правильностью индекса, при обращении к его элементам, и знают свой размер.

Это значит, что можно создать функцию, которая выводит информацию о содержимом массива на дисплей, передавая в нее в качестве аргумента только сам массив (точнее ссылку на него, чтобы при вызове функции не создавался новый объект и не вызывался конструктор копий).

```
void ViewArray(TimesArray& Ta)
{
  for(i = 0; i < Ta.GetSize(); i++)
    cout << "A[" << i << "] = " << Ta[i] << endl;
}
```

На дисплей будет выведены строки с информацией о времени

A[0] = 01:02:03

а число строк будет равно числу элементов массива. Информация о времени будет выводиться правильно, если мы перегрузили **операцию <<** для класса **CTime**.

Чтобы иметь возможность выводить в строке не только **A[0]**, но и другое имя массива, нужно передать в эту функцию соответствующую строку и тоже выводить ее в поток вывода.

```
void ViewArray(const char * str, TimesArray& Ta)
{
  for(i = 0; i < Ta.GetSize(); i++)
    cout << str << "[" << i << "] = " << Ta[i] << endl;
}
```

Тогда при вызове функции, например, так:

```
ViewArray("A", A);
```

будет выводиться

A[0] = 01:02:03

а при таком вызове:

```
ViewArray("Элемент", A);
```

будет выводиться

Элемент[0] = 01:02:03

Для тестирования класса создадим два объекта типа **TimesArray**, затем проверим методы, созданные для класса:

- изменить размеры массива и задать значения,
- вывести все элементы массивов,
- выбрать элемент массива по индексу, задать/изменить/отобразить его свойства,

```
int main()
{
  TimesArray A, B(5);
  UINT i;
  setlocale(0, "Russian");

  A.SetSize(10);  // Изменили размер массива A
  for(i = 0; i < A.GetSize(); i++)  // Задаем значения элементов
    A[i].SetTime(i + 1, i*2 + 2, i*3 + 3); // массива с шагом
  cout << "Задали свойства массива A с шагом\n";
  ViewArray("A", A); // Выводим значения элементов массива

  B.SetSize(8);  // Изменили размер массива B
  for(i = 0; i < B.GetSize(); i++)  // Задаем значения элементов
    B[i].SetTime(i + 12, i*2 + 20, i*3 + 30); // массива с шагом
  cout << "Задали свойства массива B с шагом\n";
  ViewArray("B", B); // Выводим значения элементов массива

  // Выбираем элемент массива (задаем его индекс) (напр., 3)
  cout << "Задать индекс: ";
  cin >> i;
     // вызываем для него разные методы .......
  A[i].SetTime("22:15:15");
        // и выводим на дисплей
  cout << "Изменили A[" << i << "] = " << A[i] << endl;
  cout << "Измененный массив A\n";
  ViewArray("A", A);

  cin.get();
  return 0;
}
```

```
A[0] = 01:02:03
A[1] = 02:04:06
A[2] = 03:06:09
A[3] = 04:08:12
A[4] = 05:10:15
A[5] = 06:12:18
A[6] = 07:14:21
A[7] = 08:16:24
A[8] = 09:18:27
A[9] = 10:20:30
```

```
Задать индекс: 3
Изменили A[3] = 22:15:15
```

ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ

в начале программы ОБЯЗАТЕЛЬНО выводить:
ФИО, группа, номер лаб. работы, номер варианта.

Задание

Разработать класс **Массив объектов**, и программу, демонстрирующую возможности данного класса.

Методы доступа к элементам класса должны включать проверку выхода за пределы массива. Размер массива может быть задан любым.

Класс должен включать методы позволяющие:

- задавать (изменять) размер массива;
- обращаться к элементам массива (перегрузить **операцию []**);

Все проверки корректности ввода должны проводиться в методах класса.

В программе необходимо создать объект данного класса и проиллюстрировать его возможности:

- задать (изменить) размер объекта-массива;
- *задать свойства элементов объекта-массива случайными величинами; **
- *задать свойства элементов объекта-массива с некоторым шагом; **
- вывести на дисплей основные свойства всех элементов объекта-массива в виде таблицы;
- просмотреть все свойства любого элемента объекта-массива, выбрав его по номеру (индексу);
- задать (изменить) свойства любого элемента объекта-массива, выбрав его по номеру (индексу);

*Для получения максимального балла нужно выполнить требования, выделенные курсивом и отмеченные *.*

Варианты заданий

Вариант 1. Массив Четырехугольников

Разработать класс **Четырехугольник** и класс **Массив Четырехугольников**, а также программу, демонстрирующую возможности данных классов.

Класс **Четырехугольник** должен хранить координаты его вершин (не более 100 пикселов по каждой координате) и включать:

– методы, позволяющие задавать и читать координаты вершины (с индексом i);

– *методы, позволяющие вычислять длины сторон и периметр четырехугольника. **

Требования к классу **Массив** перечислены выше.

Вариант 2. Массив Трапеций

Разработать класс **Трапеция** и класс **Массив Трапеций**, а также программу, демонстрирующую возможности данных классов.

Класс **Трапеция** должен хранить координаты (не более 100 пикселов по каждой координате) и размеры трапеции и включать:

– методы, позволяющие задавать и читать координаты и размеры трапеции;

– *методы, позволяющие вычислять площадь трапеции, длины сторон, периметр. **

Требования к классу **Массив** перечислены выше.

Вариант 3. Массив Ромбов

Разработать класс **Ромб** и класс **Массив Ромбов**, а также программу, демонстрирующую возможности данных классов.

Класс **Ромб** должен хранить координаты (не более 100 пикселов по каждой координате) и размеры ромба и включать:

методы, позволяющие задавать и читать координаты и размеры ромба;

*методы, позволяющие вычислять площадь ромба, длины сторон, периметр. **

Требования к классу **Массив** перечислены выше.

Вариант 4. Массив Параллелограммов

Разработать класс **Параллелограмм** и класс **Массив Параллелограммов**, а также программу, демонстрирующую возможности данных классов.

Класс **Параллелограмм** должен хранить координаты (не более 200 пикселов по каждой координате) и размеры параллелограмма и включать:

методы, позволяющие задавать и читать координаты и размеры параллелограмма;

*методы, позволяющие вычислять площадь параллелограмма, длины сторон, периметр. **

Требования к классу **Массив** перечислены выше.

Вариант 5. Массив Конусов

Разработать класс **Конус** и класс **Массив Конусов**, а также программу, демонстрирующую возможности данных классов.

Класс **Конус** должен хранить радиус основания (не более 100 пикселов) и высоту конуса (не более 200 пикселов) и включать:

методы, позволяющие задавать и читать радиус основания и высоту;

*методы, позволяющие вычислять объем конуса, площадь основания, боковой стороны и общую площадь. **

Требования к классу **Массив** перечислены выше.

Вариант 6. Массив Резистор

Разработать класс **Резистор** и класс **Массив Резисторов**, а также программу, демонстрирующую возможности данных классов.

Класс **Резистор** должен хранить параметры резистора: сопротивление, макс. рассеиваемую мощность (не более 10 Вт) и включать:

методы, позволяющие задавать и читать сопротивление, макс. рассеиваемую мощность;

*методы, позволяющие вычислять ток и рассеиваемую мощность при рабочем (реально приложенном) напряжении, при превышении макс. мощности на 10% резистор перегорает. **

Требования к классу **Массив** перечислены выше.

Вариант 7. Массив Равносторонних пятиугольников

Разработать класс **Равносторонний пятиугольник** и класс **Массив Равносторонних пятиугольников**, а также программу, демонстрирующую возможности данных классов.

Класс **Равносторонний пятиугольник** должен хранить координаты (не более 100 пикселов по каждой координате), длину стороны пятиугольника и включать:

методы, позволяющие задавать и читать размеры пятиугольникам;

*методы, позволяющие вычислять площадь пятиугольника, длины сторон, периметр. **

Требования к классу **Массив** перечислены выше.

Вариант 8. Массив Трехмерных векторов

Разработать класс **Трехмерный вектор** и класс **Массив Трехмерных векторов**, а также программу, демонстрирующую возможности данных классов.

Класс **Трехмерный вектор** должен хранить координаты (не более 100 пикселов по каждой координате) и включать:

– задавать и читать отдельные компоненты вектора (как числа типа float);

*– задавать и читать компоненты вектора строкой в формате "**13; -23.5; 0.8**";* *

– вычислять модуль.

Требования к классу **Массив** перечислены выше.

Вариант 9. Лампочка

Разработать класс **Лампочка** и класс **Массив Лампочек**, а также программу, демонстрирующую возможности данных классов.

Класс **Лампочка** должен хранить параметры лампочки: номинальную мощность (не более 200 Вт), номинальное напряжение (не более 300 В) и включать:

методы, позволяющие задавать и читать номинальную мощность, номинальное напряжение;

методы, позволяющие вычислять ток и рассеиваемую мощность при рабочем (реально приложенном) напряжении, при превышении номинальной мощности на 10% лампочка перегорает.

Требования к классу **Массив** перечислены выше.

Вариант 10. Комплексное число

Разработать класс **Комплексное число** и класс **Массив Комплексных чисел**, а также программу, демонстрирующую возможности данных классов.

Класс **Комплексное число** должен хранить реальную и мнимую части числа (тип float) и включать методы, позволяющие:

*– задавать и считывать комплексное число строкой в формате "**1.3 + i * 12.6**";* *

– задавать и считывать отдельно реальную и мнимую части;

– вычислять модуль и фазу.

Требования к классу **Массив** перечислены выше.

Вариант 11. Массив Аквариумов (Параллелепипед с жидкостью)

Разработать класс **Аквариум** и класс **Массив Аквариумов**, а также программу, демонстрирующую возможности данных классов.

Класс **Аквариум** должен хранить геометрические размеры аквариума (не более 100 пикселов по каждой координате), уровень жидкости, налитый в него, и включать:

методы, позволяющие задавать и читать размеры аквариума и уровень налитой жидкости;

методы, позволяющие вычислять объем аквариума, объем налитой жидкости, процент заполненности аквариума. *

Требования к классу **Массив** перечислены выше.

Вариант 12. Массив Колес со спицами

Разработать класс **Колесо со спицами** и класс **Массив Колес**, а также программу, демонстрирующую возможности данных классов.

Класс **Колесо со спицами** должен хранить радиус колеса (не более 100), толщину обода (не более половины радиуса), количество спиц (от 8 до 20) и включать:

методы, позволяющие задавать и читать параметры колеса;

методы, позволяющие вычислять общую площадь колеса, площадь обода, длину окружности колеса.

Требования к классу **Массив** перечислены выше.

Вариант 13. Массив Равносторонних шестиугольников

Разработать класс **Равносторонний шестиугольник** и класс **Массив Равносторонних шестиугольников**, а также программу, демонстрирующую возможности данных классов.

Класс **Равносторонний шестиугольник** должен хранить координаты (не более 200 пикселов по каждой координате), длину стороны шестиугольника и включать:

методы, позволяющие задавать и читать размеры шестиугольникам;

методы, позволяющие вычислять площадь шестиугольника, длины сторон, периметр.

Требования к классу **Массив** перечислены выше.

Вариант 14. Массив Цилиндров

Разработать класс **Цилиндр** и класс **Массив Цилиндров**, а также программу, демонстрирующую возможности данных классов.

Класс **Цилиндр** должен хранить радиус основания (не более 100 пикселов) и высоту цилиндра (не более 200 пикселов) и включать:

методы, позволяющие задавать и читать радиус основания и высоту;

методы, позволяющие вычислять объем цилиндра, площадь основания, боковой стороны и общую площадь.

Требования к классу **Массив** перечислены выше.

Вариант 15. Массив Обыкновенных дробей

Разработать класс **Обыкновенная дробь** и класс **Массив Обыкновенных дробей**, а также программу, демонстрирующую возможности данных классов.

Класс **Обыкновенная дробь** должен хранить величину числителя и знаменателя и включать:

методы, позволяющие задавать и читать значение числителя и знаменателя (знаменатель не должен быть равным 0);

методы, позволяющие считывать дробь десятичным числом;

Все проверки корректности ввода должны проводиться в методах класса.

Требования к классу **Массив** перечислены выше.

Вариант 16. Бочка (Цилиндр с жидкостью)

Разработать класс **Бочка** и класс **Массив Бочек**, а также программу, демонстрирующую возможности данных классов.

Класс **Бочка** должен хранить геометрические размеры бочки (не более 100 пикселов по каждой координате), уровень жидкости, налитый в нее, и включать:

методы, позволяющие задавать и читать размеры бочки и уровень налитой жидкости;

методы, позволяющие вычислять объем бочки, объем налитой жидкости, процент заполненности бочки.

Требования к классу **Массив** перечислены выше.

Вариант 17. Массив Логических элементов ИЛИ

Разработать класс **Логический элемент ИЛИ** и класс **Массив Логических элементов ИЛИ**, а также программу, демонстрирующую возможности данных классов.

Класс **Логический элемент ИЛИ** должен хранить количество входов логического элемента (2, 3, 4 или 8 входов), значения входных сигналов (0 или 1) и включать:

– методы, позволяющие задавать и читать количество входов логического элемента;

– методы, позволяющие задавать и читать значение входного сигнала на i-том входе (0 или 1);

– *методы, позволяющие вычислять сигнал на выходе логического элемента в зависимости от сигналов на входах. **

Требования к классу **Массив** перечислены выше.

Вариант 18. Массив Логических элементов И

Разработать класс **Логический элемент И** и класс **Массив Логических элементов И**, а также программу, демонстрирующую возможности данных классов.

Класс **Логический элемент И** должен хранить количество входов логического элемента (2, 3, 4 или 8 входов), значения входных сигналов (0 или 1) и включать:

– методы, позволяющие задавать и читать количество входов логического элемента;

– методы, позволяющие задавать и читать значение входного сигнала на i-том входе (0 или 1);

– *методы, позволяющие вычислять сигнал на выходе логического элемента в зависимости от сигналов на входах. **

Требования к классу **Массив** перечислены выше.

Вариант 19. Треугольник

Разработать класс **Треугольник** и класс **Массив Треугольников**, а также программу, демонстрирующую возможности данных классов.

Класс **Треугольник** должен хранить координаты вершин треугольника (не более 200 пикселов по каждой координате) и включать:

методы, позволяющие задавать и читать координаты вершины (с индексом i);

*методы, позволяющие вычислять длины сторон треугольника и периметр.**

Требования к классу **Массив** перечислены выше.

Вариант 20. Пирамида

Разработать класс **Пирамида** и класс **Массив Пирамид**, а также программу, демонстрирующую возможности данных классов.

Класс **Пирамида** должен хранить размеры основания (не более 200 пикселов) и высоту пирамиды (не более 200 пикселов) и включать:

методы, позволяющие задавать и читать размеры основания и высоту;

*методы, позволяющие вычислять объем пирамиды, площадь основания, боковой стороны и общую площадь. **

Требования к классу **Массив** перечислены выше.