

## ЛАБОРАТОРНАЯ РАБОТА № 27 ДОРАБОТКА КЛАССА-МАССИВА

### ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

При работе с массивами в языке **C++** есть масса ограничений: 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. Доработаем классы-массивы, разработанные в предыдущей лабораторной работе так, чтобы было можно:

- задавать (изменять) размер массива, **с сохранением, по возможности, прежних данных массива**;
- обращаться к элементам массива (перегрузить операцию **[]**) и обрабатывать исключительные ситуации;
- сравнивать два массива на полное равенство (перегрузить операцию **==**);
- копировать один массив в другой (перегрузить операцию **=**);
- копировать часть одного массива в другой;
- сортировать массив.

В предыдущей лабораторной работе был разработан класс

```
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
};
```

И его методы:

```
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
}
```

Рассмотрим его недостатки и постараемся их исправить.

## Изменение размера массива, с сохранением, по возможности, прежних данных массива

При изменении размера массива старые данные, которые в нем хранились, безвозвратно теряются, а желательно было бы сделать так, чтобы информация, по возможности, сохранялась. Например, если размер объекта-массива, созданного на базе нашего класса, был равен 5:

```
FArray A(5);
```

и хранил значения **{5, 10, 15, 20, 25}**, заданные так:

```
for(UINT i = 0; i < A.GetSize(); i++)
    A[i] = (i+1) * 5;
```

то при его увеличении

```
A.SetSize(10);
```

первые пять элементов массива остались такими же. При уменьшении размера массива желательно, чтобы те элементы, что вошли в новый массив, тоже сохранили бы свои значения.

Для этого нужно переделать метод **SetSize** класса так, чтобы при изменении размера массива сохранять, по возможности, прежние данные, которые до этого хранились в массиве. Для этого сначала не освобождаем память, а выделяем новую, используя для нее временный указатель. Затем копируем элементы из прежней памяти в новую, и только после этого освобождаем ранее выделявшуюся память. Адрес выделенной памяти запоминаем в указателе **Arr**.

```
void FArray::SetSize(UINT k) // Задать/изменить размер массива
{
    float* tmp; // Временный указатель для нового массива
    tmp = new float[k]; // Выделяем память для нового массива
    if (Arr) // Если память выделялась ранее
    {
        for (UINT i = 0; i < k && i < Len; i++)
            tmp[i] = Arr[i]; // Копируем элементы из старого в новый
        delete[] Arr; // и освобождаем память
    }
    Len = k; // Запоминаем новый размер
    Arr = tmp; // запоминаем в указателе адрес вновь выделенной
               // памяти
}
```

## Переделка пределов индексации

Если нам не нравится то, что индексация в массиве начинается с **0**, то при перегрузке операции **[]** это легко изменить. Если при обращении к элементу будем писать не **Arr[i]**, а **Arr[i-1]**, то индексация будет в пределах от **1** до **Len**.

```
float& FArray::operator[] (UINT i) // i лежит от 1 до Len включительно
{
    // Обращение к i-тому элементу массива
    if (i > 0 && i < Len) // Если индекс не вышел за пределы массива,
        return Arr[i - 1]; // возвращаем ссылку на этот элемент,
    throw 5; // иначе - вызываем исключительную ситуацию
}
```

## Обработка исключительных ситуаций

В перегруженной операции `[]` мы вызывали исключительную ситуацию. Рассмотрим, что это такое и как ее можно обрабатывать в случае ошибочных индексов.

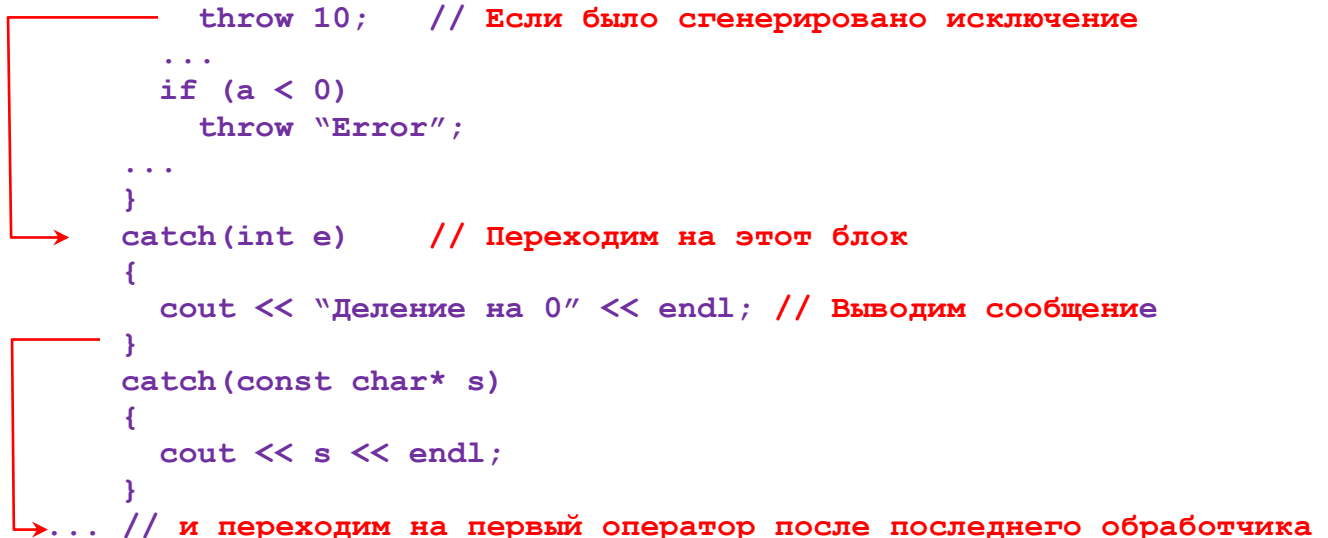
Обработка исключений в C++ используется тогда, когда функция обнаруживает ошибку, но не способна сама с ней управиться. Такая функция **генерирует исключение** (или, как иногда говорят, возбуждает или выбрасывает исключение). Чтобы его перехватывать, нужно заключить в так называемый **блок try** (блок испытания или охранный блок) тот код, который может сгенерировать ошибку, возбуждающую исключение.

Следом за блоком **try** записывается **один или более блоков catch** (блоки перехвата). Каждый блок **catch** определяет тип исключений, которые он может перехватывать и обрабатывать. Каждый блок **catch** содержит программу обработки — обработчик исключения.

```
try          // охранный блок
{
    . . . операторы
    throw ... // генерация исключения
    . . . операторы
}
catch(тип параметр) // блок перехвата
{
    . . .
}
catch(тип параметр) // блок перехвата
{
    . . .
}
. . .
```

Если исключительная ситуация **соответствует типу параметра** в одном из блоков **catch**, то выполняется код этого блока. Если при выполнении блока **try** не генерируется ни одно исключение, все обработчики исключений **пропускаются** и управление передается первому оператору после последнего обработчика.

```
...
try
{
    ...
    if (a == 0)
        throw 10; // Если было сгенерировано исключение
    ...
    if (a < 0)
        throw "Error";
    ...
}
catch(int e) // Переходим на этот блок
{
    cout << "Деление на 0" << endl; // Выводим сообщение
}
catch(const char* s)
{
    cout << s << endl;
}
... // и переходим на первый оператор после последнего обработчика
```



Ключевое слово **throw** используется для того, чтобы указать, какое исключение генерируется. Это называется генерацией исключения или возбуждением исключения. Обычно **throw** имеет один операнд. Операнд **throw** может быть любого типа. В нашем классе мы сгенерировали исключительную ситуацию типа **int** :

```
throw 5; // вызываем исключительную ситуацию типа int
```

Теперь можно ее перехватывать (переменной **e** будет присвоено значение 5):

```
catch(int e)
{
    . . .
}
```

и обрабатывать, например, вывести соответствующее сообщение.

```
catch (int e)
{
    if (e == 5)
        cout << "Неверный индекс!" << endl;
}
```

Если будет необходимость, можно и в других методах и функциях генерировать исключительные ситуации и задавать им просто новые значения целого типа. Они тоже будут перехватываться в этом же блоке **catch**..

Можно было генерировать исключение другого типа, например, **const char \***:

```
throw "Ошибка"; // вызываем исключит. ситуацию типа const char*
```

Тогда блок перехвата, соответствующий ему, тоже должен быть типа **const char \***.

```
catch(const char* s)
{
    cout << s << endl;
}
```

Когда исключение сгенерировано, программное управление покидает текущий блок **try** и передается соответствующему обработчику **catch** (если он существует), расположенному после данного блока **try**.

Исключение должно генерироваться только внутри блока **try**. Исключение, сгенерированное вне блока **try**, вызывает обращение к функции **terminate** – прерыванию программы.

Обработчики исключений содержатся в блоках **catch**. Каждый блок **catch** начинается с ключевого слова **catch**, за которым следуют круглые скобки, содержащие тип и необязательное имя параметра. Затем в фигурных скобках записываются операторы обработки исключения. Когда исключение перехвачено, начинает выполняться программа, заключенная в блоке **catch**.

Исключение перехватывается первым обработчиком **catch**, следующим за блоком **try** и соответствующим типу сгенерированного объекта. Если после **catch** в круглых скобках записано многоточие:

```
catch (...)
```

это означает, что будут перехватываться все исключения.

Пример использования блоков **try – catch**.

При установке блока **try** нужно помнить: если произошло исключение, то операторы, которые стоят после той строки, где произошло исключение, не будут выполнены. Например, нам нужно выбрать элемент массива и умножить его значение на **100**. Закладываем в блок **try** те строки кода, где вводится индекс и вызывается операция **[]**.

```
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;
    . . . .
try
{
    // Выбираем элемент массива
    cout << "Задать индекс: ";
    cin >> i;
    A[i] *= 100; // в операции [] может быть исключение типа int
}
catch (int e)
{
    if (e == 5)
        cout << "Неверный индекс!" << endl;
    else
        cout << "Неизвестная ошибка!" << endl;
}
for(UINT i = 0; i < A.GetSize();i++) // Вывод на дисплей
    cout << " A[" << i << "] = "A[i] << endl;
return 0;
}

```

Если индекс неверный, в перегруженной операции будет сгенерирована исключительная ситуация (**throw 5**), и программа перейдет на блок перехвата, где на дисплей будет выведено сообщение **“Неверный индекс!”**. Программа продолжит выполняться дальше и **не вернется** снова к запросу индекса. Чтобы запросить индекс снова, нужно заключить эти операторы в цикл, который выполнялся бы, пока не будет введен корректный индекс, например, так:

```

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;
        . . . .
    for(;;) // Бесконечный цикл
    {
        try
        {
            // Выбираем элемент массива
            cout << "Задать индекс: ";
            cin >> i;
            A[i] *= 100;
            break; // Выход из бесконечного цикла
        }
        catch (int e)
        {
            if (e == 5)
                cout << "Неверный индекс!" << endl;
            else
                cout << "Неизвестная ошибка!" << endl;
        }
    }
}

```

```

for(UINT i = 0; i < A.GetSize();i++) // Вывод на дисплей
    cout << " A[" << i << "] = "A[i] << endl;
    . . . .
return 0;
}

```

Если используется меню, то охранный блок лучше поставить так, чтобы он включал все меню, в том числе и переключатель **switch**:

```

int main()
{
    FArray A(30); // Объявление объекта-массива размером 30    char
    int w;
    . . . .
    for (;;) // Бесконечный цикл
    {
        try // охранный блок
        {
            cout << "\n\nМеню:\n";
            cout << "0 - выход из программы\n";
            cout << "1 - заполнить случайными числами: \n";
            cout << "2 - заполнить по возрастанию: \n";
            cout << "3 - просмотр: \n";
            cout << "4 - сравнить: \n";
            cout << "5 - умножить на 100 \n";
            . . . . . и т.п.
            cin >> w;
            switch (w)
            {
                case 0: // выход
                    return 0;
                case 1: // случайными числами
                    . . . . .
                    break;
                case 2: // по возрастанию
                    . . . . .
                    break;
                case 3: // просмотр
                    for(UINT i = 0; i < A.GetSize();i++) // Вывод на дисплей
                        cout << " A[" << i << "] = "A[i] << endl;
                    break;
                case 4: // сравнить
                    . . . . .
                    break;
                case 5: // Выбираем элемент массива
                    cout << "Задать индекс: ";
                    cin >> i;
                    A[i] *= 100; // Здесь может быть сгенерировано исключение
                    break;
                . . . . . и т.д.
            }
        } // Конец охранного блока
        catch (int e) // Блок перехвата
        {
            if (e == 5)
                cout << "Неверный индекс!" << endl;
            else

```

```

        cout << "Неизвестная ошибка!" << endl;
    } // Конец блока перехвата
} // Конец бесконечного цикла
return 0;
}

```

В данном примере при неверном индексе программа перейдет в блок перехвата, выведет соответствующее сообщение и снова предложит выбрать пункт меню.

### Перегрузка операции присваивания

Если в программе созданы два (или больше) объекта-массива, то может понадобиться копировать один объект-массив в другой, например, так:

```
A = B;
```

Для этого необходимо перегрузить операцию присваивания. Операцию присваивания можно перегружать **только как метод класса**, следовательно она должна быть объявлена в классе следующим образом:

```
void operator=(const FArray& Tarr);
```

где аргумент – ссылка на копируемый объект-массив.

В перегружаемом операторе сначала необходимо проверить, не происходит ли копирование объекта самого в себя, например, так:

```
A = A;
```

В этом случае не нужно ничего копировать, следует просто выйти из функции:

```
if (this == &Tarr) return;
```

Затем нужно проверить, совпадают ли размеры исходного и копируемого массивов:

```
if (Len != Tarr.Len)
```

и, если они **не совпадают**, освободить память и выделить новую.

```
Len = Tarr.Len; // Устанавливаем новую длину строки
```

```
if (Arr) // Если память уже выделялась
```

```
{
```

```
    delete[] Arr; // освобождаем память
```

```
    Arr = 0;
```

```
}
```

```
Arr = new float[Len]; // Выделяем новую
```

Далее последовательно копируем элементы из переданного в качестве аргумента объекта-массива в исходный:

```
for (UINT i = 0; i < Len; i++)
```

```
    Arr[i] = Tarr.Arr[i]; // Копируем элементы
```

На этом оператор-функция завершается.

```
void FArray::operator=(const FArray& Tarr) // Копирование
```

```
{
```

```
    if (this == &Tarr) return; // Если копирование самого в себя
                                // не копируем, а выходи из функции
```

```
    if (Len != Tarr.Len) // Если длины отличаются
```

```
{
```

```
        Len = Tarr.Len; // Устанавливаем новую длину строки
```

```
        if (Arr) // Если память уже выделялась
```

```
{
```

```
            delete[] Arr; // освобождаем память
```

```
            Arr = 0;
```

```
}
```

```
        Arr = new float[Len]; // Выделяем новую
```

```
}
```

```
for (UINT i = 0; i < Len; i++)
```

```
    Arr[i] = Tarr.Arr[i]; // Копируем элементы
```

```
}
```

Чтобы иметь возможность делать множественное присваивание, например, так:

```
A = B = C = D;
```

нужно возвращать ссылку на сам объект, для чего используется разыменованный указатель **this**. В заголовке оператор-функции возвращаемое значение будет ссылкой на объект класса-массива:

```
const FArray& FArray::operator=(const FArray& Tarr);
```

а оператор **return** будет возвращать **\*this**.

```
const FArray& FArray::operator=(const FArray& Tarr) // Копирование
{
    if (this == &Tarr) return *this; // Если копирование
                                     // самого в себя - не копируем
    if (Len != Tarr.Len) // Если длина другая
    {
        Len = Tarr.Len; // Устанавливаем новую длину строки
        if (Arr) // Если память уже выделялась
        {
            delete[] Arr; // освобождаем память
            Arr = 0;
        }
        Arr = new float[Len]; // Выделяем новую
    }
    for (UINT i = 0; i < Len; i++)
        Arr[i] = Tarr.Arr[i]; // Копируем элементы
    return *this; // Возвращаем ссылку на сам объект
}
```

### Перегрузка операции сравнения

Чтобы сравнить два массива на равенство с помощью операции **==** перегрузим ее. В ней сначала сравниваем длины массивов, и, если они равны, сравниваем массивы поэлементно. При несовпадении элементов, возвращаем **false**.

```
bool FArray::operator==(const FArray & Tarr) // Сравнение
{
    // массивов на полное равенство
    if (Len != Tarr.Len) return false;
    for (UINT i = 0; i < Len; i++)
        if (Arr[i] != Tarr.Arr[i]) return false; // сравниваем элементы
    return true;
}
```

Можно добавить операцию сравнения на неравенство, используя при этом результат предыдущей операции:

```
bool FArray::operator!=(const FArray& Tarr)
{
    return !(operator==(Tarr));
}
```

Объявления перегруженных операций добавляем в класс.

```
class FArray
{
    . . . . .
public:
    . . . . .
    const FArray& operator=(const FArray& Tarr); // Копирование
    bool operator==(const FArray & Tarr); // Сравнение на равенство
    bool operator!=(const FArray& Tarr); // Сравнение на неравенство
    . . . . .
};
```



## Копирование части массива

Для копирования части массива напомним соответствующую функцию. Чтобы иметь возможность копировать часть одного массива в другой (например, элементы с **iBeg** по **iEnd** включительно), напомним функцию, в которую вставим следующие аргументы: ссылку на массив в который копируем (**A1**), ссылку на массив из которого копируем (**A2**), номера элементов **iBeg** и **iEnd**. Размеры массивов передавать в качестве параметров не нужно, т.к. сами объекты-массивы знают свой размер.

Перед непосредственным копированием в функции необходимо сделать проверки на копирование массива самого в себя, на проверку корректности начального и конечного индексов. Далее определяем размер нового массива, с учетом того, что копирование будет проводиться с индексов **iBeg** по **iEnd** включительно, выделяем память и копируем.

```
void Copy(FArray& A1, FArray& A2, UINT iBeg, UINT iEnd)
{
    // Копирование с iBeg по iEnd
    if (&A1 == &A2) return; // Если копирование самого в себя
                                // - не копируем
    if (iEnd >= A2.GetSize()) // Если конечный инд. вышел за предел
        iEnd = A2.GetSize() - 1; // делаем его равным конечному
    if (iEnd <= iBeg) return; // Если начальный индекс >= конечного -
                                // не копируем
    UINT L = iEnd - iBeg + 1; // Устанавливаем новую длину строки
    A1.SetSize(L); // Изменяем размер массива
    for (UINT i = iBeg, j = 0; i <= iEnd; i++, j++)
        A1[j] = A2[i]; // Копируем элементы
}
```

## Использование класса

В тестовой программе нам понадобятся функции, которые задают размеры массивов, заполняют массивы случайными числами или по порядку, выводят массивы на дисплей. Эти функции, как и предыдущая, будут иметь в качестве аргумента ссылку на массив, с которым нужно работать.

```
void SetSize(FArray& Ta) // Задать размер массива
{
    int k;
    cout << "Задать размер массива: ";
    cin >> k;
    Ta.SetSize(k);
}

void FillArrayStep(FArray& Ta) // Заполнить по порядку
{
    int kbeg, kstep;
    cout << "Начальное значение: ";
    cin >> kbeg;
    cout << "Шаг: ";
    cin >> kstep;
    for(int i = 0; i < Ta.GetSize(); i++)
        Ta[i] = i * kstep + kbeg;
}

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

Для тестирования создадим в программе два объекта-массива, а в меню пункты для тестирования всех возможностей класса:

```
int main()
{
    FArray A(30), B(20); // Объявление объектов-массивов
    int w, k, n;

    setlocale(0, "Russian");
    cout.fixed;

    for (;;) // Бесконечный цикл
    {
        try // охранный блок
        {
            cout << "\n\nМеню:\n";
            cout << "0 - выход из программы\n";
            cout << "1 - Задать размер A\n";
            cout << "2 - Задать размер B\n";
            cout << "3 - заполнить A по возрастанию: \n";
            cout << "4 - заполнить B по возрастанию: \n";
            cout << "5 - просмотр A \n";
            cout << "6 - просмотр B \n";
            cout << "7 - сравнить A == B \n";
            cout << "8 - копировать B = A \n";
            cout << "9 - копировать часть из A в B \n";
            cout << "10 - умножить элемент массива A на 100 \n";

            cin >> w;
            switch (w)
            {
                case 0: // выход
                    return 0;
                case 1: // Размер A
                    SetSize(A);
                    break;
                case 2: // Размер B
                    SetSize(B);
                    break;
                case 3: // по возрастанию A
                    FillArrayStep(A);
                    break;
                case 4: // по возрастанию B
                    FillArrayStep(B);
                    break;
                case 5: // просмотр A
                    ViewArray("A", A);
                    break;
                case 6: // просмотр B
                    ViewArray("B", B);
                    break;
                case 7: // сравнить массивы
                    if (A == B)
                        cout << "Массивы равны" << endl;
                    else
                        cout << "Массивы НЕ равны" << endl;
            }
        }
    }
}
```

```

        break;
    case 8: // Скопировать В в А
        B = A;
        break;
    case 9: // Скопировать часть из А в в
        cout << "Задать начальный индекс: ";
        cin >> k;
        cout << "Задать конечный индекс: ";
        cin >> n;
        Copy(B, A, k, n);
        break;
    case 10: // Выбираем элемент массива
        cout << "Задать индекс: ";
        cin >> k;
        A[k] *= 100; // Здесь может быть исключение
        break;

    }
} // Конец охранного блока
catch (int e) // Блок перехвата
{
    if (e == 5)
        cout << endl << "Неверный индекс!" << endl;
    else
        cout << endl << "Неизвестная ошибка!" << endl;
} // Конец блока перехвата
} // Конец бесконечного цикла
return 0;
}

```

### Доработка класса-массива объектов типа CTime

Аналогично можно доработать класс-массив объектов **CTime**, разработанный в предыдущей лабораторной работе.

```

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
};

```

Для этого по аналогии с классом **FArray**:

- Напишем метод класса, позволяющий задавать (изменять) размер массива, с сохранением, по возможности, прежних данных массива.
- Перегрузим операцию **[]** для обращения к элементу массива. В ней, для контроля корректности индекса, должны использоваться исключительные ситуации, которые будем перехватывать в основной программе в блоке try.
- Перегрузим операцию сравнения на равенство двух массивов.
- Перегрузим операцию копирования одного массива в другой (операция = ).

- Напишем функцию, позволяющую копировать часть одного массив в другой.
- Добавим в класс метод, позволяющий сортировать элементы массива, например, по возрастанию времени.

```
void TimesArray::SetSize(UINT k) // Задать/изменить размер массива
{
    CTime* tmp; // Временный указатель для нового массива
    tmp = new CTime[k]; // Выделяем память
    if (Arr) // Если память выделялась ранее
    {
        for (UINT i = 0; i < k && i < Len; i++)
            tmp[i] = Arr[i]; // Копируем элементы
        delete[] Arr; // освобождаем память
    }
    Len = k; // Запоминаем новый размер
    Arr = tmp; // запоминаем в указателе адрес вновь выделенной памяти
}

bool TimesArray::operator==(const TimesArray& Tarr) // Сравнение
{
    // массивов на полное равенство
    if (Len != Tarr.Len) return false;
    for (unsigned int i = 0; i < Len; i++)
        if (Arr[i] != Tarr.Arr[i]) return false; // сравниваем элементы
    return true;
}

bool TimesArray::operator!=(const TimesArray& Tarr)
{
    return !operator==(Tarr);
}

void TimesArray::operator=(const TimesArray& Tarr) //
{
    // Копирование
    if (this == &Tarr) return; // Если копирование самого в себя -
    // не копируем, а выходим из функции
    if (Len != Tarr.Len) // Если длина другая
    {
        Len = Tarr.Len; // Устанавливаем новую длину строки
        if (Arr) // Если память уже выделялась
        {
            delete[] Arr; // освобождаем память
            Arr = 0;
        }
        Arr = new CTime[Len]; // Выделяем новую
    }
    for (UINT i = 0; i < Len; i++)
        Arr[i] = Tarr.Arr[i]; // Копируем элементы
}
```

Функции копирования части массива и сортировки предлагается написать самим.

## ЭКСПЕРИМЕНТАЛЬНАЯ ЧАСТЬ

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

### Задание

Доработать класс **Массив объектов** из лабораторной работы 26 и программу, иллюстрирующую возможности данного класса.

Методы доступа к элементам класса должны включать проверку выхода за пределы массива, при этом обязательно использовать исключительные ситуации и перехватывать их в программе с помощью блоков **try-catch**.

Класс должен включать методы (или глобальные функции) позволяющие:

- задавать (изменять) размер массива, **с сохранением**, по возможности, прежних данных массива;
- обращаться к элементам массива (перегрузить **операцию []**);
- сравнивать два массива на полное равенство (перегрузить **операцию ==**);
- копировать один массив в другой (перегрузить **операцию =**);
- *копировать часть одного массив в другой; \**
- *сортировать массив (параметр, по которому проводится сортировка, зависит от объектов); \**

Все проверки корректности ввода должны проводиться в методах класса.

В программе должно быть два объекта данного класса, а также меню с пунктами, позволяющими проиллюстрировать их возможности:

- задать (изменить) размер каждого объекта-массива;
- 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 любого объекта-массива, выбрав его по номеру (индексу);
- *задать свойства элементов объектов-массивов случайными величинами; \**
- *задать свойства элементов объектов-массивов с некоторым шагом; \**
- *сортировать любой объект-массив; \**
- скопировать один объект-массив в другой целиком;
- *скопировать заданную часть одного объекта-массива в другой; \**
- сравнить объекты-массивы.

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