

# СОДЕРЖАНИЕ

<b>Библиотека typedPL.....</b>	3
<b>Подключение библиотеки typedPL.....</b>	4
1. Использовать ссылку на директорию .h файла: .....	4
2. Копирование файла в папку проекта .....	4
<b>Компонент FunctionalLIB_Array (не поддерживается) .....</b>	6
Функции и методы для статических и динамических массивов стандарта C++ .....	6
<b>Компонент FunctionalLIB_Array – класс List (NEW).....</b>	16
1. Операторы и размерность .....	18
1. 2. Функции класса.....	19
3. Методы класса (манипуляция массивом) .....	22
<b>Компонент FunctionalLIB_String .....</b>	28
1. Стандартные операторы и методы класса .....	28
2. Функции класса .....	31
3. Методы класса (манипуляции строками) .....	34
<b>Компонент FunctionalLIB_File.....</b>	38
1. Чтение и запись файлов.....	38
2. Действия над файлами (не работает на коротких ссылках).....	40
3. Получение информации о файлах (не работает на коротких ссылках) .....	41
<b>Компонент FunctionalLIB_CustomConsole.....</b>	43
1. Методы вне класса ConsoleSetting .....	43
2. Рамка консольного приложения (изменение свойств).....	45
3. Пространство консольного приложения .....	49
4. Класс анимации .....	55
<b>Компонент GeometryLIB_Figure .....</b>	58
1. Периметр.....	58
2. Площадь .....	61
3. Объем .....	68
4. Площадь поверхности .....	72
5. Иные формулы.....	73
<b>Константы GeometryLIB_Figure .....</b>	74
<b>Компонент GeometryLIB_Expression.....</b>	75
1. Класс для работы с комплексными числами .....	75
2. Класс для работы с векторами .....	76
<b>ПРИЛОЖЕНИЕ FunctionalLIB_Array (не поддерживается) .....</b>	78

ПРИЛОЖЕНИЕ FunctionalLIB_Array – Class List.....	82
ПРИЛОЖЕНИЕ FunctionalLIB_String.....	85
ПРИЛОЖЕНИЕ FunctionalLIB_File .....	89
ПРИЛОЖЕНИЕ FunctionalLIB_CustomConsole .....	90
ПРИЛОЖЕНИЕ GeometryLIB_Figure .....	93
ПРИЛОЖЕНИЕ GeometryLIB_Expression .....	97
BETA -0.10 Version 2.0 .....	99
BETA -0.10 Version 3.0 .....	101
BETA -0.10 Version 3.2 .....	106
BETA -0.10 Version 3.5 .....	113

## **Библиотека typedPL**

**Описание:** Библиотека, содержащая в себе полезные функции, которые не включены в стандарт C++. Данная библиотека позволит более гибко работать с массивами, без необходимости создавать громоздкие алгоритмы непосредственно в коде программы.

Имеется возможность более гибко работать со строками благодаря уникальным методам, перекрывающим стандарт языка C++. Несмотря на принадлежность к разным классам и пространствам имен, массивы могут взаимодействовать со строковым String из данной библиотеки (если речь идет о кастомном контейнере List).

TypedPL позволяет взаимодействовать с файлами и консольным приложением. Вы можете записывать, считывать, создавать, перемещать, а также осуществлять другие операции над файлами, а также изменять параметры консольного приложения с помощью функций, построенных на основе WinAPI.

Помимо этого, typedPL содержит набор функций по вычислению периметра, площади, и объема различных геометрических фигур. Это поможет быстрее работать с геометрическими объектами, а также в решении задач на вычисления их параметров.

**На данный момент библиотека содержит два глобальных модуля:**

- FPFC.h;
- Geometry.h.

**Компоненты FPFC.h:**

- FunctionalLIB\_Array;
- FunctionalLIB\_String;
- FunctionalLIB\_File;
- FunctionalLIB\_CustomConsole.

**Компоненты Geometry.h:**

- GeometryLIB\_Figure;
- GeometryLIB\_Expression;
- директива \_USE\_CONSTANT\_

...

# Подключение библиотеки typedPL

Для того, чтобы подключить данную библиотеку к своему проекту, можно воспользоваться несколькими способами:

## 1. Использовать ссылку на директорию .h файла:

Данный способ может быть полезен в тех случаях, когда нужно подключиться напрямую к файлу другого проекта, если не имеется возможности или желания воспользоваться вторым способом, о котором будет рассказано позже.

Для подключения ссылки переходим в свойства проекта, где во вкладке C++ находим параметр «Дополнительные каталоги включаемых файлов». Затем нажимаем «Изменить» и в открывшейся панели вставляем полный путь к файлу.

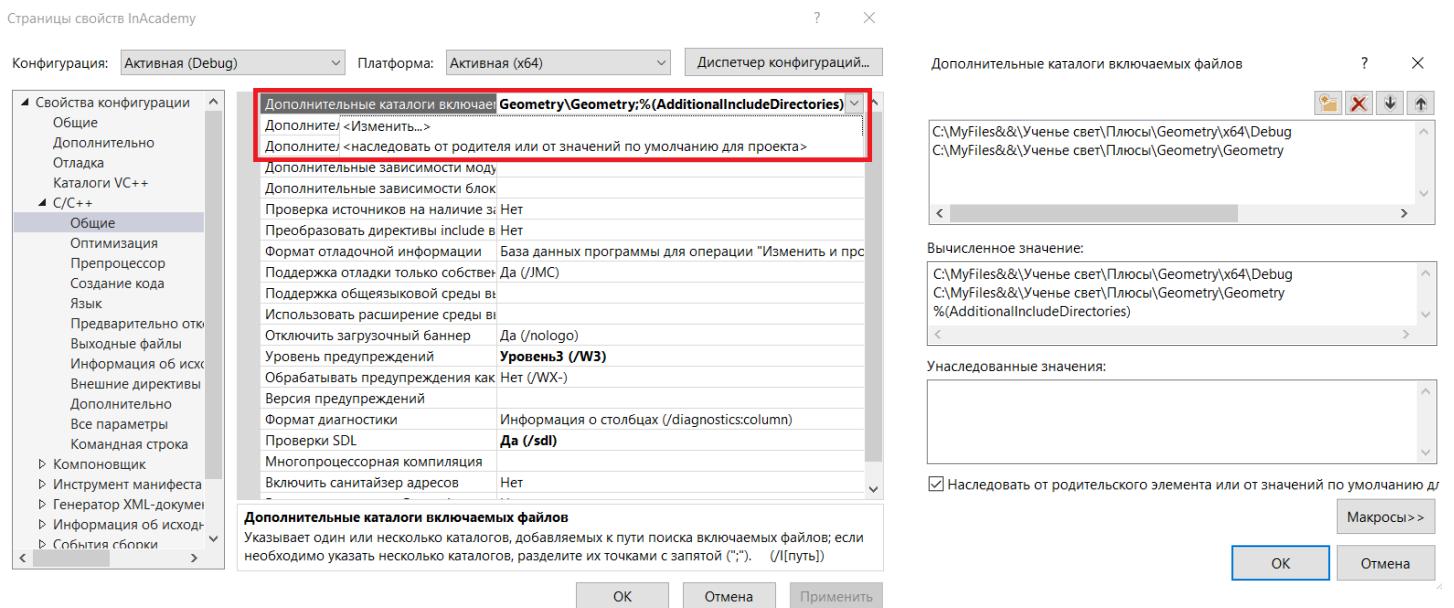


Рисунок 1 – Подключение dll через ссылку к файлу

Важно отметить, что при подключении, необходимо указывать путь к .h файлу проекта, а не к самому dll файлу, который формируется после компиляции.

## 2. Копирование файла в папку проекта

Данный способ реализовать достаточно просто, поскольку все, что необходимо, так это скопировать .h файл из проекта библиотеки в нужный вам проект. Теперь данный файл будет доступен для подключения через include. Если скопированный файл не появляется в обозревателе решений проекта, даже после пересборки или перезагрузки приложения – это значит, что данный файл невозможно подключить к проекту.

Несмотря на то, какой способ подключения вы выбрали – **подключается файл следующим образом:**

```
#include "Geometry.h" // для модуля геометрии  
#include "FPFC.h" // для функционального модуля (сокращение от названия functional pack for C++)
```

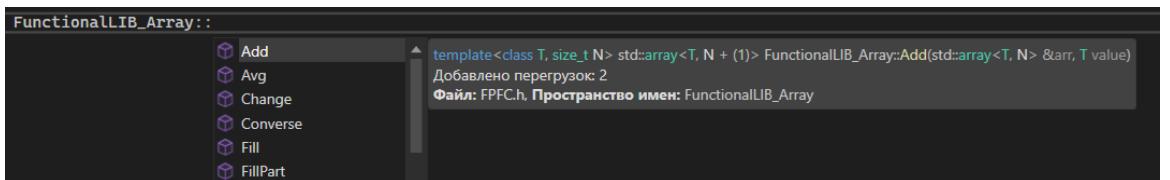
Теперь вы можете воспользоваться всеми функциями данной библиотеки для написания своих программ.

Для получения доступа к функционалу библиотеки, **вы должны сначала объявить пространство имен, с которым будете работать**. После чего вы можете выбирать любую функцию объявленного модуля.

```
array<string, 3> arr = { "start", "center", "end" };  
FunctionalLIB_Array::Converse(arr); // разворот массива
```

Рисунок 2 – Пример использования функций библиотеки

При затруднении написания аргументов функций библиотеки, можете воспользоваться **подсказками о структуре функции от Visual Studio**:



В данной панели показаны названия функций, их параметры и типы значений, которые принимают аргументы. Также показан файл, к которому отнесен тот или иной метод, а также пространство имен, в которые заключена функция.

Также во время заполнения аргументов вы можете вновь воспользоваться подсказками Visual Studio:

```
char sms[] = "marry christmas!";  
FillPart(sms, );  
system("n")  
▲ 2 из 3 ▼ T FillPart<T>(T arr[], int size, T value, int start, int end)
```

Если вы все равно испытываете затруднения в использовании функций typedPL библиотеки – можете прочитать более подробную документацию по каждой функции ниже.

Запись, показанная [на рисунке 2](#) не совсем удобна, в частности в читабельности кода. Поэтому **данную запись можно сократить**, объявив пространство имен заранее.

```
using namespace FunctionalLIB_Array;
```

Теперь наша запись может иметь следующий вид:

```
using namespace GeometryLIB_Figure;  
cout << Polygon2D_Side(10, 34);  
// вывод площади правильной 34-х угольной фигуры со стороной 10
```

## Компонент FunctionalLIB\_Array (не поддерживается)

Прежде чем приступить к описанию всех функций данного компонента, следует отметить, что **каждая функция FunctionalLIB\_Array является перегруженной**. То есть принимает на вход различные аргументы, но при этом имеет одно и тоже имя.

Это было сделано для того, чтобы не нагружать пользователя большим количеством разных функций, которые делают примерно одно и тоже. Таким образом, **каждая функция выполняет работу**:

- с перечислением параметров;
- с динамическим массивом;
- со статическим массивом;
- с динамическим массивом типа `array<type, size>`.

Данный компонент поддерживает работу с различными типами массивов, поскольку в C++ для работы с каждым из перечисленных типов – имеется разный алгоритм действия. В случае со статическими массивами – данный компонент намного упрощает алгоритм реализации многих стандартных функций над массивами. В случае с динамическим `vector` массивом – дополняет функциями, которые нет в обыденном пространстве имен `<vector>`.

Таким образом, **данная библиотека объединяет разные алгоритмы работы с массивами и подводит их под единый стандарт записи**.

**ОСНОВНОЕ, ЧТО НУЖНО ЗНАТЬ:**

**Аргументы функций-перечислений:** `FUNC (arg1, arg2... argN);`

**Доступные для работы массивы:**

`Vector<type> name;`

`Type name[size];`

`Array<type, size> name;`

**Функции и методы для статических и динамических массивов стандарта C++**

### 1. Сумма

**Данная функция принимает следующие параметры:**

```
template<typename T, typename... Args> double Sum(T arg, Args... args){ ... }

template<typename T> double Sum(vector<T> arr){ ... }

template<typename T, size_t N> double Sum(T(&arr)[N]){ ... }

template<typename T, size_t N> double Sum(array<T, N> arr){ ... }
```

Подобный шаблон аргументов для функции будет во многом схож с другими функциями библиотеки.

### Примеры использования:

```
long double st[] = { 4,4,2,5,1 };
array<int, 5> arr = { 0,2,0,0,1 };
vector<float> dinamic = { 4,11,2,3 };
int a = -23;

cout << Sum(st) << " " << // 16
    Sum(arr) << " " << // 3
    Sum(dinamic) << " " << // 20
    Sum(M_PI, 30.1, a); // 10.2416
```

Используя один метод Sum, мы можем находить сумму неопределенного количества аргументов. Возможность работы с различными типами массивов позволяет более гибко работать с ними и стандартизировать их запись.

Помимо заранее известных аргументов, можно также использовать аргументы из разных массивов, разных структур, или вводить значения собственноручно.

```
map<string, int> Count{
    {"Milk", 54},
    {"Eat", 21}
};

cout << Sum(Count.at("Milk"), Count["Eat"]); // Sum принимает два аргумента
```

(обращаться к элементам map можно обращаться по двум вариациям, разницы нет)

## 2. Среднее значение

Данная функция имеет такие же перегрузки метода, как и Sum. Поэтому ее использование будет иметь такой же синтаксис:

```
double numUser, IdNum, IndexUtility;
cin >> numUser >> IdNum >> IndexUtility;
long double st[] = { numUser, IdNum, IndexUtility };

array<int, 5> arr = { 0,2,0,0,1 };
vector<float> dinamic = { 4.2f,11.21,2.2f,3 };

cout << Avg(st) << " " << // среднее чисел, введенных пользователем
    Avg(arr) << " " << // 0.2
    Avg(dinamic) << " " << // 5.15.25
    Avg(M_LN10, M_SQRT1_2, IndexUtility); // среднее аргументов
```

Пример: предположим, необходимо подсчитать среднее количество поинтов, которые дают за прохождения по 1 уровню каждой сложности, имеющейся в игре. Это можно реализовать с помощью функции Avg следующим образом:

```

enum list { // список всех сложностей игры
    easy = 1,
    normal = 3,
    hard = 5,
    insane = 10
};

array<double, sizeof(list)> difficultAvg = { easy, normal, hard, insane };
cout << Avg(difficultAvg);

```

### 3 и 4. Максимальное и Минимальное

**Максимум и минимум идентичны предыдущим двум примерам.** Данные функции являются мощным инструментом, поскольку поддерживают неограниченное количество элементов. К примеру, если min и max из библиотеки math позволяют сравнивать только два числа, а, чтобы сравнить больше: необходимо воспользоваться либо макрос конструкцией, либо тернарным оператором. То благодаря Min / Max библиотеки Geometry – это можно сделать более быстрее и удобнее.

```

int numbers[int(3)]{};
puts("Введите ряд чисел:");
for (int i = 0; i < Lgth(numbers); i++)
    cin >> numbers[i];

cout << Max(numbers);

```

**Главное преимущество данного метода – это его простота реализации.** Часто, при сравнении нескольких аргументов, приходится использовать один из предоставленных ниже алгоритмов.

```

int a = 23, b = 17, c = -34;
cout << min(a, min(b, c)); // использование min из math.h

int min = (a < b) ? ((a < c) ? a : c) : ((b < c) ? b : c);
cout << min; //тернарная конструкция для сравнения

```

3 чисел

```

int max = a;

if (b > max)
    max = b;
if (c > max)
    max = c;

cout << max; // сравнение трех чисел по условному оператору if

```

Однако, все эти алгоритмы будет гораздо труднее реализовать, когда будет необходимость сравнивать десятки и более чисел. Поэтому функции Min / Max из библиотеки typedPL облегчат эту задачу.

```

cout << Max(a, b, c); // Max библиотеки typedPL

```

В случае, если вы не желаете пользоваться не одним из выше перечисленных способов.

**Вы можете воспользоваться конструкцией из библиотеки algorithm**, которая выглядит так:

```
cout << max({ a,b,c }); // max библиотеки algorithm
```

Практически две идентичные записи (Max из typedPL и max из algorithm) за исключением того, что в качестве аргумента, последний принимает листовый массив из пространства имен initializer\_list, а затем обрабатывает этот аргумент как единый массив.

## 5 и 6. Сортировка и переворот

Данные функции уже не поддерживают одиночные аргументы, а **работают только с массивами**. Принимают в качестве аргументов – тот же список, что и на рисунке в пункте 1.

**Сортировка массива:**

```
Sort(dinamic); // сортировка по возрастанию
```

Для того, чтобы сортировать массив по убыванию, можно воспользоваться функцией Converse, которая переворачивает массив.

**Сортировка по убыванию:**

```
Sort(arr);
Converse(arr); // сортировка по убыванию
```

Помимо чисел, Sort может взаимодействовать и с такими данными – как **символы или строки**. К примеру, отсортировать массив символом можно следующим образом:

```
char charSort[] = {'a', 'f', 'b', 's'};
Sort(charSort, sizeof(charSort));
for (auto s : charSort)
    cout << s << "\n";
```

P.S sizeof крайне полезен, когда нужно найти длину статического массива (если она вам заранее известна).

P.S x2 - В качестве альтернативы можно использовать функцию Lgth

Следует отметить, что порой компилятор может выводить сообщение о том, что функция является неоднозначной, но несмотря на наличие ошибки, воспроизводить работу. В этом случае можете перезапустить проект, чтобы убрать данное сообщение. Подобное уведомление возникает из-за того, что при использовании разных библиотек – названия методов могут совпадать, что приводит к неопределенности используемого метода.

## 7. Инверсия

**Инверсия поддерживает как массивы, так и одиночные аргументы.**

```

for (int i = 0; i < dinamic.size(); i++)
{
    cin >> num;
    cout << Inversion(num) << " ";
}
// инверсия одиночных аргументов

Inversion(dinamic);
for (auto s : dinamic)
    cout << s; // инверсия динамического массива

```

Инверсия не работает со строковыми типами данных, поскольку в алгоритм обработки аргументов входит операция умножения на -1. Поэтому при попытке инверсии строки будет выведено сообщение о невозможности выполнения операции.

## 8. Рандомы данных

В некоторых случаях необходимо воспользоваться рандомными значениями. Для этого в библиотеке typedPL содержится два перегруженных метода: **RandInt** и **RandDouble**. Первый метод – генерирует рандомные (не псевдорандомные) значения целых чисел. Второй, соответственно, вещественных типа double.

**Данные функции имеют немного иной шаблон заполнения аргументов:**

```

template<typename T> int RandInt(T min, T max){ ... }

template<typename T> int RandInt(T min, T max, vector<T>& arr){ ... }

template<typename T, size_t N> int RandInt(T min, T max, T(&arr)[N]){ ... }

template<typename T, size_t N> int RandInt(T min, T max, array<T, N>& arr){ ... }

```

Сначала указывается диапазон значений от минимума, до максимума. После чего тип массива.

Данные функции можно использовать для генерации рандомных последовательностей и заполнения массивов данными:

```

RandInt(-200, 500);
// получение рандома в диапозоне

array<double, 100> arrF;
RandDouble(-10.5, 10.5, arrF);
// заполнение массива рандомом в диапозоне

```

**Пример для динамических массивов:**

```

vector<int> arrINT(10); // указание размерности заранее
RandInt(0, 1, arrINT);
for (int s : arrINT)
    cout << s << "\n";

vector<double> arrDOUBLE;
arrDOUBLE.resize(5); // указание размерности через resize
RandDouble(0.0, 1.0, arrDOUBLE);
for (auto s : arrDOUBLE)
    cout << s << "\n";

vector<int> arrINT2 = {0,0,0}; // указание размерности "пустыми"
for (int s : arrINT2)           // значениями
{
    s = RandInt(-20, 35);
    cout << s << "\n";
}

```

## 9. Медиана

Данная функция работает исключительно с массивами. Выводит медиану массива.

Медиана – центральное значение в массиве.

**Пример применения:**

```

float arr[] = {0.0f, 1.2f, 5.1f, -31.2f};
cout << Median(arr);

```

Данный метод точно работает с массивами, чья размерность является нечетными. В ситуациях, когда **массив является четным**, то функция **Median** выведет среди двух центральных элементов - правый. Но если надо вывести оба элемента, то можно воспользоваться следующей конструкцией:

```

float arr[] = {0.0f, 1.2f, 5.1f, -31.2f};
int size = Lgth(arr) / 2;
cout << arr[size - 1] << " " << arr[size];

```

## 10. Заполнение и Частичное заполнение

Еще одна мощная функция в результате которой можно заполнять массивы нужными значениями, а также изменять их независимо от их индекса.

Для полного заполнения массива используется **Fill**, а для частичного – **FillPart**. **Их входные аргументы отличаются от выше описанных функций:**

```

template<typename T> T Fill(vector<T>& arr, T value) { ... }

template<typename T, size_t N> T Fill(T(&arr)[N], T value) { ... }

template<typename T, size_t N> T Fill(array<T, N>& arr, T value) { ... }

```

Указывается тип массива, а затем значение, которым массив будет заполнен. В случае статического массива, как и в других ситуациях – необходим аргумент длины массива.

```

char str;
array<char, 10> strFILL{};
puts("Выберите, каким значением вы хотите заполнить массив:");
cin >> str;
Fill(strFILL, str);

```

// пример заполнения пустого массива значением, введенным пользователем

**Для частичного заполнения аргументы выглядят так:**

```

template<typename T> T FillPart(vector<T>& arr, T value, int start, int end){ ... }

template<typename T, size_t N> T FillPart(T(&arr)[N], T value, int start, int end){ ... }

template<typename T, size_t N> T FillPart(array<T, N>& arr, T value, int start, int end){ ... }

```

Также указывается:

- тип заполняемого массива;
- значения, на которое нужно поменять какую-то область;
- диапазон индексов (start и end).

При этом диапазон указывается не как в программировании (с нуля), а как при счете, начиная с единицы.

```

vector<double> dinamic = { 4,11,2,3 };
FillPart(dinamic, 0.0, 2, 3);
// Заполнение нулями 2 и 3 элемента массива

```

Важно учитывать, что заменяемое значение должно соответствовать типу массива.

Поэтому в качестве второго аргумента идет значение 0.0, которое относится к типу double.

**Примеры использования двух функций:**

```

// заполнение arr нулями
array<long float, 5> arr;
Fill(arr, 0.0);

// стирание первого слова
char sms[] = "Marry Christmas!";
FillPart(sms, ' ', 1, 5);

for (auto s : sms)
    cout << s;

```

## 11. Изменение значений массива

Функция Change позволяет заменять одни элементы массива на другие. Данная функция является аналогом Replace.

**Функция принимает следующие параметры:**

```

template<typename T> T Change(vector<T>& arr, T found, T replace){ ... }

template<typename T, size_t N> T Change(T(&arr)[N], T found, T replace){ ... }

template<typename T, size_t N> T Change(array<T, N>& arr, T found, T replace){ ... }

```

То есть тип массива, искомый элемент, и значение, на которое меняется найденное совпадение.

## Пример замены:

```
vector<int> INT = { 23,1,6,2,7,1,-2,15,9,1,-1,
|     0,14,744,16,-21,5,2 };

Change(INT, 1, 0);
// замена всех единиц на нули
```

## 12. Объединение массивов

Объединение массивов работает только с массивами типа vector (то есть с динамическими массивами). В качестве аргументов, данный метод принимает лишь два параметра – первый и второй массив, которые объединяются в один.

```
vector<double> dinamic = { 4,11,2,3 };
vector<double> INT = { 23,1,6,2,7,1,-2,15,9,1,-1,
|     0,14,744,16,-21,5,2 };

for (auto s : Union(dinamic, INT))
    cout << s << " ";
```

## 13. Размерность массива

Ранее было показано несколько примеров, в которых использовалась функция Lgth (сокращение от Length). Данная функция предназначена для того, чтобы считывать размерность массива и возвращать его в качестве целочисленного типа данных.

Данную функцию достаточно легко заменить, поскольку для динамических массивов есть метод size, который выполняет идентичную функцию. **Прежде всего Lgth предназначен для считывания размерности статических массивов, заменяя тем самым sizeof(arr) / sizeof(arr[0])**. Однако, данная функция работает и с динамическими массивами, чтобы иметь общий стандарт записи массивов.

## Примеры использования:

```
struct MyStruct
{
    vector<string> str{
        "X??1", "DF_21", "*2_k",
        "^1", "Gg%__", "<St&le>",
        "W_W", "J00J00"
    };
};

MyStruct arrStruct;
cout << "Длина массива: " << Lgth(arrStruct.str);
```

## 14. Добавление элемента

Функция Add позволяет добавлять к массиву новые элементы. Аналогом данной функции является push\_back пространства имен <vector>. На данный момент не особо поддерживает взаимодействие со статическими массивами std::Array.

```
array<double, 2> arrDOUBLE = { 0.5,1.8 };
const int size = sizeof(arrDOUBLE) / sizeof(arrDOUBLE[0]);
array<double, size + 1> result = Add(arrDOUBLE, 2.4);
for (auto s : result)
    cout << s << "\n";
```

**Главная проблема работы со статикой** заключается в том, что **размерность подобных массивов должна быть заранее известна и быть константной**. Поэтому использование функции Lgth вызывает ошибку компиляции.

```
vector<int> arr0 = { 0,1 };
vector<int> arr0_1 = { 0,1 };

for (int i = 0; i < Lgth(arr0_1); i++)
    Add(arr0, arr0_1[i]);

for (int out : arr0)
    cout << out << " ";
// аналог Union с применением Add

typedef int INTEGER;
INTEGER arr[] = {1, 23, 1, 4, 1, 4};
for (auto s : Add(arr, 100))
    cout << s << "\n";
// добавление в статический массив числа
```

Следует отметить, что **шаблон с перегрузкой стандартного статического массива возвращает значение динамического массива**. Поэтому приравнивание статики к статике в данном случае не будет иметь никаких успехов.

Аналогичная запись будет выглядеть следующим образом:

```
int arr[] = { 1, 23, 1, 4, 1, 4 };
vector<int> result = Add(arr, 100);
for (auto s : result)
    cout << s << "\n";
```

Здесь мы создаем динамический массив, к которому добавляем значения из статического целочисленного arr.

## 15. LINQ имитация

**LINQ запросы крайне полезны, когда необходимо найти какую-либо информацию из массива данных.** Однако, в обычном C++, отсутствуют подобные инструменты и приходится использовать обходные пути, такие как использование итераций или использование стандартных инструментов из algorithm.

**LINQ функция в библиотеке typedPL лишь подобие универсальный и удобных запросов из профессиональных библиотек.** Ее функционал крайне ограничен и поддерживает лишь операторы сравнения в своем распоряжении. Однако, с использованием других функций библиотеки typedPL можно приблизить вычисления к настоящим LINQ запросам.

Шаблон LINQ функции состоит из:

```
template<typename T> bool LINQ_OPERATION(T arg, string operation, T paramIF) { ... }

template<typename T> vector<T> LINQ_OPERATION(vector<T>& arr, string operation, T paramIF) { ... }
```

Библиотека поддерживает на данный момент две перегрузки данной функции. Первая возвращает булевое значение аргумента. То есть 1 или 0, в зависимости от условия. Вторая работает непосредственно с динамическим массивом данных и способна выводить отсортированные по условию данные.

**Примеры использования:**

```
vector<double> result = LINQ_OPERATION(arr0, ">=", 1.5);
result = LINQ_OPERATION(result, "<=", 5.5);
Sort(result);
for (auto s : result)
    cout << s << " ";

// вывод значений отсортированного массива в диапазоне от 1.5 до 5.5

vector<double> arr = { M_PI, M_LOG2E, M_SQRT2, RandDouble(-1.0, 1.0) };
bool __TRUE = LINQ_OPERATION(arr[0], ">", Median(arr));
if (__TRUE)
    cout << "Утверждение верно!\n";

// сравнение двух значений массива (в данном случае первой константы и
константы, являющейся медианой массива)
```

## Компонент FunctionalLIB\_Array – класс List (NEW)

Данный класс является измененным подходом к использованию массивов и методов для работы с ними. Если в предыдущих пунктах создавались методы под разные типы массивов, то в данном классе представляется лишь один новый динамический массив.

**Синтаксис объявления массива:**

```
List<double> MyArray = { 0, 4.2, -5.33, TWINS_B };

List<size_t> SIZE{0,0,1,0};

List<string> _STR_ ( {"4", "строчка"});
```

*//объявление массива*

Данный массив принимает параметр в виде списка аргументов, заключенных в фигурные скобки. Выше показано три разных примера объявления динамического листового массива. Можно отметить, что по своему синтаксису переменные List класса идентичны динамическому vector, однако, не включают методы из других пространств имен (за исключением методов собственного класса).

Большинство методов перекочевали с предыдущих пунктов, описывающих данный компонент библиотеки typedPL.

Важная особенность данного класса состоит в том, что **все методы привязаны к экземпляру класса List**, что позволяет **сократить количество аргументов**. (То есть нам не нужно указывать обрабатываемый массив в качестве аргумента).

**Для сравнения:**

```
List<double> MyArray = { 0, 4.2, -5.33, TWINS_B };
cout << MyArray.Sum();

vector<double> MyArray2 = { 0, 4.2, -5.33, TWINS_B };
cout << Sum(MyArray2);
```

*// использование LIST*

*массивов и стандартных методов из библиотеки*

С одной стороны, разница на первый взгляд не кажется существенной. Однако, на дистанции будет видно, что **применение LIST массивов наиболее удобное**, поскольку при повторении множества операций нет необходимости прописывать повторяющийся аргумент (так как он автономно изменяется внутри класса).

Помимо этого, возможность класса более гибкая, чем его отсутствие – в первую очередь, класс поддерживает модификаторы доступа, что могут скрыть некоторые элементы и использоваться только внутри класса. Во-вторых, класс позволяет создать конструкторы, что в свою очередь позволяет создавать цепочки методов.

**Рассмотрим еще один пример сравнения использования класса List и методов вне класса:**

```
List<double> MyArray = { 0, 4.2, -5.33, TWINS_B };
MyArray.Inversion()
    .Sort()
    .Converse(); // сортировка с поворотом позволяет
                  // сделать сортировку по убыванию

for (double Out : MyArray.Union(MyArray)) // дублирование
    cout << Out << " ";
// List class
```

В данном примере мы производим операции инверсии и сортировки по убыванию. А также при выводе массива – дублируем его результаты. Как видно на скриншоте, для реализации мы можем использовать цепочку методов, что является более удобным (*также о цепочках методов вы можете узнать в разделе FunctionalLIB\_ConsoleSetting или в сноске об обновления документации в конце документа*).

```
vector<double> MyArray2 = { 0, 4.2, -5.33, TWINS_B };
Inversion(MyArray2);
Sort(MyArray2);
Converse(MyArray2);

for (double Out : Union(MyArray2, MyArray2))
    cout << Out << " ";
// использование методов
```

для стандартных массивов

Как видно, здесь мы не можем использовать цепочку методов, поскольку все методы находятся в каком-либо конкретном классе.

### **РАЗДЕЛЕНИЕ КЛАССА.**

Класс можно условно разделить на функции и функции, возвращающие значения класса. Первые возвращают значение и являются заключительными элементами любой цепочки. Вторые же являются связующими цепочки и изменяют массив структурно: например, добавляют новые элементы или изменяют их.

**Примеры функций:** Count, Max, Min, Sum – возвращают конкретное значение (будь то длина, максимальное значение, сумма, и так далее).

**Примеры функций, влияющих на содержимое массива:** Converse, Union, Fill – изменяют массив структурно (переворачивают его, сортируют, заменяют элементы, и так далее).

### **МЕТОДЫ КЛАССА:**

## 1. Операторы и размерность

Класс `List` поддерживает несколько операторов и функций для определения размерности массива класса.

```
List<double> a = { 0,3 };
a = a + a; // 0 3 0 3
for (auto s : a)
    cout << s << " ";
// сложение массивов в один (0 3 0 3)

List<double> a = { 0,3 };
a = a.operator+({1,4});
for (auto s : a)
    cout << s << " ";
// добавление к массиву N элементов (0 3 1 4)

List<double> a = { 0,3 };
initializer_list<double> b = { 1,4,2 };
a = a + b;
for (auto s : a)
    cout << s << " ";
// еще одна реализация добавления N
элементов (0 3 1 4 2)
```

Оператор `<+>` имеет несколько перегрузок – одиночного аргумента, массива, неопределенного количества аргументов. Первая перегрузка добавляет к каждому элементу массива какой-либо аргумент (тип `double`), вторая перегрузка позволяет объединять массивы в один. Третья перегрузка добавляет в массив новые аргументы (принимает параметр `initializer_list<T>`).

```
List<double> a = { 0,3 };
a = a + 10, a * 3;
// сначала сложение всех на 10, затем их умножение на 3,
a = a + 10 + a * 3;
// сложение на 10, плюс массив, умноженный на 3
//(РЕКОМЕНДУЮ ИСПОЛЬЗОВАТЬ ПЕРВУЮ ЗАПИСЬ)
for (auto s : a)
    cout << s << " ";
// операторы для
каждого элемента массива (ответы: 30 39 И 120 127)
```

Элементы можно складывать, вычитать, умножать, делить на определенный аргумент. Существует несколько типов записей. Рекомендуется использовать запись через запятую или использовать operator напрямую:

```
List<double> a = { 0,3 };
a = a.operator+(10).operator*(3); // использование operator напрямую
```

```

string a; cin >> a;
TEST[0] = a;

for (int i = 0; i < TEST.Count(); i++)
    cout << TEST[i] << " ";
// использование оператора «[]»

```

**Благодаря существованию оператору «[]» мы можем использовать индексы массива** (то есть имеется в виду обращаться к массиву по индексу). Без данного оператора была бы невозможна работа с классом List, как показано на скриншоте выше.

```

typename vector<T>::iterator begin() { return value.begin(); } // считывание начала массива
typename vector<T>::iterator end() { return value.end(); } // считывание конца массива

```

**Класс List также содержит два итератора, благодаря которым возможно считывание массива циклом-переборкой** (цикл типа foreach). Однако, поскольку использование данных итераторов является не всегда удобным, было добавлено две функции:

*First – начало массива;*

*Last – конец массива.*

Данные функции позволяют обойти использование итераторов в цикле, тем самым упрощая и сокращая запись:

```

for (int i = TEST.First(); i != TEST.Last(); i++)
    cout << TEST[i] << " ";
// использование функций First,

```

*Last.*

Функция First константно равняется нулю, поскольку любой индекс массива начинается с нуля. Функция Last же возвращает размерность массива.

Вы также можете вернуть последний элемент, воспользовавшись функцией **LastElement**. Или же первый – **FirstElement**.

**Размерность массива** также можно узнать с помощью функции **Count** (пример применения показан выше в примере с оператором «[]»).

## 2. Функции класса

### Sum, Avg, Min, Max, Median

Данные функции идентичны в структуре с их аналогами, располагаемыми вне класса. Кратко: вывод суммы элемента, вывод среднего элемента, вывод минимального / максимального, вывод центрального элемента.

**Sum и Avg – возвращают тип double.** Поэтому данные функции используются только с массивами данных, представляющими из себя числовые значения.

**Min, Max, Median способны возвращать любой параметр** (поскольку в отличии от той же суммы, строку можно сравнить по размерам – то есть найти максимум и минимум).

Рассмотрим несколько примеров:

```
List<float> arr{};  
int Len = RandInt(1, 5);  
for (; Len != 0; Len--)  
{  
    static float input;  
    cin >> input;  
    arr.Add(input);  
}  
  
cout << "Сумма ряда равна: " << arr.Sum() << endl; // вывод суммы ряда,
```

*введенного пользователем*

```
typedef string STR;  
List<STR> arr{ "ВВОД", "ВЫВОД", "ПАРОЛЬ" };  
cout << arr.Min() << endl; // ВВОД  
cout << arr.Max() << endl; // ПАРОЛЬ //вывод максимального и
```

*минимального элемента массива строк*

```
ifstream readFILE("file.txt");  
List<string> file{}; string text;  
while (getline(readFILE, text))  
    file.Add(text);  
  
cout << "Центральный элемент файла: " << file.Median(); // определение
```

*среднего элемента текстового файла*

## GetElement

Иногда бывают ситуации, когда необходимо получить конкретное значение массива.

Обычно для этого используются индексы массива.

Библиотека TypedPL предлагает следующие решения подобных задач:

```
List<size_t> MyArray = { 0, 5, 1, 4 };  
size_t a = MyArray.Sort().operator[](MyArray.Last() - 1);  
cout << a; // сортировка массива и вывод последнего элемента  
  
List<unsigned long int> Size = { 10, 4, 200, 13 };  
cout << Size.Sort().GetElement(3);  
// получение элемента по индексу с GetElement  
  
List<wchar_t> symbol = { 'a', 'b', 'c', 'z' };  
cout << symbol.LastElement();  
// получение последнего элемента по LastElement  
  
cout << MyArray[MyArray.Count() - 1];  
// получение последнего элемента по индексам // примеры получения
```

*последнего элемента массива (в случае с 1, 2, 4 вариантом – возможно получение любого индекса, не только последнего)*

## Search

Булева функция для определения – имеются ли совпадения данных с массивом List.

Имеет несколько перегрузок: List массив, список значений.

```
List<float> FL = { 3, 1.4, -3.1f };
puts("Введите действие над массивом: ");
string comand; cin >> comand;

List<string> FUNCTIONAL_LIST{
    "Count",
    "Median",
    "Sum",
    "Converse",
    "RandInt",
    "operator[]"
};

bool flag = FUNCTIONAL_LIST.Search(comand);

flag ? puts("ECTЬ") : puts("НЕТ");
// применение Search для поиска
```

совпадений

## Equals

Сравнивает два массива между собой на их равнозначность. Если они равны – вывод 1 (true), иначе 0 (false). Изначально проверяет массив на их размерность. А если они равны – запускается проверка на значения внутри массивов.

```
List<int> A = { 0,1,2 };
List<int> B = { 0,1,2,3 };
bool equal = A.Equals(B);
cout << equal << endl; // 0

equal = A.Equals(A);
cout << equal << endl; // 1 (логично)
```

## ToCharArray, Split

**ToCharArray** необходим для разбиения строки на символы. Поскольку данный метод относится к классу List – применяется только к массивам (**ВАЖНО! Работает только с типом массива char**).

Принимает один параметр в качестве аргумента – строку, которую нужно преобразовать в символьный массив. (**ВАЖНО! При наличии каких-то значений – перезаписывает массив.** То есть, если массив был {'a', 'b'}, то при ToCharArray эти значения сотрутся).

```
string text = "Разбиение строки на символы!";
List<char> crArray{};
crArray.ToCharArray(text);
// разбиение на символы
```

**Split** используется для разбиения строки на подстроки. Как и ToCharArray переписывает массив, если в нем были какие-то значения до этого. **Работает только с типом string.**

```
string text = "Попробуй использовать функцию Split\n"
              "Посмотрим, что у тебя выйдет!";

List<string> crArray{};
crArray.Split(text, ' ');

for (string s : crArray)
    cout << s << endl;

// Попробуй
// использовать
// функцию
// Split
// Посмотрим,
// что
// у
// тебя
// выйдет!
// использование Split
```

При необходимости вы можете удалить ненужные символы или заменить значения.

```
string text = "Попробуй использовать функцию Split\n"
              "Посмотрим, что у тебя выйдет!";

List<string> strArray{}; List<char> crArray{};
initializer_list<char> symbol = { '!', ',', ' '};

crArray.ToCharArray(text);
crArray.DeleteByArgs(symbol).Change('П', 'п').Change('С', 'с');

text = "";
for (char s : crArray) // переписываем видоизмененную строку
    text += s;

strArray.Split(text, ' '); // разделяем на подстроки

for (string s : strArray)
    cout << s << endl;
// изменение
```

*изначальной строки и разбиение ее на строки*

## Tostring

Преобразует массив в строку. Не принимает никаких параметров.

## 3. Методы класса (манипуляция массивом)

Методы класса List позволяют изменять структурность массива, то есть изменять его элементы и размерность.

### Sort, Converse

Сортировка массива является довольно частым занятием. Особенно, когда нужно упорядочить большие массивы данных или найти лучшие и худшие показатели.

Для сортировки можно использовать метод Sort, а также связать этот метод с Converse, чтобы достичь сортировки элементов по убыванию, в случае необходимости.

```
_CONST_ = {  
    DOTTY,  
    SOLDNER,  
    M,  
    _GOLD_R_,  
    _SQRT_5_,  
    LPS,  
    TREFETENA  
};  
  
for (auto CNST : _CONST_.Sort())  
{  
    cout << CNST << " - " << GET_VARIABLE_NAME(CNST) << endl;  
}  
// сортировка
```

констант из директивы `_USE_CONSTANT_` компонента `Geometry`

```
_CONST_.Sort().Converse();  
for (int INDEX = _CONST_.First(); INDEX < _CONST_.Last();)  
{  
    const type_info& Type = typeid(_CONST_[INDEX]);  
    cout << _CONST_[INDEX] << " - " << Type.name() << endl;  
    ++INDEX;  
}  
// применение
```

сортировки на убывание значений констант из предыдущего примера

### Inversion

Изменяет значения массива на противоположные. Не работает с типом `string` (что естественно).

```
List<double> CR{ 4,-1,4,1.3 };  
for (auto S : CR.Inversion())  
    cout << S << " ";  
// инверсия double массива
```

### RandInt, RandDouble

Методы заполняют массив рандомными значениями в диапазоне, указанным пользователем. `RandInt` принимает параметры целочисленного `int` (начало, конец диапазона). `RandDouble` соответственно принимает вещественные аргументы типа `double`.

```
List<int> Array = { 0,1,0,1,4 };  
Array.RandInt(0, 9);  
for (int A : Array)  
    cout << A << " ";  
// заполнение заполненного массива
```

```

List<double> arr{};
for (int i = 0; i < 5; i++)
{
    arr.Add(RandDouble(0.0, 1.0));
    cout << arr[i] << " ";
}
// заполнение пустого массива

```

### Fill, FillPart, Change

Позволяют «перерисовывать» элементы массива. Fill заменяет все значения массива (например, можно использовать, когда нужно заполнить массив нулями, как память при заполнении потоков). FillPart позволяет изменять часть массива, исходя из диапазона, который указывает пользователь. Change – аналог метода Replace, который выборочно изменяет одно значение на другое.

```

ConsoleSetting console{
    console.Title("Documentation_TEST")
        .BackgroundColor("758180")
        .Resize(true)
};

List<POINT> sizeBOX{};

while (console.GetConsoleSize_X() > 10 && console.GetConsoleSize_Y() > 5)
{
    POINT point;
    point.x = console.GetConsoleSize_X();
    point.y = console.GetConsoleSize_Y();
    sizeBOX.Add(point);

    if (sizeBOX.Count() > 100)
        sizeBOX.Fill({0,0}); // пример Fill (заполнение нулями)

    Sleep(1000);
}

```

// заполнение массива нулями, если его размерность превысила определенный лимит

```

List<unsigned int> arr{10,2,4,1,5};
arr.FillPart(0, 3, arr.Last()); // пример заполнения части массива

```

FillPart в качестве параметров принимает: значение, на которое меняются элементы, начало – конец диапазона. При этом второй и третий параметр отображают элементы массива так, как мы видим (то есть отсчет начала массива начинается с 1, а не с 0, как принято в программировании).

Change позволяет изменять один элементы на другие. Например, если у нас имеется строка «привет», разбитая посимвольно, то при значении LIST.Change(e, E); строка станет «привЕт».

### Union, Add, AddFirst

Union – данный метод позволяет объединять несколько массивов в один. Но важно учитывать, к какому именно массиву вы прикрепляете значения. К примеру:

```
List<double> A{ 0,0 };
List<double> B({ 1.4, 1 });
List<double> C = { 4.3, 0 };

A.Union(B).Union(C);

for (auto s : A)
    cout << s << " "; // выведет все три массива вместе

cout << "\n";

for (auto s : B)
    cout << s << " "; // выведет только B, поскольку объединен был только A

cout << "\n";

for (auto s : C.Union(A))
    cout << s << " "; // выведет C + три массива
```

Последний вывод С массива будет содержать в себе массивы С, объединение всех массивов (поскольку А – это объединение трех массивов).

Add – с помощью данного метода можно добавлять новые элементы к массиву. Принимает лишь один параметр – значение, которое будет добавлено. Это значение также должно совпадать с типом массива, к которому это значение прикрепляется.

Когда Add добавляет элементы к концу массива, AddFirst делает те же действия, но с начала массива. Если массив был 0,0, то при использовании AddFirst(1) результат будет 1,0,0.

### **RemoveFirst, RemoveLast**

Аналогичны по принципу действия группе Add, но не имеют принимаемых параметров и перегрузок. Удаляют элементы либо с начала, либо с конца массива

### **Copy**

Копирует массив и добавляет все его элементы в этот же массив (процесс дупликации). Схож с Union, когда мы прописывали конструкции по типу A.Union(A), но не требует аргументов в качестве параметра. А также имеет перегрузку, в которой можно указывать количество повторений дублирования.

```
void Duplicate(List<string> arr) {
    arr.Copy();
    Sleep(10000);
} // дублирование массива каждые 10 секунд
```

```

List<size_t> MyArray = { 0 };
MyArray.Copy(10);
// дублирование 0 десять раз
// заполнение массива нулями
// дублирование 10 раз (итог: массив на 11 нулей)

```

### NewList

Дает возможность перезаписать существующий массив на другой такого же типа.

Поддерживает две вариации записи:

```

List<double> A = { 0,1,-3 };
List<double> B = { -4, 1, 0, 3 };
A.NewList(B).Sort();
A.NewList({ -4,1,0,3 }).Sort(); // вывод A: -4 0 1 3 (в обоих случаях)

```

### DeleteByIndex, DeleteByArgs, CutStart, CutEnd

Методы, предназначенные для удаления элементов массива (Delete группа) и его обрезания (Cut группа).

DeleteByIndex удаляет символы согласно индексу элемента в массиве. Также можно удалять группу индексов с помощью initialized\_list аргумента. DeleteByArgs выполняет ту же функцию по удалению, но вместо индексов использует значения элемента. Имеет такие же перегрузки, как и DeleteByIndex.

**CutStart и CutEnd отвечают за обрезку массива.** В качестве параметра принимают длину сокращения (длина в Cut в отличии от Index отличается началом отсчета с 1, а не с 0)

```

string text = "Запись для демонстрации Delete and Cut";
List<char> crArray{};
crArray.ToCharArray(text);

crArray.DeleteByArgs(' ') // удаление пробелов
    .DeleteByArgs({ 'e', 'a', 'и' }) // удаление гласных английских
    .DeleteByIndex(0, 5) // удаление первого слова
    .CutEnd(7); // удаление английских символов

// ИТОГ: для демонстрации

```

работа с Delete and Cut

### Unique

Позволяет удалить все повторяющиеся элементы массива. В своей основе используется обработка контейнера set.

```

int X(5), Y(5);
List<double> arr; double input = 666;
ConsoleSetting con = { con.ChangeEncode() };

while (input != 0)
{
    cin >> input;
    arr.Add(input);
    con.PositionCursor(X, Y += 1);
}

for (double s : arr.Unique())
    cout << s << ' ';

```

*// пример использования Unique*

ПРИМЕЧАНИЕ: автоматически сортирует все неповторяющиеся элементы в порядке убывания. Не работает с классовым типом String, поскольку происходит ошибка в файле type\_traits, поскольку не существует операнда под тип String.

### Действия над массивами.

Ранее были показаны некоторые действия над массивами. В частности – умножение всего массива на аргумент того же типа, что и сам List массив. Сейчас мы рассмотрим стандартные действия не в связке массив + аргумент, а в связке массив + массив.

```

List<int> A = { 3,1,2 };
List<int> B = { 0,23,0,0,10 };
A += B; // 3 24 2 0 10
A = A - B * A; // 3 -528 2 0 -90
for (auto s : A)
    cout << s << " ";

```

*// операции над массивами*

```

List<int> A = { 3,1,2 };
List<int> B = { 0,23,0,0,10 };
A.Addition(B); // 3 24 2 0 10
A = A.Substract(B * A); // 3 -528 2 0 -90

A = A.Substract(B).Multiplication(A);
// ответ будет иной, поскольку операции выполняются последовательно
|           // независимо от операций

for (auto s : A)
    cout << s << " ";

```

*// операции над массивом*

Следует отметить, что при использовании методов – массив будет изменяться последовательно, независимо от типа оператора.

## Компонент FunctionalLIB\_String

Компонент для работы с типом данных string. Стандарт C++ поддерживает достаточное количество методов для работы со строками. Однако, в данном компоненте будут собраны уникальные методы, которых нет в поддерживающем стандарте.

Многие методы основаны уже на существующих методах пространства <string>, а также на методах для массивов данных. Сама структура класса String и использование этого класса не сильно отличается от List класса пространства FunctionalLIB\_Array.

### Подключение и инициализация класса String:

```
#include "FPFC.h"
using namespace FunctionalLIB_String; // подключение .h файла и пространства имен
```

```
using namespace FunctionalLIB_String;

String str = "пример инициализации переменной класса";

String str2("EXAMPLE 2");

string text = "text for 3";
String str3 = text.c_str();

FunctionalLIB_String::String str4; // инициализации
```

переменной класса String

В данном компоненте собраны различные методы для работы со строками. Включая методы преобразования строк или получения данных из строки. Условно данный компонент разделяется на следующие части:

- стандартные операторы и методы класса;
- функции класса;
- методы класса (манипуляция со строками).

### 1. Стандартные операторы и методы класса

Как и в классе List – класс String поддерживает несколько операторов для более эффективной работы со строками. Класс String поддерживает небольшое количество операторов, которые позволяют вводить / выводить информацию, складывать и индексировать строки.

Пример использования некоторых операторов библиотеки показаны ниже:

```

String str = "Пример использования операторов";
str += "!!!!"; // пример оператора +=
String str2("\nНовая строка");
str = str + str2; // пример оператора + и =
cout << str << endl; // пример << оператора
str = str[2]; // пример индексации
str.Print();
// работа операторов класса

```

### *String*

Также существуют операторы `>>` и `<<`, которые не показаны напрямую пользователю при использовании переменной класса. Благодаря данным операторам возможен ввод и вывод данных с помощью потоков `cin` и `cout`.

```

String str;
str = 120 + 3, str += " ", str.operator+=(120);

// ВЫВОД: 123 120

cout << str;
// целые числа в String

```

Класс `String` благодаря перегрузкам операторов поддерживает и работу чисел. Вы в автономном режиме можете конвертировать любые типы данных в строку. Из примера выше: мы складываем 120 и 3, а затем приравниваем к полученным значениям пробел и число 120. В итоге мы получим строку: 120 123.

**ПРИМЕЧАНИЕ:** осторожнее пользуйтесь с типами данных. Вы не всегда сможете получить нужный результат, особенно при попытке сложить неподходящие типы или использовать не так операнды.

**Урезание строки можно осуществить посредством оператора `-`,** который имеет множество перегрузок, которые позволяют урезать строку, удалять символы или куски текста.

```

String str = "Урезание строки";
String S = str - "рез" - 7;
// -7 - урезает с конца на семь символов
// принимает параметры как string так и String
// ОТВЕТ: Уание

string s = (str / 2).ToString();
// урезание строки в половину
// при делении на 1 или 0 - строка будет пустой
// ОТВЕТ: Урезание
cout << s;
// операторы - и /

```

**ВАЖНО!** Переменная `str` видоизменяется уже при первом урезании. Поэтому ответ может различаться в зависимости от применения разных методов до применения операторов.

В качестве аналога, **для вывода, можно использовать метод Print**, который выводит на экран строку. Данный метод также имеет несколько перегрузок. Может принимать в качестве параметра аргументы типа string и String. В случае этих перегрузок будет выводиться две строки сразу. Таким образом можно переходить на новые строки без использования дополнительных потоков.

### Length

Аналог length для string. Позволяет узнать количество символов в строке. Можно использовать как в циклах, так и на других переменных типа int (поскольку Length возвращает именно целочисленный тип).

```
String str = "Пример использования LENGTH";
for (int i = 0; i < str.Length(); i++)
    str.Print("\n"); // вывод с переходом на новую строку

int len = str.operator+=(newLength).Length(); // получение длины

cout << str.Length() << " " << len;
```

### ToString

Метод, который позволяет конвертировать строку типа String в стандартную string строку. **Данный метод может быть полезен**, в случае, **когда** методов класса String не хватает и **нужно прибегнуть к стандарту C++**.

Содержит несколько перегрузок за счет чего может конвертировать не только класс String, но и различные переменные или списки.

```
String str = "Пример использования ToString";
string str2 = str.ToString();

cout << str2 << endl; // вывод string, а не String

String ConvertToString;
ConvertToString.ToString(100); // конвертирование в string аргумента

String ConvertToList;
ConvertToList.ToString({'3','1','2'}); // преобразование списка

List<float> flArray = { 0, 1.3, 3.2f, TREFETENA };
ConvertToList.ToString(flArray); // конвертирование List массива

cout << ConvertToList << endl;

// ВАЖНО! при использовании ToString для нескольких аргументов (массивы и списки)
//        данные записываются через пробел, например 3 1 2, а не 312
```

Как было указано в блоке с комментариями – **при использовании аргументов из initializer\_list или List – данные записываются через пробел**. Так было сделано с целью разграничения значений массива. В случае, **если вам не нужен пробел между значениями массива можете воспользоваться**:

```

ConvertToList.RemoveSymbol(' ');
ConvertToList.Replace(' ', NULL); // убирают пробелы между значениями
massiva

```

**ПРИМЕЧАНИЕ! Пробелы в перегрузках БЫЛИ УДАЛЕНЫ. Теперь все методы ToString без пробелов добавляют значения к строке.**

Вы также можете конвертировать значения строки в класс String. Для этого можете воспользоваться функцией ToTypedPL\_String.

```

String textA = "23";
string textB = textA.ToString(); // конвертация в str

textA = textB; // присваивание к уже объявленному классу
String textC = ToTypedPL_String(textB); // конвертация в класс str
String textD = textB.c_str(); // конвертация в класс str

```

На скриншоте выше показано несколько способов конвертации строки в класс String. В зависимости от ситуации и удобства вы можете воспользоваться тем или иным методом.

### Clear

Аналог clear из стандарта C++. Как следует из названия – очищает строку.

## 2. Функции класса

### GetElement, GetFirstElement, GetLastElement

Функции, возвращающие тип char (поскольку функции завязаны на индексации, было бы неправильно выводить их в string или в другом типе). **Позволяют получить элементы из строки по их индексу.**

GetFirstElement – аналог front и выводит первый индекс (нулевой) из строки.

GetLastElement – аналог back и вывод последний индекс из строки.

GetElement – аналог at, но без возможности изменять элементы, вытягиваемые из метода (*что имело в виду будет показано на скриншоте ниже*).

```

String str = "Пример использования GetElement Group";
char index = str.GetFirstElement(); // п
index = str.GetLastElement(); // п
index = str.GetElement(6); // ' '
// использование

```

*GetElement*

```

string str2 = "Пример использования GetElement Group";
str2.at(6) = 'Z';
str2.getElement(6) = 'X'; // ошибка
// почему at

```

*функциональней методы GetElement (что имелось в виду в описании выше)*

### Search, SearchCount

Данные функции применяются для поиска значений в строке. Группа перегрузок Search возвращает булево значение true или false в зависимости от совпадений аргумента со строкой. SearchCount возвращает целочисленное число – а именно число совпадений по тексту. Даная функция может быть полезна при подсчете символов или слов.

**Search, как и SearchCount имеет следующие перегрузки:**

- char;
- string;
- initializer\_list<char>;
- List<char>.

```
String str = "Пример использования Search Group";
List<int> flag(4);

flag[0] = str.Search({ 'Z', ' ', 'x' }); // true
flag[1] = str.CutEnd(6).Search('G'); // false
flag[2] = str.Replace('и', 'И').Add("LIST").Search("Использования"); // true

List<char> a = { '0', '2', '1' };
flag[3] = str.Search(a); //false
```

//

*использование массива флагов для Search разных перегрузок*

```
String str = "Пример использования Search Group";

cout << Max(str.SearchCount('и'),
            str.SearchCount({ ' ', 'S' })),
            str.SearchCount("ова"));
            // вывод 4 (второй вариант)
```

// *использование*

*SearchCount в методе Max*

**Equals, EqualsMax, EqualsMin**

Функции, которые возвращают true или false. Сравнивают строки между собой. Если строки равны – 1, если нет – 0.

```
String strONE = "text Equals";
String strTWO = strONE;

bool a = strONE.Equals(strTWO); // true
bool b = strONE.Equals(strTWO.AddSymbol('0')); // false
bool c = strONE.Replace('x', 'X').Equals(strTWO); // false

cout << a << " " << b << " " << c;
```

// *использование*

*сравнения Equals*

Функция Equals сравнивает не строки не только по длине, но и по символам самой строки. Например, bool с будет false потому, что x в первой строке будет отличаться от второй.

**Функции EqualsMax и EqualsMin сравнивают строку только по количеству символов.** Предполагается, что если первая строка Max (*то есть, если str.EqualsMax(str2)*) – то вывод будет 1, иначе 0. И также наоборот.

```
String strONE = "text Equals";
String strTWO = "text Equals...";
string str;

bool a = strONE.EqualsMax(strTWO); // false (сравнение String)
bool b = strTWO.EqualsMin(str); // false (сравнение с string)
// использование
```

### EqualsMax и EqualsMin

Группа Equals поддерживает перегрузки с типами данных `string` и собственно класса `String`. Таким образом мы можем сравнивать строки как классовые, так и стандартные.

### ToCharArray

Функция, которая возвращает `List<char>` значение. То есть из строки получается массив (**строка преобразуется в массив символов**). Подобная функция может пригодится в случаях, когда инструментов `String` не хватает и нужно воспользоваться массивом (например, сортировка или центральный символ).

```
String str = "Разбиение строки на символы";
List<char> charArray = str.ToCharArray().Sort();
for (auto s : charArray)      // конвертирование в массив для сортировки
    cout << s;

cout << endl;
cout << charArray.Median() << "\n" <<
    charArray.ToString() << "\n";
    // конвертирование в строку

String str2 = charArray.ToString().c_str();
// преобразование в класс String

str2.Print("\n");
// использование ToCharArray
```

В примере выше мы сначала конвертируем строку `String` в массив символов `List<char>`, а также сортируем этот массив в порядке возрастания. Затем мы выводим центральную позицию отсортированного массива (поскольку `Median` нет в классе `String`) и конвертируем массив обратно в строку.

**ПРИМЕЧАНИЕ.** Несмотря на то, что мы конвертируем массив в строку (то есть несколько параметров) – пробелы между значениями массива будут отсутствовать. Метод `ToString` осуществляет пробелы исключительно в перегрузках с `initializer_list` или `List` в классе `String`.

### Split

Предназначен для разбиения строки на подстроки. Возвращает значение `List<string>`

```

String str = "Разбиение строки на подстроки";
List<string> splitArray;

splitArray = str.Split(' ');
    // часто применяют для отсеивания ' ', '!' и подобного

typedef int _INTEGER_;
_INTEGER_ word = splitArray.Count();
while (word != 0)
{
    String Str = splitArray[word - 1].c_str();
    Str.Print("\n"); // обратный вывод слов строки
    word--;
}

// подстроки
// на
// строки
// Разбиение

```

// *Split в классе*

## *String*

### Конвертация в vector и обратно.

Как и в классе *String*, для *List* также был реализован метод, который может преобразовать *List* массив в *vector* (поскольку он является наиболее популярным в использовании). Это может пригодится в моментах, когда нужно воспользоваться стандартными функциями C++. Для конвертации используются методы *ToVector*, и *ToTypedPL\_List* для конвертации контейнеров в *List*.

## 3. Методы класса (манипуляции строками)

### NewString

Переназначает текущее значение строки на новое. Равнозначно записи *string = «Новое значение»*.

```

String str = "new str";

str = "новая строка";
str.NewString("новая строка"); // переназначение строки

```

### Reverse и Replace

*Reverse* – аналог стандартного метода со строками в C++, который позволяет переворачивать строку в обратную сторону. *Replace* – аналог уже функции *Change* из класса *List* библиотеки *typedPL* (в то же время есть подобный метод и в обычном *string*, но *Replace* почти 100% копирует именно *Change*).

```

String str = "переверни меня, семпай";
str.Reverse().Replace('e', 'E').AddSymbol('!');
cout << str; // переворот и замена символов

```

строки

### SwapString

Аналог Swap из `<string>`, принцип которого заключается в замене значений двух строк на противоположные. То есть – строка 1 станет строкой 2. А строка 2 станет строкой 1.

```

String str1, str2;
puts("Введите значения двух строк:");
cin >> str1 >> str2;

cout << str1.SwapString(str2) << "\n" << str2; // замена строк

```

### CutStart, CutEnd, CutRange

Обрезают строку по заданным параметрам. Вы можете обрезать строку с начала или с конца, указав лишь длину удаления. Либо вы можете указать диапазон вырезки, используя CutRange.

```

String str("РЕЖЬ, РЕЖЬ! РЕЗНЯЯЯЯ");
str.CutEnd(3).CutStart(6).CutRange(1, 6);
cout << str; // РЕЗНЯ // использование обрезки

```

### Add, AddSymbol

Методы для добавления к строке новых символов. Add имеет ряд перегрузок, которые дают возможность прибавлять сразу несколько параметров, когда AddSymbol добавляет лишь 1 символ за одно применение метода.

```

string str = "Add??";
String STR = str.c_str();

STR.Add("!!").Add({ '!' }).AddSymbol('!');
cout << STR; // добавление «!!!» к строке

```

### Аналоги Add:

```

STR.ToString({ '!', '!', '!' });
// лучше не использовать char при конвертировании

string standart_string = STR.ToString();
standart_string.push_back('!');
STR = standart_string.c_str();

STR += "!!!";

STR.operator+=( "!!!");
STR = STR + "!!!"; // аналоги Add

```

## **Remove, RemoveSymbol, Remove\_FirstSymbol, Remove\_LastSymbol**

Предназначены для удаления символов.

**RemoveSymbol** – удаляет конкретный символ и принимает тип char.

**Remove\_FirstSymbol** – удаляет первый символ строки.

**Remove\_LastSymbol** – удаляет последний символ строки.

Метод **Remove** содержит несколько перегрузок, среди которых: символ, список, List массив, удаление по тексту.

```
size_t size; String str;
ThisBold("Введите вашу строку...\n");
cin >> str;
ThisBold("Сколько символов вы хотите удалить?\n");
cin >> size;

for (size_t symbol = 0; symbol < size;)
{
    str.Remove_LastSymbol().Remove_FirstSymbol();
    symbol++;
}

cout << str; // сокращение строки с ее
```

начала и конца

Более быстрый способ:

```
size_t size; String str;
ThisBold("Введите вашу строку...\n");
cin >> str;
ThisBold("Сколько символов вы хотите удалить?\n");
cin >> size;

cout << str.CutStart(size).CutEnd(size); // аналог предыдущего
```

примера

## **RemoveA\_z, RemoveA\_я, RemoveNum, RemoveAnother**

Методы, предлагающие инструменты для удаления категорий символов (русские, английские, числа или спец символы). Может быть более полезен, чем тот же Remove с поддерживаемым списком, поскольку данная группа методов удаляет определенные символы без необходимости добавления аргументов к конструктору.

Если вам необходимо убрать исключительно числа из строки – то лучшим вариантом будет использовать RemoveNum, нежели Remove ({0,1,2,3,4,5,6,7,8,9}).

```

String regular("!%STR_404 = ошибка");

regular.RemoveAnother().Print("\n"); // STR404 ошибка

regular.RemoveA_z().RemoveA_я().Print("\n"); // !_404 = ?

regular.RemoveNum().Print("\n"); // !_STR_ = ошибка?
                                            // удаление групп

```

### **ExceptA\_z, ExceptA\_я, ExceptNum**

Данная группа методов убирает все символы, кроме указанной группы. К примеру, если вы будете использовать ExceptNum – из строки будут удалены все символы, кроме чисел. Также и с другими методами из данной группы.

```

String regular("!%STR_404 = ошибка");
cout << regular.ExceptA_я() << endl;
                                            // ошибка
                                            // удаление всех групп, КРОМЕ

```

### **Regular\_Remove, Regular\_Replace**

Принимают аргументы типа string. Они позволяют прописать собственное выражение для удаления или замены символов. Таким образом, данные методы могут быть более подходящим, в некоторых ситуациях, способом записи для форматирования строки.

#### **Рассмотрим импровизированную задачу.**

Имеется предложение и необходимо изменить тон его прочтения.

```

String regular("Как я могу... узнать ваш адрес?");
cout << regular.Regular_Remove("[.]")
    .Regular_Replace("[?]", "!?");
                                            // использование регулярных

```

выражений (ответ: Как я могу узнать ваши адрес!?)

В данном случае мы не можем использовать какой-либо метод для удаления групп (*но в конкретно в этом примере можно использовать Remove (<.›)*). Поэтому мы можем воспользоваться регулярным выражением. Обычно, подобные выражения заключаются в квадратные скобки. Таким образом они помечаются, как выражение. В нашем случае мы удаляем все точки, чтобы читать предложение без запинки.

**Regular\_Replace также может быть более эффективен** в некоторых ситуациях, поскольку регулярные выражения принимают в качестве параметра строки, а не символы, как это делает Replace из данного класса. Таким образом, мы изменяем «?» на «!», тем самым произнося предложение более эмоционально. (*В принципе все можно решить и с помощью Add символа в конец строки, но через выражения, однозначно, быстрее и проще*).

## Компонент FunctionalLIB\_File

Компонент предназначен для работы с файлами в C++. Благодаря данному компоненту вы сможете хранить, перезаписывать информацию в файлах. Часто это необходимо, когда происходит подгрузка данных извне. Помимо этого, компонент предлагает работу с каталогами. Вы можете создавать файлы, удалять, открывать для редактирования, или получать информацию о состоянии файлов.

### 1. Чтение и запись файлов

FunctionalLIB\_File предлагает несколько методов для записи информации в файлы и чтение данных из них. Названия методов для чтения и записи по своим названиям схожи с методами из System.IO в C#.

#### ReadAllText, ReadAllLines

**Чтение данных из файлов осуществляется за счет функций ReadAllText и ReadAllLines.** Первая функция возвращает строковой тип данных (то есть все данные записываются в одной переменной). ReadAllLines считывает файл построчно и возвращает значение массива.

```
File fLCOPY = "C:\\\\Users\\\\Vandi\\\\OneDrive\\\\Рабочий стол\\\\test.txt";
File shortFLCOPY = "file.txt"; // короткая запись, если файл
                                // находится внутри проекта

cout << fLCOPY.ReadAllText();

List<string> arr = fLCOPY.ReadAllLines();
for (string A : arr)
    cout << A << "\\n"; // использование
```

методов для чтения файла

Важно отметить, что **обе функции возвращают тип string**, поскольку чтение происходит с файла, в котором хранится текст (не важно какой вид он имеет).

Вместо List вы также можете использовать стандартный vector для передачи значений. Поскольку нельзя осуществлять перегрузки только по входному значению, то для vector массива существует отдельная функция ReadAllLines\_Vector.

```
vector<string> arr = fLCOPY.ReadAllLines_Vector();
for (string A : arr)
    cout << A << "\\n";

List<string> arr_list = fLCOPY.ReadAllLines();
for (string A : arr_list.Sort().Converse())
    cout << A << "\\n"; // использование методов
```

считывания по линиям для разных типов массивов

## **WriteAllText, WriteAllLines, AppendAllText, AppendAllLines**

Для записи данных есть методы группы **Write** и **Append**. Первая группа методов записывает данные в файл, но перезаписывает информацию в нем, которая была до использования метода. Вторая же группа сохраняет и предыдущие записи файла.

```
File fLCOPY = "C:\\\\Users\\\\Vandi\\\\OneDrive\\\\Рабочий стол\\\\test.txt";
fLCOPY.WriteAllText("ЗАПИСЬ ТЕКСТА В ФАЙЛ .TXT"); // запись в одну строку

List<double> array({ 0,0,0,0.1 });
fLCOPY.WriteAllLines(array); // запись построчно массива

string a;
for (int i = 0; i < array.Count(); i++)
    a += array[i] + ' '; // если использовать методы File здесь,
    |                   //то в результате в файле будет лишь последний символ, поскольку AllText перезаписывает файл

fLCOPY.WriteAllText(a); // запись массива в одну строку через пробел
//
```

*использование Write методов для записи данных в текстовый файл*

Третий комментарий говорит о том, что если использовать метод `WriteAllText` в цикле, то в итоге будет записан в файл только последний индекс массива. Поэтому мы собираем все значения в одну строку, после чего выводим ее отдельно от массива.

Упрощение третьего примера можно достичь за счет использования `Append` методов. Можно воспользоваться одним из двух вариантов, предложенных ниже:

```
for (int i = array.First(); i < array.Last(); i++)
    fLCOPY.AppendAllText(array[i], "\\n"); // запись текста без перезаписи

fLCOPY.AppendAllLines(array); // альтернатива в случае массива
//
```

*использование Append методов для записи информации в текстовый файл*

Следует отметить, что методы типа `AllText` имеют перегрузку. Они могут принимать как единичный параметр в качестве записи, так и параметр записи совместно с форматом записи. То есть, мы можем использовать «`\n`» для перехода на новую строку или «`_`» для отступа между записываемыми значениями. (Пример использования подобной перегрузки показан выше).

Вы можете применять методы компонента совместно со структурами или объединениями данных. Вот один из примеров использования `Append` метода для записи массива объединений.

```
List<pair<String, double>> pairARRAY = { {"INDEX", 0}, {"COEFFICIENT", 0.21}, {"TIMER", 1000} };
pairARRAY.AddFirst({ "UTILITY", 198 });

for (auto& s : pairARRAY)
    fLCOPY.AppendAllText(s.first + " " + s.second + "\\n"); // запись построчно массива с несколькими параметрами
//
```

*pair и методы компонента*

Если вы не хотите использовать пространство `FunctionalLIB_Array`, то вы все равно можете воспользоваться методами `Write` и `Append`. **Данные группы поддерживают работу с векторами и статическими массивами** (поддерживают лишь такие контейнеры, поскольку они наиболее распространены).

```

vector<LPCWSTR> coor = { L"type", L"and", L"test" };
fLCOPY.WriteAllLines(coor);

float arguments[] = { 4.3f, 3.96f, 3.14f };
for (float args : arguments)
    fLCOPY.AppendAllText(args, " ");

```

// использование методов

*Write и Append без класса List*

## 2. Действия над файлами (не работает на коротких ссылках)

### FileOpen

Открывает файл по ссылке. Поскольку метод реализован через ShellExecute, имеется возможность запускать не только текстовые файлы, но и приложения по типу телеграмма и прочих. Открытие ссылок из интернета также осуществимо с помощью данного метода.

```

File fl = "C:\\\\Users\\\\Vandi\\\\OneDrive\\\\Рабочий стол\\\\test.txt";
fl.FileOpen(); // запуск txt файла

fl = "D:\\\\К видео\\\\Photoshop\\\\Adobe Photoshop 2022\\\\Photoshop.exe";
fl.FileOpen(); // запуск фотошопа
|           // потребуется время для запуска

fl = "https://habr.com/ru/companies/playrix/articles/465181/";
fl.FileOpen(); // открытие ссылки из интернета

```

// пример

*использования FileOpen*

### FileCreate, FileRemove

Отвечают за создание и удаление файлов. Не принимаю в качестве аргументов никаких параметров. То есть, данные методы работают со строкой переменной класса File. Поэтому если вам нужно создать один файл, а удалить другой – то необходимо использовать несколько экземпляров класса.

**Вы можете создавать с помощью FileCreate файлы .txt, .docx, .pdf, .xlsx и прочие форматы файлов.** С удалением файлов такая же ситуация.

**При необходимости, возможно реализовать файл своего формата «MyFormat.frm»** или любое другое название по вашему желанию.

### FileMove

Метод для перемещения файлов на диске.

```

File fl = "C:\\\\Users\\\\Vandi\\\\OneDrive\\\\Рабочий стол\\\\MyFormat.frm";
fl.FileCreate();

string name = fl.GetFileName();
fl.FileMove("D:\\\\Win10Files\\\\" + name);

```

*// FileMove для переноса файла с диска C на D*

### FileRename

Переименовывает файл. При этом важно учитывать, чтобы оба пути к файлу были абсолютны. Иначе метод может работать иначе и проигнорировать переименование.

### 3. Получение информации о файлах (не работает на коротких ссылках)

Получение размеров файла с помощью **GetBitSize**, **GetByteSize** и так далее.

```
File fFSIZE = "C:\\\\Users\\\\Vandi\\\\OneDrive\\\\Рабочий стол\\\\Documentation.docx";  
  
double MB = fFSIZE.GetByteSize() / 1024 / 1024; // перевод из байтов в МБ  
cout << MB << "\\n";  
  
double MB2 = fFSIZE.GetMBSIZE(); // получение сразу МБ  
cout << MB2 << "\\n"; // примеры
```

использования *Get* функция для получения размеров файла

```
File fFSIZE = "C:\\\\Users\\\\Vandi\\\\OneDrive\\\\Рабочий стол\\\\Documentation.docx";  
  
wstring wsize = L"C:\\\\Users\\\\Vandi\\\\OneDrive\\\\Рабочий стол\\\\Documentation.docx";  
LPCWSTR lpc = wsize.c_str();  
  
if (fFSIZE.GetMBSIZE() > 20)  
    puts("файл слишком большой!");  
else  
    ShellExecute(NULL, L"open", lpc, NULL, NULL, SW_SHOW); // проверка на
```

размеры файла

#### **GetFileName, GetFilePath, GetFileExtension, GetFileDepth**

Функции, которые позволяют получить имя файла, его расширение, путь до него и глубину его местоположения (то есть сколько папок предшествует файлу). Кроме последнего, всего остальные возвращают тип данных *string*.

```
File shortFlCOPY = "C:\\\\Users\\\\Vandi\\\\OneDrive\\\\Рабочий стол\\\\InAcademy\\\\x64\\\\Debug\\\\file.txt";  
  
ThisRGB(shortFlCOPY.GetFilePath() + "\\n" +  
        shortFlCOPY.GetFileName() + "\\n" +  
        shortFlCOPY.GetFileExtension() + "\\n" +  
        to_string(shortFlCOPY.GetFileDepth()) + "\\n"  
        , 170, 170, 250);
```

// получение информации о пути файла

**Вид вывода:**

```
C:\\\\Users\\\\Vandi\\\\OneDrive\\\\Рабочий стол\\\\InAcademy\\\\x64\\\\Debug  
file.txt  
txt  
8
```

Исходя из результатов этих функций можно определять, допустим, длину наименований файлов. Таким образом возможно осуществить запрет на обработку файлов слишком с большим наименованием или по определенному расширению.

```
File FlCOPY = "C:\\\\Users\\\\Vandi\\\\OneDrive\\\\Рабочий стол\\\\InAcademy\\\\x64\\\\Debug\\\\file.txt";  
String extension = ToTypedPL_String(FlCOPY.GetFileExtension());  
  
if (extension == "txt") // проверка на тип файла  
    cerr << "Невозможно обработать текстовый файл";  
  
int size = FlCOPY.GetFileName().size();  
  
size > 20 ? cerr << "Ошибка наименования!" : cout << extension; // проверка на размер имени
```

// использование функций для дальнейших проверок по условию

## Компонент FunctionalLIB\_CustomConsole

Этот компонент специализируется на внешнем виде консольного приложения на языке C++. Благодаря CustomConsole можно изменять как само окно приложения, так и его взаимодействовать с его элементами: например, изменять параметры текста, пространства и курсора.

Основные методы и функции данного компонента базируются на классе ConsoleSetting. Поэтому условно можно разделить функционал на две большие группы, а затем содержимое класса на другие подгруппы.

**Таким образом, компонент содержит:**

- элементы вне класса;
- элементы ConsoleSetting.

Данный компонент подключается через FPFC. Чтобы иметь доступ к элементам необходимо добавить пространство используемых имен «using namespace FunctionalLIB\_CustomConsole».

```
#include "FPFC.h"  
using namespace FunctionalLIB_CustomConsole;
```

Многие методы реализованы с помощью команд-модификаторов из stdlib.h файла и на WinAPI. Стоит отметить, что **данные модификаторы влияют на весь текст в консольном приложении**. Поэтому нужно осторожно пользоваться подобными методами, поскольку **они могут перекрывать друг друга**.

### 1. Методы вне класса ConsoleSetting

Большинство методов данного класса связаны на единичном изменении текста, выводимого из консольного приложения. То есть, они основаны на изменении одного аргумента и последующем приведении консоли к нормальному состоянию.

В некотором аспекте данные методы похожи на puts, но без перехода строки на новую линию.

#### ThisBold, ThisNormal, ThisLine

Данные методы преобразуют текстовый фрагмент в один из стилей.

#### Пример использования:

```
ThisBold("Добро пожаловать!\n"  
        "Введите свое имя: ");  
  
char name[50];  
cin >> name;
```

### Для сравнения:

```
Добро пожаловать!
Введите свое имя: Вадим
```

Можно заметить, что текст отличается друг от друга яркостью и четкостью. Текст метода является наиболее ярким, чем стандартный текст.

Подсвечивать можно не только заранее известный текст. Для того, чтобы пользователь мог вводить жирный текст применимы аналогичные методы из `ConsoleSetting`.

**Ввод текста разного стиля показан на рисунке ниже:**

```
string text;
ConsoleSetting s{};

s.Bold(); // ввод жирного текста
cin >> text;

s.Underline(); // текст станет жирным с подчеркиванием
cin >> text;

s.Normal(); // преобразование жирного с подчеркиванием в обычный
cin >> text;
```

Важно отметить, что методы на подчеркивание текста являются дополнением. То есть они не перекрывают существующий стиль, а добавляют его.

Таким образом мы можем изменять глобальный стиль текста всей консоли, а не только для определенного текстового фрагмента.

### ThisRGB, ThisForegroundRGB

Данные методы позволяют придать тексту цвет по RGB системе, а также установить задний фон этого текста.

Принимают четыре параметра: преобразуемый аргумент, `red`, `green`, `blue`.

**Пример использования:**

```
typedef array<double, 5> MASSIV;

MASSIV arr = { 50.4, -10.4, 0.04, 12, 5.3f };

for (auto out : arr)
{
    ThisRGB(out,
        RandInt(0, 255),
        RandInt(0, 255),
        RandInt(0, 255));

    cout << " ";
}
```

В данном случае мы выводим double массив. Для каждого его элемента устанавливается свой уникальный цвет по средствам использования функции RandInt из FunctionalLIB\_Array в качестве параметров метода ThisRGB.

Вывод будет следующим:

```
50.4 -10.4 0.04 12 5.3
```

**Метод ThisForegroundRGB является неоднозначным**, поскольку ее применение нельзя назвать целесообразным и полезным. В- первую очередь, ограниченная область действия на определенный фрагмент текста. Во-вторых, данный метод способен изменять только задний фон, что означает, что сам текст будет плохо виден, что значительно снизит читабельность текста.

### Close и Clear

Как понятно из названия – методы для закрытия и очистки содержимого консоли. Первый может применяться в случаях, когда необходимо быстро выйти из программы. А вторая при наличии большого количества информации (К примеру, когда постоянно информация в консоли обновляется. В таких случаях будет неудобно читать всю консоль, а можно оставлять лишь актуальные данные).

## 2. Рамка консольного приложения (изменение свойств)

Данный блок основывается на методах, которые влияют на рамку (то самое место, где находится название проекта и которое позволяет передвигать консоль). Здесь собраны некоторые методы, которые изменяют ее вид, а также некоторые свойства.

Прежде, чем начать разбирать данный блок, следует сказать, что здесь отсутствуют методы и функции (вернее они есть, но в ином виде). **В качестве содержимого класса используются конструкторы класса. Данные конструкторы позволяют достичь цепочки методов**, что является более удобным способом записи (К примеру, в компоненте FunctionalLIB\_Array отсутствует данная возможность, поскольку конструкторы не взаимодействуют с элементами вне класса).

Для сравнения двух типов записи:

```
static void ARRAY() {  
    unsigned int massiv[] = { 0,1,1,0 };  
  
    Sort(massiv);  
    Inversion(massiv);  
    Converse(massiv); // использование трех функций для сортировки  
                      // инверсии и разворота одного массива  
    for (int index : massiv)  
        cout << index++ << "\n";  
}
```

```
// Запись без цепочки методов
```

```
static void SETTING(ConsoleSetting config) {  
    config.FontFamily("Times New Roman").Bold();  
    // пример цепочки методов  
  
    bool cfg = false;  
    config.Resize(false).  
        ScrollBarShow(cfg).  
        WindowPanelShow(0);  
    // пример цепочки методов с разными вариациями записи bool  
    // данную запись в столбец можно использовать для удобства чтения  
}
```

```
// Запись цепочкой методов
```

На первый взгляд разница может быть не совсем заметна. Ведь что в первом, что во втором варианте набирается комбинация методов. Однако, в первом случае приходится постоянно использовать один и тот же аргумент в качестве параметра. В случае конструкторов, данную операцию можно заменить лишь одним повторением. Да и второй способ имеет разные вариации записи, и выглядит более элегантно.

### WindowShow

Данный метод позволяет указывать – запускать консоль вместе с рамкой или без нее. Поэтому в качестве параметра принимается булева переменная.

Рамка может играть большую роль в использовании программы. В первую очередь она отображает название и иконку программы. Помимо этого, без нее не будет возможности закрыть приложение, или перетащить его по экрану (если не использовать программный способ).

### WindowPanelShow

Панель рамки – в данном компоненте под панелью понимаются три функциональные кнопки рамки: сворачивание, растягивание, закрытие консоли.



Этот метод позволяет скрывать данную панель. То есть, если флаг будет 0, то мы не сможем воспользоваться данными кнопками, но при этом по-прежнему сможем видеть название или перетаскивать приложение с места на место.

### WindowMinShow, WindowMaxShow, WindowCloseShow

Предполагается, что разработчику не всегда будет выгодно иметь в наличии возможность сворачивания или растягивания приложения (например, если отсутствует адаптация под различные размеры приложения). Поэтому для каждой кнопки WindowPanel имеются свои методы. Это позволит скрывать и отображать кнопки в зависимости от

поведения пользователя или использовать данные параметры в качестве изначального ограничения.

### Пример:

```
ifstream read("password.docx"); // чтение пароля с файла
string text;
getline(read, text);

ConsoleSetting console; // инициализация ConsoleSetting
console.WindowMaxShow(0).WindowMinShow(0);

string pass;

console.TextColor("e5c8f5").Bold(); // назначение цвета по HEX
puts("Для разблокировки функций приложения введите пароль...");

console.TextColor(203,150,231); // переназначение цвета в чуть темнее в RGB системе
cin >> pass;

if (pass == text) // включение функционала приложения при правильном пароле
    console.WindowMinShow(1).WindowMaxShow(1);
```

При запуске в рамке отсутствуют кнопки сворачивания и растягивания: . Для их разблокировки нужно ввести пароль. Если введенный пароль совпадает с тем, что находится в файле, то дополнительный функционал программы будет доступен:

– □ × .

### Icon

Позволяет устанавливать иконку для приложения. Не работает с форматами png, jpg, bmp и так далее. Для отображения иконки – изображение необходимо конвертировать в формат .ico, или же заранее скачать изображение данного формата.

```
ConsoleSetting con{};
con.Title("ИКОНА????").Icon("C:\\\\Users\\\\Vandi\\\\Downloads\\\\ico.ico");
con.Title("ИКОНА????").Icon("ico.ico"); // сокращенная запись
```

//

*сокращенная запись в случае, если иконка находится в папке проекта*

### ScrollBarShow

У рамки также есть такой элемент, как полоса прокрутки. Порой, есть необходимость ограничить ее видимости. Обычно это может быть связано с тем, что данный элемент мешает видимости или дизайну. Либо информации настолько мало, что нет необходимости в полосе прокрутки. Принимает ScrollBarShow такие же параметры, как и предыдущие методы – true или false.

### Resize

Важное свойство для любой рамки – возможность ее растягивать. Данное свойство в зависимости от ситуации также можно отключить. Обычно, Resize важен в графических

приложениях, нежели в консольном. Это связано с тем, что консольное приложение во многом используется не для потребителя. Поэтому параметры свойств консоли особо не требуют вмешательства. Но в качестве примера возможностей WinAPI было добавлено и это свойство (как и многие из ранее или в будущем описанных).

### FullScreen

Отображает приложение по ширине и высоте экрана. Не принимает никаких параметров. Может быть использован для первоначального отображения консоли (чтобы при запуске отображалось не маленькое окошко).

### PositionConsole

Для отображения консольного приложения в определенном положении на экране. Так можно отобразить программу в любом углу экрана. В качестве аргументов принимает два целочисленных значения (X и Y).

### Size и MaxSize

Позволяют изменять размеры окна. Может пригодиться при наличии шаблонной информации (что-то на подобие подгонки размеров под контент).

Size и MaxSize желательно использовать вместе друг с другом. Это связано с тем, что возможности записи Size зависят от значений в буфере консольного приложения. Например, **если значения пользователя больше, чем запись в буфере** – приложение не поменяет своего размера. Поэтому **для изменения значений в буфере – используется MaxSize**.

```
ConsoleSetting console{
    |
    console.Size(100,50), // не сработает
    |
};

// использование Size

ConsoleSetting console{
    |
    console.MaxValue(171,50).Size(170,45)
    |           // сначала вводим максимально допустимые значения
    |           // затем вводим размеры консоли
    |
};

//
```

*использование Size с MaxSize*

Стоит отметить, что многое зависит также от параметров экрана. Допустим, **если ввести значение больше, чем высота вашего экрана – консоль не поменяет размер** (при этом появится полоса прокрутки, которая будет отображать длину ваших данных). Также желательно, чтобы **максимальная запись была минимум на единицу больше действительной**. В противном случае отображение может быть некорректным.

## **GetConsoleSize\_X, GetConsoleSize\_Y**

Функции возвращают размеры консоли. В качестве аргументов не принимают никаких значений. Возвращают тип int.

```
while (true)
    if (console.GetConsoleSize_X() < 50)
        return 0;
    // ограничение растягивания консоли
```

## **Title**

Позволяет изменять отображаемое на рамке название программы.

### **Пример использования:**

```
ConsoleSetting console{
    console.Title("Рамка консольного приложения");

    ThisLine("...\n"
             "...\\t\\b" // использование разных escape последовательностей
             "...\\n"
            );

    console.Title(TitleChange().c_str());
    // переименование приложение

string TitleChange() {
    string tit; cin >> tit;
    return tit;
}
```

## **Измерение консоли в Pixel**

Установка размеров консоли через Size может быть немного некомфортным, поскольку параметры зависят от размеров в буфере. Помимо этого, размерность через Size задается посимвольно, что может привести к непредсказуемым результатам при использовании метода.

Измерение консоли в пикселях позволит более безопасно устанавливать размеры консоли. Для установки размеров в пикселях необходимо воспользоваться методом **SizePixel**, принимающий два параметра: X и Y в пикселях. Для того, чтобы получить размеры консоли в качестве числа – можете воспользоваться геттерами (Get функций).

## **3. Пространство консольного приложения**

Данный блок посвящен содержимому программы: текст, фон, и так далее.

### **Opacity**

Настраивает прозрачность приложения. Принимает параметр типа double в диапазоне от 0 до 1. Ноль означает 100% прозрачность – единица 0%.

### **PositionCursor**

Позволяет изменять положение курсора на консольном пространстве. Здесь имеется в виду не курсор, которым мы двигаем мышью, а курсор внутри консоли. Именно с этого курсора мы начинаем вводить текст. Таким образом, благодаря данному методу мы сможем делать записи в разных местах программы.

### Пример применения:

```
ConsoleSetting console{};  
  
COORD coor;  
coor.X = 50; coor.Y = 10;  
  
console.PositionCursor(coor.X, coor.Y).  
    FontSize(20).  
    TextColor(15);  
  
string input;  
while (input != "ZERO")  
{  
    cin >> input;  
    console.PositionCursor(coor.X, coor.Y += 1);  
}
```

### Вывод:

```
Погода действительно раздражала...  
  
Еще вчера было все хорошо,  
  
А уже сегодня ты идешь на экзамен и дохнешь от холода
```

*P.S (Конструкция делает дополнительные отступы, поскольку считает пробелы в строке за отступы в линиях)*

### GetCursorPosition\_X, GetCursorPosition\_Y

Данные функции возвращают значение типа int, которое указывает на положение курсора (имеется в виду курсор консольный, а не тот, который может ставить пользователь в любом месте консоли).

Мы можем использовать эти методы для разных целей. Например, чтобы ограничивать ввод данных (устанавливать размерность строки), или устанавливать максимальное поле для работы консоли.

```

string workLIST;
while (console.GetCursorPosition_Y() != 10)
    cin >> workLIST; // в программе можно будет

```

*работать только до тех пор, пока лимит в 10 строк не будет исчерпан*

```

puts("Введите максимальное число считывания: ");
int len; cin >> len;

ifstream readTEXT("test.txt");
string text;
while (getline(readTEXT, text))
{
    if (console.GetCursorPosition_Y() > len)
        cout << "Данные в файле больше заданного параметра!";
    else
        cout << text;
} // установка

```

*количества данных, которые будут считаны с файла*

### FontSize и FontFamily

Методы изменяют размер шрифта и его семейство. При работе с FontSize важно учесть, что вместе с изменением размеров – меняется и размер консоли (хотя так по-хорошему быть не должно).

#### Пример использования:

```

int main()
{
    setlocale(LC_ALL, "Russian");

    ConsoleSetting cfg{};
    SETTING(cfg);

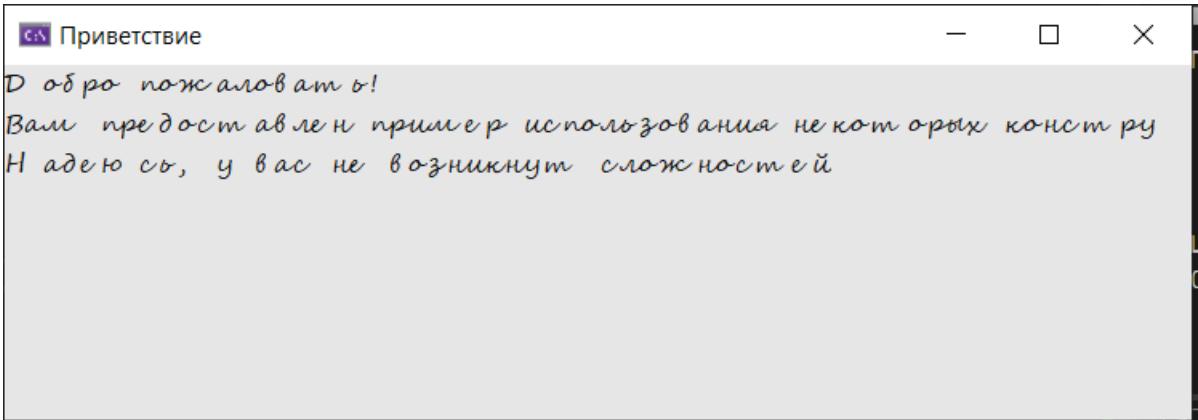
    cout << "Добро пожаловать!\n"
        "Вам предоставлен пример использования некоторых конструкторов.\n"
        "Надеюсь, у вас не возникнут сложностей";

    system("pause");
    return 0;
}

void SETTING(ConsoleSetting config) {
    config.FontSize(20).
        FontFamily("Segoe Script").
        Bold().
        BackgroundColor(230, 230, 230).
        TextColor(15, 15, 15).
        Title("Приветствие").
        MaxSize(60, 10).
        Size(59, 7).
        ScrollBarShow(false);
}

```

В этом примере мы создаем метод SETTING, который является набором правил для консоли. В конфиге мы используем ранее описанные методы (некоторые будут описаны позже). Таким образом мы получим следующее окно:



*P.S (Данный пример никак не изменился вручную. Немного с размером не угадал (не вместились до конца), а еще ошибки в тексте есть)*

### TextBG\_Color

Не имеет никаких аргументов в качестве параметра. Сама по себе функция позволяет изменять цвет текста, основываясь на цвете консоли. Рассматривает цвет консоли по RGB системе.

### GetRedValue, GetGreenValue, GetBlueValue

Функции, которые позволяют определить текущий цвет консоли. Функции не содержат аргументов и возвращают тип int.

```
console.BackgroundColor(RandInt(0, 255),
    RandInt(0, 255),
    255);

cout << "Цвет консольного окна в RGB системе:\n"
    "R - " << console.GetRedValue() << "\n"
    "G - " << console.GetGreenValue() << "\n"
    "B - " << console.GetBlueValue() << "\n"; // получение цветов из
```

случайной палитры (голубого – синих оттенков)

### TextColor

Функция, которая позволяет изменять цвет текста. Поддерживает несколько вариаций (перегрузок): изменение по RGB, по HEX, по стандартному консольному цветовому пространству.

Перегрузки функций:

```
ConsoleSetting& TextColor(int R, int G, int B) { ... }

ConsoleSetting& TextColor(string hexColor) { ... }

ConsoleSetting& TextColor(int color) { ... }
```

При работе с последней перегрузкой нужно быть внимательнее, поскольку она не зависит от привычных систем цветопередачи. Для работы можете воспользоваться следующей таблицей:

Чёрный — "Black" — 0
Тёмно-синий — "DarkBlue" — 1
Тёмно-зелёный — "DarkGreen" — 2
Тёмно-голубой — "DarkCyan" — 3
Тёмно-красный — "DarkRed" — 4
Тёмно-малиновый — "DarkMagenta" — 5
Тёмно-жёлтый — "DarkYellow" — 6
Серый — "Gray" — 7
Тёмно-серый — "DarkGray" — 8
Синий — "Blue" — 9
Зеленый — "Green" — 10
Голубой — "Cyan" — 11
Красный — "Red" — 12
Малиновый (Пурпурный) — "Magenta" — 13
Желтый — "Yellow" — 14
Белый — "White" — 15

При использовании других параметров могут быть неожиданные результаты:

```
ConsoleSetting cfg{};  
cfg  
  
.PositionCursor(40,20)  
  
.FontSize(20);  
  
cfg.TextColor(666);  
  
cout << "TEXT PREVIEW FOR TESTING THIS...\n";  
  
cfg.PositionCursor(35, 22);
```

Будет вывод:



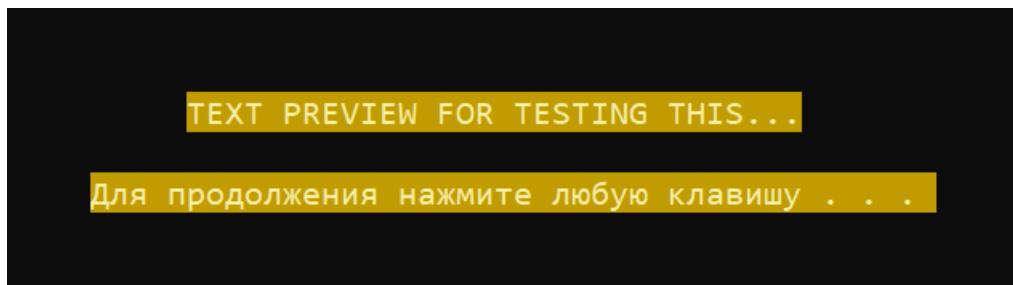
TEXT PREVIEW FOR TESTING THIS...

Для продолжения нажмите любую клавишу . . .

При значении в 1005256 следующее:



При 110:



*P.S (Как по мне вполне приятный вариант для черного фона)*

В свою очередь, если вы хотите точечные результаты по знакомым и применяемым в большинстве случаев системам – можете воспользоваться первыми двумя перегрузками.

### **BackgroundColor**

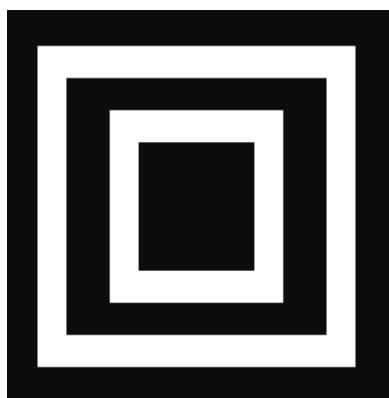
Работает таким же образом, как и TextColor за исключением того, что содержит лишь две перегрузки: RGB и HEX. Пример использования данного метода был показан чуть ранее (предыдущий лист).

### **ForegroundColor**

Ранее говорилось о неоднозначности применения данного свойства в одиночном случае. Здесь же, Foreground выступает в качестве дополнения. То есть он может применяться с разными параметрами настройки консоли, поскольку может входить в цепочку методов.

**Так, к примеру, мы можем даже создавать некоторые контуры и фигуры** (хотя, чтобы реализовать нечто подобное понадобиться довольно много времени. Поэтому лучше воспользоваться готовыми решениями).

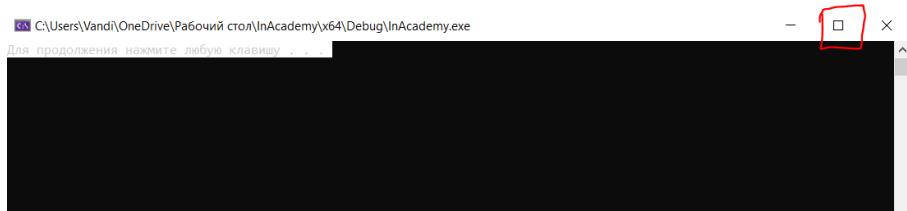
К примеру, такую:



## Перекраска фона консоли через Foreground:

```
ConsoleSetting cfg{};  
cfg.ForegroundColor(255, 255, 255);
```

Затем кликаем на кнопку растягивания консоли:



И получаем по итогу фон:



Также работает и с другими цветами.

## ChangeEncode и GetEncode

У многих мог возникать вопрос, почему при вводе русских символов и последующем их выводе нарушается кодировка.

```
кодировка  
Е҃҃Ёа҃҃Ё 866 // вывод русского текста
```

Проблема заключается в том, что Windows содержит несколько кодировок для консоли (cp1251, cp866). По умолчанию используется последняя.

Чтобы решить подобную проблему вывода – можно изменить кодировку с помощью метода ChangeEncode, который в зависимости от текущей кодировки производит замену на другую (если текущая 866 – будет 1251. И наоборот).

```
ConsoleSetting con = {con.ChangeEncode()};  
  
String a; cin >> a;  
cout << a << " " << con.GetEncode(); // изменение кодировки
```

С помощью функции GetEncode можно получить информацию о текущей кодировке

## 4. Класс анимации

Небольшой класс, который специализируется на анимации некоторых компонентов. Для объявления функций данного класса необходимо обращаться не к пространству ConsoleSetting, а Animation.

```
string str; cin >> str;

Animation Animation;
Animation.AnimationText(str, 100).ConsoleFadeOut(3000);
    // анимация текста      // анимация консоли
```

### AnimationText

Анимирует текст. Данный метод посимвольно выводит значение, записанное в качестве первого аргумента с задержкой, описанной в качестве второго аргумента. При заполнении следует учитывать, что задержка указывается в миллисекундах (как общепринято при описании задержки).

### ConsoleFadeIn, ConsoleFadeOut

Данный методы позволяют плавно показывать исчезновение и появление консольного окна. Данные методы принимают только 1 параметр – параметр задержки, который также указывается в миллисекундах. В своей логике используется метод Opacity из ConsoleSetting.

```
Animation Animation{};

for(;;)
{
    Animation.ConsoleFadeOut(1000); // исчезновение консоли
    Animation.ConsoleFadeIn(1000); // появление консоли
}
```

**Console\_LeftToRight,**                   **Console\_RightToLeft,**                   **Console\_UpToDown,**  
**Console\_DownToUp**

Методы, которые позволяют плавно (псевдо-плавно) проявлять консоль в разных направлениях. Все методы представляют собой конструкции с циклом внутри, который изменяет размеры формы по заданному количеству времени (установлено автором).

```
Animation ConsoleAnimation{};
ConsoleAnimation.Console_LeftToRigth(console.GetConsoleSize_X(),
                                         console.GetConsoleSize_Y());
```

Все четыре метода содержат два параметра: X, Y. Данные значения являются целочисленными и обозначают конечную позицию окна, которую должна принять консоль после анимации.

**ВАЖНО учитывать**, что при анимации справа-налево и слева-направо в качестве параметра в цикле будет X, а Y не будет изменяться – он является «константной» длиной окна. Также и наоборот. Например, если мы анимируем консоль слева-направо, то при значении 100, 35 – консоль начнет прорисовку с X = 0 и Y = 35 до тех пор, пока X не будет 100.

### Console\_CustomAnimate

Метод, который позволяет более гибко работать с анимацией, однако, более сложен в использовании, поскольку принимает 6 аргументов в качестве параметров.

Прототип имеет следующий вид:

```
void Console_CustomAnimate(int x0, int x1, int y0, int y1, int time, string direction)  
  
// стартовый размер консоли по X  
// конечный размер консоли по X  
// стартовый размер консоли по Y  
// конечный размер консоли по Y  
// время анимации  
// направление (откуда начало)
```

Разберем небольшой пример применения `Console_CustomAnimate`:

```
Animation AN;  
AN.Console_CustomAnimate(80, NULL, // значения X начального и X конечного  
                        console.GetConsoleSize_Y(), 10, // значения Y начального и Y конечного  
                        1000, AN.DOWN); // время отрисовки и направление старта анимации
```

Поскольку направление анимации начинается снизу (то есть анимация будет идти снизу-вверх), то X будет «константным» для размера консоли в данной анимации. В связи с этим мы можем указать в качестве первого параметра значение, а в качестве второго оставить `NULL`, поскольку X изменяться в данной анимации не будет.

Затем указываем параметры Y. Так как Y начинает свою анимацию снизу, то Y конечный должен быть меньше Y начального. В качестве начального параметра указываем текущую длину консоли по Y с помощью `GetConsoleSize_Y`. Конечным Y можем поставить любое число, главное, чтобы оно было меньше начального значения.

Устанавливаем время анимации. Как и в предыдущих методах, анимация указывается в миллисекундах. В конце прописываем начало анимации (с какой стороны она будет начинаться). Можно использовать готовые переменные класса, как показано на скриншоте выше, а можно использовать значения «LEFT», «RIGHT», «UP», «DOWN».

## Компонент GeometryLIB\_Figure

Данный компонент библиотеки typedPL посвящен формулам нахождения периметра, площади, объема различных геометрических фигур.

**Функционал каждого метода можно определить по приписке в каждом названии.** Прежде всего, объекты делятся на 2D и 3D – то есть на фигуры плоскости и пространства.

**Различия названий функций по назначению:**

- 2D (площадь фигуры);
- 3D (объем фигуры);
- 2D\_P (периметр фигуры);
- 3D\_S (площадь поверхности фигуры).

Таким образом данный компонент можно разделить, преимущественно, на 4 блока, который включает в себя перечень геометрических фигур.

### 1. Периметр

#### ТРЕУГОЛЬНИК

##### Triangle2D\_P\_MiddleLine

Периметр треугольника по средним линиям

Принимает аргументы: **сторона А, сторона Б, сторона С**

Формула:  $2a + 2b + 2c$

---

##### Triangle2D\_P\_TwoSide

Периметр треугольника по двум сторонам и углу между ними

Принимает аргументы: **сторона А, сторона Б, угол**

Формула:  $a + b + \sqrt{a^2 + b^2 - 2ab \cdot \cos(\text{angle})}$

---

##### Triangle2D\_P\_Isosceles

Периметр равнобедренного треугольника

Принимает параметры: **сторона А (основание), сторона Б (боковая линия)**

Формула:  $2b + a$

---

##### Triangle2D\_P\_Isosceles\_HeightAndMain

Периметр равнобедренного треугольника по высоте и основанию

Принимает параметры: **сторона А (основание), высота**

Формула:  $a + 2\sqrt{\left(\frac{a}{2}\right)^2 + h^2}$

---

### **Triangle2D\_P\_Isosceles\_HeightAndSide**

Периметр треугольника по высоте и боковым сторонам

Принимает параметры: **сторона А (боковая линия), высота**

Формула:  $2a + 2\sqrt{a^2 - h^2}$

---

## **КВАДРАТ**

### **Square2D\_P**

Периметр квадрата по сторонам

Принимает параметры: **сторона А**

Формула:  $4a$

---

### **Square2D\_P\_Diagonal**

Периметр квадрата по диагонали

Принимает параметры: **диагональ**

Формула:  $2\sqrt{2} \cdot d$

---

### **Square2D\_P\_FromS**

Периметр квадрата, исходя из его площади

Принимает параметры: **площадь**

Формула:  $4\sqrt{S}$

---

### **Square2D\_P\_CircleOut**

Периметр квадрата по описанной окружности

Принимает параметры: **радиус**

Формула:  $4\sqrt{2} \cdot R^2$

---

### **Square2D\_P\_CircleIn**

Периметр квадрата по вписанной окружности

Принимает параметры: **радиус**

Формула:  $8R$

---

## **ПРЯМОУГОЛЬНИК**

### **Rentangle2D\_P**

Периметр прямоугольника по сторонам

Принимает параметры: **сторона А, сторона Б**

Формула:  $2(a+b)$

---

### Rentangle2D\_P\_Diagonal

Периметр прямоугольника по диагонали

Принимает параметры: **сторона А, диагональ**

Формула:  $2(a+\sqrt{d^2-a^2})$

## ПАРАЛЛЕЛОГРАММ

### Parallelogram2D\_P

Периметр параллелограмма по сторонам

Принимает параметры: **сторона А, сторона Б**

Формула:  $2(a+b)$

---

### Parallelogram2D\_P\_Diagonal

Периметр параллелограмма по диагоналям и стороне

Принимает параметры: **сторона А, диагональ 1, диагональ 2**

Формула:  $2a+\sqrt{2d_1^2+2d_2^2-4a^2}$

---

### Parallelogram2D\_P\_Corner

Периметр параллелограмма по стороне, высоте и углу

Принимает параметры: **сторона А, высота, угол**

Формула:  $2\left(a+\left(\frac{h}{\sin(angle)}\right)\right)$

---

## РОМБ

### Rhombus2D\_P

Периметр ромба по сторонам

Принимает параметры: **сторона А**

Формула:  $4a$

---

### Rhombus2D\_P\_Diagonal

Периметр ромба по диагоналям

Принимает параметры: **диагональ 1, диагональ 2**

Формула:  $2\sqrt{d_1^2+d_2^2}$

## **КОЛЬЦО**

### **CircleHole2D\_P**

Периметр кольца по внутреннему и внешнему радиусу

Принимает параметры: **внутренний радиус (r), внешний радиус (R)**

Формула:  $2\pi \cdot (R+r)$

## **ПРАВИЛЬНЫЙ МНОГОУГОЛЬНИК**

### **Polygon2D\_P**

Периметр правильного многоугольника по длине стороны и их количеству

Принимает параметры: **сторона А, количество сторон**

Формула:  $a \cdot n$

## **2. Площадь**

### **ТРЕУГОЛЬНИК**

#### **Triangle2D**

Площадь треугольника по высоте и основанию

Принимаемые параметры: **сторона А (основание), высота**

Формула:  $\frac{1}{2}ah$

---

#### **Triangle2D\_TwoSide**

Площадь треугольника по двум сторонам и углу между ними

Принимаемые параметры: **сторона А, сторона Б, угол**

Формула:  $\frac{1}{2}ab \cdot \sin(angle)$

---

#### **Triangle2D\_AllSide**

Площадь треугольника по всем его сторонам

Принимаемые параметры: **сторона А, сторона Б, сторона С**

Формула:  $\sqrt{P \cdot (P-a)(P-b)(P-c)}$

---

#### **Triangle2D\_CircleOut**

Площадь треугольника по описанной окружности

Принимаемые параметры: **сторона А, сторона Б, сторона С, радиус**

Формула:  $\frac{abc}{4R}$

---

### **Triangle2D\_CircleIn**

Площадь треугольника по вписанной окружности

Принимаемые параметры: **сторона А, сторона Б, сторона С, радиус**

Формула:  $\frac{abc}{2}R$

---

### **Triangle2D\_Equilateral\_Height**

Площадь равностороннего треугольника по его высоте

Принимаемые параметры: **высота**

Формула:  $\frac{h^2}{\sqrt{3}}$

---

### **Triangle2D\_Equilateral\_AllSide**

Площадь равностороннего треугольника по трем сторонам

Принимаемые параметры: **сторона А**

Формула:  $\frac{\sqrt{3}a^2}{4}$

---

### **Triangle2D\_Equilateral\_CircleOut**

Площадь равностороннего треугольника по описанной окружности

Принимаемые параметры: **радиус**

Формула:  $\frac{3\sqrt{3}a^2}{4}$

---

### **Triangle2D\_Equilateral\_CircleIn**

Площадь равностороннего треугольника по вписанной окружности

Принимаемые параметры: **радиус**

Формула:  $3\sqrt{3}R^2$

---

### **Triangle2D\_Isosceles\_AllSide**

Площадь равнобедренного треугольника по его сторонам

Принимаемые параметры: **сторона А, сторона Б**

Формула:  $\frac{b}{4}\sqrt{4a^2-b^2}$

---

## **КВАДРАТ**

### **Square2D**

Площадь квадрата по сторонам

Принимаемые параметры: **сторона А**

Формула:  $a^2$

---

### **Square2D\_Diagonal**

Площадь квадрата по его диагонали

Принимаемые параметры: **диагональ**

Формула:  $\frac{d^2}{2}$

## **ПРЯМОУГОЛЬНИК**

### **Rentangle2D**

Площадь прямоугольника по сторонам

Принимает параметры: **сторона А, сторона Б**

Формула:  $ab$

---

### **Rentangle2D\_Diagonal**

Площадь прямоугольника по стороне и диагонали

Принимаемые параметры: **сторона А, диагональ**

Формула:  $a\sqrt{d^2 - a^2}$

---

### **Rentangle2D\_DiagonalAndCorner**

Площадь прямоугольника по диагонали и ее углу

Принимаемые параметры: **диагональ, угол**

Формула:  $\frac{d^2 \cdot \sin(\text{angle})}{2}$

## **ПАРАЛЛЕЛОГРАММ**

### **Parallelogram2D**

Площадь параллелограмма по его основанию и высоте

Принимаемые параметры: **сторона А (основание), высота**

Формула:  $ah$

---

### **Parallelogram2D\_Corner**

Площадь параллелограмма по его сторонам и углу между ними

Принимаемые параметры: **сторона А, сторона Б, угол**

Формула:  $ab \sin(\text{angle})$

---

### **Parallelogram2D\_DiagonalAndCorner**

Площадь параллелограмма по его диагоналям и углу между ними

Принимаемые параметры: **диагональ 1, диагональ 2, угол**

Формула:  $\frac{d1 \cdot d2 \cdot \sin(angle)}{2}$

## РОМБ

### Rhombus2D

Площадь ромба по стороне и высоте

Принимаемые параметры: **сторона А, высота**

Формула:  $ah$

---

### Rhombus2D\_Corner

Площадь ромба по стороне и углу

Принимаемые параметры: **сторона А, угол**

Формула:  $a^2 \sin(angle)$

---

### Rhombus2D\_Diagonal

Площадь ромба по его диагоналям

Принимаемые параметры: **диагональ 1, диагональ 2**

Формула:  $\frac{d1 \cdot d2}{2}$

## ТРАПЕЦИЯ

### Trapezoid2D

Площадь трапеции по ее основаниям и высотке

Принимаемые параметры: **сторона А (основание 1), сторона Б (основание 2), высота**

Формула:  $\frac{1}{2}(a+b)h$

---

### Trapezoid2D\_AllSide

Площадь трапеции по всем ее сторонам

Принимаемые параметры: **сторона А, сторона Б, сторона С, сторона Д**

Формула:  $\frac{a+b}{|a-b|} \sqrt{(P-a)(P-b)(P-a-c)(P-a-d)}$

---

### Trapezoid2D\_Diagonal

Площадь трапеции по диагоналям и их углу

Принимаемые параметры: **диагональ 1, диагональ 2, угол**

Формула:  $d1 \cdot d2 \cdot \sin(angle)$

---

### **Trapezoid2D\_MiddleLine**

Площадь трапеции по средней линии и высоте

Принимаемые параметры: **средняя линия, высота**

Формула:  $Mh$

## **КРУГ**

### **Circle2D**

Площадь круга по радиусу

Принимаемые параметры: **радиус**

Формула:  $\pi R^2$

---

### **Circle2D\_Diameter**

Площадь круга по диаметру

Принимаемые параметры: **диаметр**

Формула:  $\pi\left(\frac{d^2}{4}\right)$

---

### **Circle2D\_CircleLength**

Площадь круга по длине окружности

Принимаемые параметры: **длина окружности**

Формула:  $\frac{L^2}{4\pi}$

---

### **Circle2D\_Part**

Площадь части круга

Принимаемые параметры: **радиус, угол**

Формула:  $\pi R^2 \cdot \left(\frac{\text{angle}}{360}\right)$

---

## **КОЛЬЦО**

### **CircleHole2D**

Площадь кольца по радиусам

Принимаемые параметры: **внутренний радиус, внешний радиус**

Формула:  $\pi(R^2 - r^2)$

---

### **CircleHole2D\_Diameter**

Площадь кольца по диаметрам

Принимаемые параметры: **внутренний диаметр, внешний диаметр**

Формула:  $\frac{\pi}{4}(D^2 - d^2)$

## ПРАВИЛЬНЫЙ МНОГОУГОЛЬНИК

### Polygon2D\_Apophemera

Площадь правильного многоугольника по его апофемере и площади

Принимаемые параметры: **площадь, апофемера**

Формула:  $\frac{1}{2}PA$

---

### Polygon2D\_Side

Площадь правильного многоугольника по его сторонам

Принимаемые параметры: **сторона А, количество сторон**

Формула:  $\frac{na^2}{4}\operatorname{ctg}\left(\frac{180}{n}\right)$

---

### Polygon2D\_CircleIn

Площадь правильного многоугольника по вписанной окружности

Принимаемые параметры: **радиус, количество сторон**

Формула:  $nr^2 \tan\left(\frac{180}{n}\right)$

---

### Polygon2D\_CircleOut

Площадь правильного многоугольника по описанной окружности

Принимаемые параметры: **радиус, количество сторон**

Формула:  $\frac{nr^2}{4} \sin\left(\frac{360}{n}\right)$

---

## ШЕСТИУГОЛЬНИК

### Hexagon2D\_Side

Площадь шестиугольника по его сторонам

Принимаемые параметры: **сторона А**

Формула:  $\frac{3\sqrt{3}a^2}{2}$

---

### Hexagon2D\_CircleOut

Площадь шестиугольника по описанной окружности

Принимаемые параметры: **радиус**

Формула:  $\frac{3\sqrt{3}R^2}{2}$

---

### **Hexagon2D\_CircleIn**

Площадь шестиугольника по вписанной окружности

Принимаемые параметры: **радиус**

Формула:  $\sqrt{3}R^2$

---

### **Hexagon2D\_Diagonal**

Площадь шестиугольника по диагонали

Принимаемые параметры: **диагональ**

Формула:  $\frac{3\sqrt{3}D^2}{8}$

---

### **Hexagon2D\_ShortDiagonal**

Площадь шестиугольника по короткой диагонали

Принимаемые параметры: **диагональ**

Формула:  $\frac{\sqrt{3}D^2}{2}$

---

### **Hexagon2D\_Perimetr**

Площадь шестиугольника, исходя из его периметра

Принимаемые параметры: **периметр**

Формула:  $\frac{3\sqrt{3}\left(\frac{P}{6}\right)^2}{2}$

---

## **ВОСЬМИУГОЛЬНИК**

### **Octagon2D\_Side**

Площадь восьмиугольника по сторонам

Принимаемые параметры: **сторона А**

Формула:  $2a^2(\sqrt{2}+1)$

---

### **Octagon2D\_CircleOut**

Площадь восьмиугольника по описанной

Принимаемые параметры: **радиус**

Формула:  $2R^2\sqrt{2}$

---

### **Octagon2D\_CircleIn**

Площадь многоугольника по вписанной окружности

Принимаемые параметры: **радиус**

Формула:  $8R^2(\sqrt{2}-1)$

## РАСЧЕТ ПО ТОЧКАМ

### Figure2D\_Points

Функция, которая находит площадь любой фигуры по точке.

Принимает параметры: **initialized\_list**

**Пример использования:**

```
using namespace GeometryLIB_Figure;
cout << Figure2D_Points({
    1,5,
    3,1,
    7,4
});
```

Обратите внимание: поскольку вычисления происходят с точками, то количество параметров аргумента должно быть четным. В противном случае функция возвращает просто ноль.

Функция реализована на аргументе «списочного типа» `#include <initializer_list>`

Данная строка уже прописана в заголовочном файле библиотеке, поэтому не обязательно объявлять ее в своем проекте. Данный аргумент принимает один параметр в виде списка на подобие: `{1,6,2,1,5,1}`.

---

### Figure2D\_Sides

Функция, которая позволяет найти площадь любой фигуры по ее сторонам

Принимает параметры: **initialized\_list**

**Пример использования:**

```
using namespace GeometryLIB_Figure;
cout << Figure2D_Sides({
    10,15,13,17
});
```

Принимает параметры в виде сторон фигуры. В данном случае вычисляет площадь прямоугольника. В основе логики лежит формула Герона.

## 3. Объем

### ТРЕУГОЛЬНИК

#### Triangle3D

Объем треугольника (тетраэдра) по его площади и высоте

Принимаемые параметры: **площадь, высота**

Формула:  $\frac{1}{3}Sh$

---

## **Triangle3D\_Equilateral**

Объем равностороннего тетраэдра по его стороне

Принимаемые параметры: **сторона А**

Формула:  $\frac{\sqrt{2}a^3}{12}$

---

## **КОНУС**

### **Cone3D**

Объем конуса

Принимаемые параметры: **радиус, высота**

Формула:  $\frac{1}{3}\pi R^2 h$

---

### **Cone3D\_Part**

Объем части конуса

Принимаемые параметры: **радиус 1, радиус 2, высота**

Формула:  $\frac{1}{3}\pi h(R1^2 + R1 \cdot R2 + R2^2)$

---

## **ОТДЕЛЬНЫЕ ФИГУРЫ**

### **Square3D**

Объем куба

Принимаемые параметры: **сторона А**

Формула:  $a^3$

---

### **Rentangle3D**

Объем прямоугольного параллелепипеда

Принимаемые параметры: **сторона А, сторона Б, высота**

Формула:  $abh$

---

### **Parallelogram3D**

Объем параллелепипеда

Принимаемые параметры: **площадь, высота**

Формула:  $Sh$

---

### **Cilinder3D**

Объем цилиндра

Принимаемые параметры: **радиус, высота**

Формула:  $\pi R^2 h$

---

### Hexagon3D

Объем правильного шестиугольника по его стороне и высоте

Принимаемые параметры: **сторона А, высота**

Формула:  $\frac{3\sqrt{3}}{2}a^2h$

## КРУГ И КОЛЬЦО

### Circle3D

Объем шара по его радиусу

Принимаемые параметры: **радиус**

Формула:  $\frac{4}{3}\pi R^3$

---

### Circle3D\_Diameter

Объем шара по его диаметру

Принимаемые параметры: **диаметр**

Формула:  $\frac{1}{6}\pi D^3$

---

### Circle3D\_CircleLength

Объем шара по длине его окружности

Принимаемые параметры: **длина окружности (L)**

Формула:  $\frac{L^3}{6\pi^2}$

---

### CircleHole3D

Объем кольца

Принимаемые параметры: **высота, внутренний радиус**

Формула:  $\pi R^2 h$

## ПИРАМИДА

### Pyramid3x3D

Объем четырехугольной пирамиды

Принимаемые параметры: **сторона А, высота**

Формула:  $\frac{ha^2}{4\sqrt{3}}$

---

### Pyramid4x3D

Объем четырехугольной пирамиды

Принимаемые параметры: **сторона А, высота**

Формула:  $\frac{1}{3}ha^2$

---

### Pyramid5x3D

Объем пятиугольной пирамиды

Принимаемые параметры: **сторона А, высота**

Формула:  $\frac{5}{6}ha\sqrt{\left(\frac{a}{2\sin(\frac{\pi}{5})}\right)^2 - \frac{a^2}{4}}$

---

### Pyramid6x3D

Объем шестиугольной пирамиды

Принимаемые параметры: **сторона А, высота**

Формула:  $\frac{\sqrt{3}}{2}ha^2$

---

### PyramidNx3D

Объем четырехугольной пирамиды

Принимаемые параметры: **сторона А, высота**

Формула:  $\frac{nha^2}{12 \tan\left(\frac{\text{angle}}{n}\right)}$

## ПРИЗМА

### Prism3x3D

Объем треугольной призмы

Принимаемые параметры: **сторона А, высота**

Формула:  $\frac{\sqrt{3}}{4}a^2h$

---

### Prism4x3D

Объем четырехугольной призмы

Принимаемые параметры: **сторона А, высота**

Формула:  $a^2h$

---

### Prism5x3D

Объем пятиугольной призмы

Принимаемые параметры: **сторона А, высота**

Формула:  $\frac{5}{4}a^2h\sqrt{\frac{1}{\sin^2(\frac{\pi}{5})} - 1}$

---

## **Prism6x3D**

Объем шестиугольной призмы

Принимаемые параметры: **сторона А, высота**

Формула:  $\frac{3\sqrt{3}}{2}a^2h$

## **4. Площадь поверхности**

### **Square3D\_S\_Side**

Площадь поверхности куба

Принимает параметры: **сторона А**

Формула:  $6a^2$

---

### **Square3D\_S\_Diagonal**

Площадь поверхности куба по диагонали

Принимает параметры: **диагональ**

Формула:  $\frac{6d}{\sqrt{2}}$

---

### **Rentangle3D\_S\_Side**

Площадь поверхности прямоугольника

Принимает параметры: **сторона А, сторона Б, высота**

Формула:  $2 \cdot ab + ah + bh$

---

### **Circle3D\_S\_Radius**

Площадь поверхности шара по радиусу

Принимает параметры: **радиус**

Формула:  $4\pi R^2$

---

### **Circle3D\_S\_Diagonal**

Площадь поверхности шара по диаметру

Принимает параметры: **диагональ**

Формула:  $\pi D^2$

---

### **Cone3D\_SPart**

Площадь поверхности части конуса

Принимает параметры: **радиус, образующая конуса**

Формула:  $\pi RL$

---

### **Cone3D\_SFull**

Площадь поверхности всего конуса

Принимает параметры: **радиус, образующая конуса**

Формула:  $2\pi R(R + L)$

---

### **Cilinder3D\_SPart**

Площадь поверхности части цилиндра

Принимает параметры: **радиус, высота**

Формула:  $2\pi Rh$

---

### **Cilinder3D\_SFull**

Площадь поверхности всего цилиндра

Принимает параметры: **радиус, высота**

Формула:  $2\pi R(R + h)$

## **5. Иные формулы**

### **LengthCircle\_Radius**

Функция для поиска длины окружности по ее радиусу

Принимает параметры: **радиус**

Формула:  $2\pi R$

---

### **LengthCircle\_Diameter**

Функция для поиска длины окружности по ее диаметру

Принимает параметры: **диаметр**

Формула:  $\pi D$

---

### **LengthCircle\_FromS**

Функция для поиска длины окружности, исходя из ее площади

Принимает параметры: **площадь**

Формула:  $\sqrt{4\pi S}$

## Константы GeometryLIB\_Figure

Компонент GeometryLIB\_Figure поддерживает некоторые константы. Для их подключения необходимо объявить директиву define перед подключением самой библиотеки (если это сделать позже – константы не будут видны для использования).

```
#define __USE_CONSTANT_ // подключение констант
#include "Geometry.h" // подключение библиотеки
#include "FPFC.h" // подключение видимости
```

констант

Во многом данная директива ориентируется на давно существующий \_\_USE\_MATH\_DEFINES. Единственным отличием можно назвать только то, что данная директива включает в себя чуть больше констант, а также затрагивает частично высшую математику.

Перечень констант компонента GeometryLIB\_Figure:

```
// МАТЕМАТИКА
#define PI 3.14159265358979323846264338327950288 // число ПИ
#define _PI_2_ 6.283185307179586 // число 2 ПИ
#define I = -1 // мнимая единица
#define _SQRT_2_ 1.41421356237309504880168872420969808 // корень 2
#define _SQRT_3_ 1.73205080756887729352744634150587237 // корень 3
#define _SQRT_5_ 2.23606797749978969 // корень 5
#define E 2.7182818284590452353602874713526625 // число Эйлера (константа Непера)
#define _LG_2_ 1.44269504088896340736 // логарифм 2
#define _LG_10_ 0.434294481903251827651 // логарифм 10
#define _LN_2_ 0.693147180559945309417232121458 // натуральный LOG 2
#define _LN_10_ 2.30258509299404568401799145468 // натуральный LOG 10
#define M 0.57721566490153286060651209008240243 // Маскерони (постоянная Эйлера)
#define LPS 0.66274341934918158097474209710925290 // предел Лапласа
#define _GOLD_R_ 1.61803398874989484820458683436563812 // золотое сечение
// ВЫСШАЯ МАТЕМАТИКА
#define APERI 1.20205690315959428539973816151144999 // Апери
#define KINKLEN 1.2824271291006226368753425688697917 // постоянная Глейшера – Кинкелина
#define CATALAN 0.915965594177219015054603514932384110774 // постоянная Каталана
#define CAENA 0.64341054629 // константа Каэна
#define DOTTY 0.73908513321516064165531208767387340 // число Дотти
#define DICKMAN 0.624329988543550870992936 // постоянная Голубма (Дикмана)
#define SOLDNER 1.451369234883381050283968485892027 // константа Солднера
#define TREFETENA 0.70258 // константа Эмбри (Трефетена)
#define TWINS 0.66016181584686957392781211001455577 // константа простых близнецов
#define TWINS_B 1.902160583104 // константа простых близнецов по Бруно
#define F 4.66920160910299067185320382046620161 // постоянная Фейгенбаума
#define KHINCHIN 2.685452001065 // постоянная Хинчина
```

## Компонент GeometryLIB\_Expression

Компонент для работы с выражениями разных типов. Компонент основывается на базе разных классов, которые содержат операции над объектами из геометрии и математики.

**Подключение компонента:**

```
#include "Geometry.h"
using namespace GeometryLIB_Expression;
```

На момент версии 3.2 данный компонент содержит два класса для комплексных чисел и векторов.

**Инициализация переменных классов:**

```
VectorExpression vec1(-1, 2, -2);
VectorExpression vec2 = {2,5};

ComplexExpression com1(-2, 0);
```

Выражения принимают параметры типа double.

### 1. Класс для работы с комплексными числами

Данный класс позволяет работать с комплексными числами, а именно осуществлять операции над ними (+ - \* /). При инициализации переменной важно придерживаться единого формата записи value (double a, double b), где первый аргумент – действительная часть числа, а b – его мнимая часть. Напомним, что мнимая часть – это та часть, которая содержит мнимую единицу i (-1).

Сам по себе класс включает в себя всего 4 метода для осуществления действий над комплексными числами, а также несколько функций для получения данных о комплексном числе.

```
ComplexExpression com1(0, 0), com2(-10,23), com3(3.12, -0.98);

ComplexExpression comRESULT = com1 * com2 + com3;
ComplexExpression comRESULT2 = com1.Complex_Multiplication(com2).Complex_Addition(com3);

// RESULT и RESULT2 равнозначны

cout << comRESULT.Get_Expression() << " " << comRESULT2.Get_Expression() << endl;

// операции над комплексными числами

ComplexExpression complex = {4,1};
double A = complex.Get_A() + complex.Get_B(); // сложение двух частей

String str = ToTypedPL_String(A);
string str2 = complex.Get_Expression(); // получение выражения

// получение данных о комплексном числе
```

Мы можем получить данные о действительной (A) или мнимой (B) части, или же получить все выражение целиком. Например, для комплексного числа 4,1 выражение будет  $4 + 1i$ , где 4 – действительная часть, и  $1i$  – мнимая.

## 2. Класс для работы с векторами

В этом классе вы сможете работать с векторами, находить длины их сторон, совершать над ними операции.

Как и ComplexExpression, VectorExpression также содержит перегрузки операторов, которые позволяют сокращенно, без использования функций, производить операции над векторами.

```
VectorExpression vec1(-1, 2, -2);
VectorExpression vec2 = {2,5};

vec1 += vec2;
VectorExpression result = vec1.Vector_Addition(vec2) * 3.1415;
```

### Получение длины вектора.

Длину вектора можно получить несколькими способами. Библиотека typedPL может предложить вам несколько вариантов нахождения длины в зависимости от известных данных.

```
VectorExpression vec0(1,1), vec1 = { 2,2,2 };

cout << vec1.Vector_Length() << endl; // получение длины по вектору
cout << vec0.Vector_Length(vec1) << endl; // получение длины по началу и концу точек
| // поскольку мы можем интерпретировать параметры вектора в точки

cout << Vector_Length(vec0.Vector_Length(), vec1.Vector_Length(), 90) << endl;
// длина вектора по двум его величинам и углу между ними
```

Вы можете воспользоваться одним из трех вариантов. Две функции находятся непосредственно внутри класса. Поэтому без наличия экземпляра класса – вы не сможете ими воспользоваться. В случае с поиском длины через угол между векторами – данная функция находится вне класса, что позволяет пользоваться ею, как и многими другими функциями из блока Geometry.

### Получить косинус для вычисления длины можно следующим образом:

```
VectorExpression vec0(1,1,1), vec1 = { 2,2,2 };
double COS = vec0.Get_Cos(vec1); // получения косинуса между углами
double A = vec0.Vector_Length();
double B = vec1.Vector_Length();

cout << "Длина вектора равна: " << Vector_Length(A, B, COS);
```

### Equals, EqualsMax, EqualsMin

Булевые функции для сравнения векторов между собой. В случае с Min и Max сравнивают не сами координаты, а именно длины векторов.

```
VectorExpression vec0(1,1,1), vec1 = { 2,2,2 };
bool a = vec0.Equals(vec1); // false
bool min = vec0.EqualsMin(vec1); // true
bool max = vec0.EqualsMax(vec1); // false
```

// использование булевых

сравнения

### Умножение векторов:

- вектор X вектор;
- скалярное произведение;
- векторное произведение
- произведение на число.

```
VectorExpression vec0(1,1,1), vec1 = { 2,2,2 };

vec0 = vec1 * vec0 * 0.005; // произведение на число и между векторами
vec0 = vec1.Vector_VectorMultiplication(vec0); // произведение векторное

double COS = vec0.Get_Cos(vec1);
double SCALAR = Vector_ScalarMultiplication(vec0.Vector_Length(), vec1.Vector_Length(), cos);
// скалярное произведение
```

На основе векторного произведения вы также можете найти площадь треугольника.

Для вычисления существует следующая функция:

```
cout << vec0.Triangle2D_VectorMultiplication(vec1); // площадь треугольника по
векторному произведению
```

### Получить значения вектора вы можете с помощью функций:

- Get\_X;
- Get\_Y;
- Get\_Z;
- Get\_Expression.

## ПРИЛОЖЕНИЕ FunctionalLIB\_Array (не поддерживается)

Аргументы функций:

### 1. Сумма

```
template<typename T, typename... Args> double Sum(T arg, Args... args){ ... }

template<typename T> double Sum(vector<T> arr){ ... }

template<typename T, size_t N> double Sum(T(&arr)[N]){ ... }

template<typename T, size_t N> double Sum(array<T, N> arr){ ... }
```

// Аргумент1, Аргумент2, ..., АргументX

// ТипМассива

// ТипМассива

// ТипМассива

### 2. Среднее значение

```
template<typename T, typename... Args> double Avg(T arg, Args... args){ ... }

template<typename T> double Avg(vector<T> arr){ ... }

template<typename T, size_t N> double Avg(T(&arr)[N]){ ... }

template<typename T, size_t N> double Avg(array<T, N> arr){ ... }
```

// Аргумент1, Аргумент2, ..., АргументX

// ТипМассива

// ТипМассива

// ТипМассива

### 3. Максимальное значение

```
template<typename T, typename... Args> double Max(T arg, Args... args){ ... }

template<typename T> double Max(vector<T> arr){ ... }

template<typename T, size_t N> double Max(T(&arr)[N]){ ... }

template<typename T, size_t N> double Max(array<T, N> arr){ ... }
```

// Аргумент1, Аргумент2, ..., АргументX

// ТипМассива

// ТипМассива

// ТипМассива

#### 4. Минимальное значение

```
template<typename T, typename... Args> double Min(T arg, Args... args){ ... }

template<typename T> double Min(vector<T> arr){ ... }

template<typename T, size_t N> double Min(T(&arr)[N]){ ... }

template<typename T, size_t N> double Min(array<T, N> arr){ ... }
```

// Аргумент1, Аргумент2, ..., АргументX

// ТипМассива

// ТипМассива

// ТипМассива

#### 5. Сортировка

```
template<typename T> T Sort(vector<T>& arr){ ... }

template<typename T, size_t N> T Sort(T(&arr)[N]){ ... }

template<typename T, size_t N> T Sort(array<T, N>& arr){ ... }
```

// ТипМассива

// ТипМассива

// ТипМассива

#### 6. Переворот

```
template<typename T> T Converse(vector<T>& arr){ ... }

template<typename T, size_t N> T Converse(T(&arr)[N]){ ... }

template<typename T, size_t N> T Converse(array<T, N>& arr){ ... }
```

// ТипМассива

// ТипМассива

// ТипМассива

#### 7. Инверсия

```
template<typename T> double Inversion(T arg){ ... }

template<typename T> double Inversion(vector<T>& arr){ ... }

template<typename T, size_t N> double Inversion(T(&arr)[N]){ ... }

template<typename T, size_t N> double Inversion(array<T, N>& arr){ ... }
```

// Аргумент

// ТипМассива

// ТипМассива

// ТипМассива

## 8. Рандомное значение

```
template<typename T> int RandInt(T min, T max){ ... }

template<typename T> int RandInt(T min, T max, vector<T>& arr){ ... }

template<typename T, size_t N> int RandInt(T min, T max, T(&arr)[N]){ ... }

template<typename T, size_t N> int RandInt(T min, T max, array<T, N>& arr){ ... }
```

// НачалоДиапазона, КонецДиапазона  
// НачалоДиапазона, КонецДиапазона, ТипМассива  
// НачалоДиапазона, КонецДиапазона, ТипМассива  
// НачалоДиапазона, КонецДиапазона, ТипМассива

## 9. Медиана

```
template<typename T> T Median(vector<T>& arr){ ... }

template<typename T, size_t N> T Median(T(&arr)[N]){ ... }

template<typename T, size_t N> T Median(array<T, N>& arr){ ... }
```

// ТипМассива  
// ТипМассива  
// ТипМассива

## 10. Полное заполнение

```
template<typename T> T Fill(vector<T>& arr, T value){ ... }

template<typename T, size_t N> T Fill(T(&arr)[N], T value){ ... }

template<typename T, size_t N> T Fill(array<T, N>& arr, T value){ ... }
```

// ТипМассива, ЗначениеЗаполнения  
// ТипМассива, ЗначениеЗаполнения  
// ТипМассива, ЗначениеЗаполнения

### 10.5 Частичное заполнение

```
template<typename T> T FillPart(vector<T>& arr, T value, int start, int end){ ... }

template<typename T, size_t N> T FillPart(T(&arr)[N], T value, int start, int end){ ... }

template<typename T, size_t N> T FillPart(array<T, N>& arr, T value, int start, int end){ ... }
```

// ТипМассива, ЗначениеЗаполнения, НачалоДиапазона, КонецДиапазона  
// ТипМассива, ЗначениеЗаполнения, НачалоДиапазона, КонецДиапазона  
// ТипМассива, ЗначениеЗаполнения, НачалоДиапазона, КонецДиапазона

## 11. Замена значений

```
template<typename T> T Change(vector<T>& arr, T found, T replace) { ... }

template<typename T, size_t N> T Change(T(&arr)[N], T found, T replace) { ... }

template<typename T, size_t N> T Change(array<T, N>& arr, T found, T replace) { ... }
```

// ТипМассива, ИскомоеЗначение, ЗначениеЗамены

// ТипМассива, ИскомоеЗначение, ЗначениеЗамены

// ТипМассива, ИскомоеЗначение, ЗначениеЗамены

## 12. Объединение

```
template<typename T> vector<T> Union(vector<T>& arr1, vector<T>& arr2) { ... }
```

// Массив1, Массив2

## 13. Размерность

```
template<typename T> int Lgth(vector<T>& arr) { ... }

template<typename T, size_t N> int Lgth(T(&arr)[N]) { ... }

template<typename T, size_t N> int Lgth(array<T, N>& arr) { ... }
```

// ТипМассива

// ТипМассива

// ТипМассива

## 14. Добавление элемента

```
template<typename T> vector<T> Add(vector<T>& arr, T value) { ... }

template<typename T, size_t N> vector<T> Add(T(&arr)[N], T value) { ... }

template<typename T, size_t N> array<T, N + 1> Add(array<T, N>& arr, T value) { ... }
```

// ТипМассива, Значение

// ТипМассива, Значение

// ТипМассива, Значение

## 15. LINQ имитация

```
template<typename T> bool LINQ_OPERATION(T arg, string operation, T paramIF) { ... }

template<typename T> vector<T> LINQ_OPERATION(vector<T>& arr, string operation, T paramIF) { ... }
```

// Аргумент, Операция, Параметр

// ТипМассива, Операция, Параметр

## **ПРИЛОЖЕНИЕ FunctionalLIB\_Array – Class List**

### **Список класса:**

- операторы и размерность массива;
- функции класса;
- методы класса (манипуляция с массивами);
- прогрессии.

### **Операторы и размерность массива:**

- begin // итератор
- end // итератор
- First; // функция, возвращающая номер первого элемента (константно 0)
- Last; // функция, возвращающая норм последнего элемента (длина)
- FirstElement; // возвращает первый элемент массива
- LastElement; // возвращает последний элемент массива
- Count; // функция, возвращающая длину массива
- operator [];// оператор возврата индекса массива
- operator + (перегрузка);
- operator - (перегрузка); // вычитание каждого элемента массива на аргумент
- operator \* (перегрузка); // умножение каждого элемента массива на аргумент
- operator / (перегрузка); // деление каждого элемента массива на аргумент
- operator % (Т аргумент); // остаток каждого элемента массива на аргумент
- operator += (перегрузка); // присвоение со сложением
- operator -= (перегрузка); // присвоение с вычитанием
- operator \*= (перегрузка); // присвоение с умножением
- operator /= (перегрузка); // присвоение с делением
- operator %= (Т аргумент); // присвоение с остатком
- operator = (Т аргумент); // присвоение элемента (влияет на все элементы)

### **Функции класса:**

- Sum; // возвращает сумму элементов массива (тип double)
- Avg; // возвращает среднее элементов массива (тип double)
- Min; // возвращает минимальное значение массива (тип T)
- Max; // возвращает максимальное значение массива (тип T)
- Median; // возвращает центральный элемент массива (тип T)
- GetElement (size\_t индекс элемента); // возвращает искомый элемент
- Search (перегрузка); // поиск совпадений со значением

```
inline bool Search(T v){ ... }  
inline bool Search(initializer_list<T> v){ ... }
```

- Equals (List<T> массив); // булева проверка на равность массивов
- ToCharArray (string текст); // превращает строку в символьный массив (работает only с char)
  - Split (string текст, char символ разделения); // разбивает строку на подстроки (работает only со string)
  - ToString; // преобразование массива в строку

#### Методы класса (манипуляция с массивами):

- Sort; // сортировка элементов массива
- Converse; // разворот массива
- Inversion; // инверсия каждого элемента массива
- RandInt (int начало диапазона, int конец диапазона); // случайное заполнение
- RandDouble (double начало диапазона, double конец диапазона); // случайное заполнение
- Fill (T значение); // заполнение элементом
- FillPart (T значение, int начало диапазона, int конец диапазона); // заполнение элементом в диапазоне
  - Change (T искомое, T значение); // изменение элемента
  - Union (List<T> массив); // объединение массивов
  - Copy (перегрузка); // копирование массива (дублирование)

```
List& Copy(){ ... }  
List& Copy(int length){ ... }
```

- Add (перегрузка); // добавление элемента

```
List& Add(T v){ ... }  
List& Add(initializer_list<T> p){ ... }
```

- AddFirst (перегрузка); // добавление элемента в начало массива
- RemoveFirst; // убрать первый элемент массива
- RemoveLast; // убрать последний элемент массива
- NewList (перегрузка); // перезапись массива

```
List& NewList(initializer_list<T> p){ ... }  
List& NewList(List<T> p){ ... }
```

- DeleteByIndex (перегрузка); // удаление элементов из массива

```
List& DeleteByIndex(int index){ ... }  
List& DeleteByIndex(int start, int end){ ... }
```

- DeleteByArgs (перегрузка); // удаление по списку аргументов

```
List& DeleteByArgs(T arg){ ... }  
List& DeleteByArgs(initializer_list<T> args){ ... }  
List& DeleteByArgs(List<T> args){ ... }
```

- CutStart (int длина); // обрезка массива с начала  
- CutEnd (int длина); // обрезка массива с конца  
- Unique; // удаление повторяющихся значений  
- Addition (List<T> массив); // сложение массивов  
- Subtract (List<T> массив); // вычитание массивов  
- Multiplication (List<T> массив); // умножение массивов  
- Divide (List<T> массив); // деление массивов

### Прогрессии:

- ProgressionA (double начальное значение, int количество шагов, double прирост шага); // арифметическая прогрессия  
- ProgressionG (double начальное значение, int количество шагов, double прирост шага); // геометрическая прогрессия

### Конвертация

- ToVector // конвертация в vector массив  
- ToTypedPL\_List (массив) // конвертация контейнера в List

# ПРИЛОЖЕНИЕ FunctionalLIB\_String

## Разделение класса:

- операторы и стандартные функции;
- функции класса;
- манипуляции со строками.

## Операторы и стандартные функции:

- оператор <<; // вывод через cout класса String
- оператор >>; // ввод через cin
- оператор []; // индексация String
- оператор + (перегрузка); // сложение строк

```
String& operator+(const char* str) { newSTR += str; return *this; } // Str + "text"
String& operator+(string str) { newSTR += str; return *this; } // Str + str
String& operator+(String text) { string txt = text.ToString(); newSTR += txt.c_str(); return *this; } // Str + Str
template <typename T> String& operator+(T arg) { newSTR += to_string(arg); return *this; } // Str + T
```

- оператор += (перегрузка); // сложение строк

```
String& operator+=(const char* str) { newSTR += str; return *this; } // Str += "text"
String& operator+=(string str) { newSTR += str; return *this; } // Str += str
String& operator+=(String text) { string txt = text.ToString(); newSTR += txt; return *this; } // Str += Str
template <typename T> String& operator+=(T arg) { newSTR += to_string(arg); return *this; } // Str += T
```

- оператор = (перегрузка); // присваивание строк

```
String& operator=(const char* str) { newSTR = str; return *this; } // Str1 = "text"
String& operator=(char symbol) { newSTR = symbol; return *this; } // Str[0] = 'S'
String& operator=(string str) { newSTR = str; return *this; } // Str1 = str2
template <typename T> String& operator=(T arg) { newSTR = to_string(arg); return *this; } // Str1 = T
```

- operator - (перегрузка); // урезание, удаление, delete text

```
String& operator-(int len){ ... }
String& operator-(char symbol){ ... }
String& operator-(const char* text){ ... }
String& operator-(string text){ ... }
String& operator-(String text){ ... }
```

- operator / (int коэффициент урезания); // урезание строки, где длина / на коэффициент

- operator ==, !=, >, <, <=, >= // операторы сравнения

```
bool operator!=(const String& str) { return newSTR != str.newSTR; }
bool operator==(const String& str) { return newSTR == str.newSTR; }
bool operator>=(const String& str) { return newSTR >= str.newSTR; }
bool operator>(const String& str) { return newSTR > str.newSTR; }
bool operator<=(const String& str) { return newSTR <= str.newSTR; }
bool operator<(const String& str) { return newSTR < str.newSTR; }
```

- Length; // длина строки

- Clear; // очистка строки

- Print (перегрузка); // вывод строки без cout

```
void Print() const{ ... }
void Print(const char* text) const{ ... }
void Print(string text) const{ ... }
void Print(String text) const{ ... }
```

- ToString (перегрузка); // конвертировать в строку (стандартное string)

- ToTypedPL\_String (string строка); // конвертировать в строку (class String)

```
string ToString() const{ ... }
template <typename T> string ToString(T arg){ ... }
template <typename T> string ToString(initializer_list<T> arg){ ... }
template <typename T> string ToString(List<T> arg){ ... }
```

### Функции класса:

- GetElement (int индекс); // получить символ по индексу

- GetFirstElement; // получение первого символа строки

- GetLastElement; // получение последнего символа строки

- Search (перегрузка); // поиск на символ

```
bool Search(char symbol){ ... }
bool Search(string text){ ... }
bool Search(initializer_list<char> arg){ ... }
bool Search(List<char> arg){ ... }
```

- SearchCount (перегрузка); // вывод количества совпадений

```

int SearchCount(char symbol){ ... }

int SearchCount(string text){ ... }

int SearchCount(initializer_list<char> arg){ ... }

int SearchCount(List<char> arg){ ... }

```

- Equals (string строка) ИЛИ (String строка) // сравнение строк
- EqualsMax (string строка) ИЛИ (String строка) // сравнение строк на максимум
- EqualsMin (string строка) ИЛИ (String строка) // сравнение строк на минимум
- ToCharArray; // разбиение строки на массив символов
- Split; // разбиение строки на подстроки

#### **Манипуляции со строками:**

- NewString; // переназначение строки
- Reverse; // переворот строки
- Replace; // замена символов в строке
- Add (перегрузка); // добавление символов

```

String& Add(string arg){ ... }

String& Add(char symbol){ ... }

template <typename T> String& Add(initializer_list<T> arg){ ... }

```

- AddSymbol (char символ); // добавление символа
- SwapString (String строка); // замена строк
- CutStart (int длина); // обрезка с начала
- CutEnd (int длина); // обрезка с конца
- CutRange (int начало, int конец) // обрезка по диапазону
- Remove (перегрузка); // удаление значений

```

String& Remove(char symbol){ ... }

String& Remove(initializer_list<char> arg){ ... }

String& Remove(List<char> arg){ ... } // delete symbol

```

```

String& Remove(String text){ ... }

String& Remove(string text){ ... } // delete text

```

- RemoveSymbol (char символ); // удаление символа
- Remove\_FirstSymbol; // удаление первого символа
- Remove\_LastSymbol; // удаление последнего символа
- RemoveA\_z; // удаление английских символов

- RemoveA\_я; // удаление русских символов
- RemoveNum; // удаление чисел
- RemoveAnother; // удаление спец символов
- ExceptA\_z; // удалить все, КРОМЕ английских символов
- ExceprA\_я; // удалить все, КРОМЕ русских символов
- ExceptNum; // удалить все, КРОМЕ чисел
- Regular\_Remove (string выражение); // удаление по выражению
- Regular\_Replace (string выражение, string значение замены); // замена по выражению

# ПРИЛОЖЕНИЕ FunctionalLIB\_File

## 1. Чтение и запись файлов

- ReadAllLines; // считывание файла построчно (return List<string>)
- ReadAllText; // считывание файла целиком (return string)
- ReadAllLines\_Vector // построчное считывание для вектора (return vector<string>);
- WriteAllText (перегрузка); // запись текста в файл

```
template <typename T> File& WriteAllText(T text) { ... }  
template <typename T> File& WriteAllText(T text, string format) { ... }
```

- WriteAllLines (перегрузка); // запись массива данных в файл

```
template <typename T> File& WriteAllLines(List<T> arr) { ... }  
template <typename T> File& WriteAllLines(vector<T> arr) { ... }  
template <typename T, size_t N> File& WriteAllLines(T(&arr)[N]) { ... }
```

- AppendAllText (аналогичная Write перегрузка); // запись текста без перезаписи
- AppendAllLines (аналогичная Write перегрузка); // запись массива без перезаписи

## 2. Действия над файлами

- FileOpen; // открыть файл
- FileCreate; // создание файла
- FileRemove; // удаление файла
- FileMove (string newPath); // перемещение файла
- FileRename (string newName); // переименование файла

## 3. Получение информации о файле

- GetBitSize; // получить размер файла в битах
- GetByteSize; // получить размер файла в байтах
- GetKBSIZE; // получить размер файла в килобайтах
- GetMBSIZE; // получить размер файла в мегабайтах
- GetGBSIZE; // получить размер файла в гигабайтах
- GetFileName; // получить имя файла
- GetFilePath; // получить путь к файлу
- GetFileExtension; // получить расширение файла
- GetFileDepth; // получить глубину файла

# **ПРИЛОЖЕНИЕ FunctionalLIB\_CustomConsole**

## **Список компонента:**

- шаблонные методы и методы вне ConsoleSetting;
- методы класса ConsoleSetting;
- класс Animation.

## **Шаблонные методы и методы вне ConsoleSetting:**

- ThisBold (шаблон T); // *жирный текст (влияет глобально на стиль)*
- ThisNormal (шаблон T); // *обычный текст (влияет глобально на стиль)*
- ThisLine (шаблон T); // *подчеркивание (дополняет текущий стиль)*
- ThisRGB (шаблон T, целое R, целое G, целое B); // *раскраска текста по RGB*
- ThisForegroundRGB (шаблон T, целое R, целое G, целое B); // *раскраска фона текста по RGB*
- Clear; // *очистка содержимого консоли*
- Close; // *закрытие консоли*

## **Методы класса ConsoleSetting:**

- WindowShow (bool); // *показать (скрыть) рамку*
- WindowPanelShow (bool); // *показать (скрыть) панель кнопок*
- WindowMinShow (bool); // *показать (скрыть) кнопку сворачивания консоли*
- WindowMaxShow (bool); // *показать (скрыть) кнопку растягивания консоли*
- WindowCloseShow (bool); // *показать (скрыть) кнопку закрытия консоли*
- Icon (string path); // *установка иконки*
- ScrollBarShow (bool); // *показать (скрыть) полосу прокрутки*
- FullScreen; // *отобразить на весь экран*
- Resize (bool); // *разрешить (запретить) растягивание*
- PositionConsole (целое X, целое Y); // *установить позицию консоли на экране*
- Size (целое X, целое Y); // *установить размер консоли*
- MaxSize (целое X, целое Y); // *установить максимально допустимый размер*
- GetConsoleSize\_X; // *возвращает значение размеров консоли по X*
- GetConsoleSize\_Y; // *возвращает значение консоли по Y*
- Title (const char\* Text); // *установить заголовок приложения*
- Bold; // *распространяется на весь текст консоли*
- Line; // *дополняется на весь текст*
- Normal; // *нормализует весь текст*
- Opacity (double opacity (0 - 1)); // *прозрачность консоли*

- PositionCursor (целое X, целое Y); // позиция курсора внутри консоли
- GetCursorPosition\_X; // функция возвращает позицию курсора по X
- GetCursorPosition\_Y; // функция возвращает позицию курсора по Y
- FontSize (целое Size); // установка размеров текста
- FontFamily (string Name); // установка семейства (шрифта) текста
- TextBG\_Color; // устанавливает цвет текста по цвету background консоли
- GetRedValue; // функция возвращает значение красного из RGB
- GetGreenValue; // функция возвращает значение зеленого из RGB
- GetBlueValue; // функция возвращает значение синего из RGB
- TextColor (перегрузка); // цвет всего текста

```
ConsoleSetting& TextColor(string hexColor){ ... }
```

```
ConsoleSetting& TextColor(int color){ ... }
```

- ForegroundColor (перегрузка); // фон всего текста

```
ConsoleSetting& ForegroundColor(int R, int G, int B){ ... }
```

```
ConsoleSetting& ForegroundColor(string hexColor){ ... }
```

- BackgroundColor (перегрузка); // фон консоли

```
ConsoleSetting& BackgroundColor(int R, int G, int B){ ... }
```

```
ConsoleSetting& BackgroundColor(string hexColor){ ... }
```

- ChangeEncode; // изменить кодировку консоли
- GetEncode; // получить значение текущей кодировки
- SizePixel (int X, int Y); // установка размеров консоли в пикселях
- GetConsoleSizePixel\_X; // получение размеров консоли в пикселях по X
- GetConsoleSizePixel\_Y; // получение размеров консоли в пикселях по Y

### Класс Animation:

- AnimationText (строка, посимвольная задержка); // анимация текста  
*(последовательное появление)*
- ConsoleFadeIn (длительность анимации); // появление консоли
- ConsoleFadeOut (длительность анимации); // исчезновение консоли
- Console\_LeftToRight (X начальное, Y начальное); // анимация слева направо
- Console\_RightToLeft (X начальное, Y начальное); // анимация справа налево
- Console\_UpToDown (X начальное, Y начальное); // анимация сверху вниз
- Console\_DownToUp (X начальное, Y начальное); // анимация снизу вверх
- Console\_CustomAnimate (X начальное, X конечное, Y начальное, Y конечное, время анимации, направление анимации) // кастомная настройка анимации



## ПРИЛОЖЕНИЕ GeometryLIB\_Figure

**Таблица 1. Периметры фигур**

№	Функция	Фигура	Параметры
1	Triangle2D_P_MiddleLine	Треугольник	a, b, c
2	Triangle2D_P_TwoSide	Треугольник	a, b, angle
3	Triangle2D_P_Isosceles	Треугольник равнобедренный	a, b
4	Triangle2D_P_Isosceles_HeightAndMain	Треугольник равнобедренный	a, h
5	Triangle2D_P_Isosceles_HeightAndSide	Треугольник равнобедренный	a, h
6	Square2D_P	Квадрат	a
7	Square2D_P_Diagonal	Квадрат	d
8	Square2D_P_FromS	Квадрат	s
9	Square2D_P_CircleOut	Квадрат	r
10	Square2D_CircleIn	Квадрат	r
11	Rentangle2D_P	Прямоугольник	a, b
12	Rentangle2D_P_Diagonal	Прямоугольник	a, d
13	Parallelogram2D__P	Параллелограмм	a, b
14	Parallelogram2D_P_Diagonal	Параллелограмм	a, d1, d2
15	Parallelogram2D_Corner	Параллелограмм	a, h, angle
16	Rhombus2D_P	Ромб	A
17	Rhombus2D_P_Diagonal	Ромб	d1, d2
18	CircleHole2D_P	Кольцо	r, R
19	Polygon2D_P	Правильный многоугольник	a, n

**Таблица 2. Площадь**

1	Triangle2D	Треугольник	a, h
2	Triangle2D_TwoSide	Треугольник	a, b, angle
3	Triangle2D_AllSide	Треугольник	a, b, c
4	Triangle2D_CircleOut	Треугольник	a, b, c, r
5	Triangle2D_CircleIn	Треугольник	a, b, c, r
6	Triangle2D_Equilateral_AllSide	Треугольник равносторонний	a
7	Triangle2D_Equilateral_CircleOut	Треугольник равносторонний	r
8	Triangle2D_Equilateral_CircleIn	Треугольник равносторонний	r
9	Triangle2D_Isosce;es_AllSide	Треугольник равнобедренный	a, b
10	Square2D	Квадрат	a
11	Square2D_Diagonal	Квадрат	d
12	Rentangle2D	Прямоугольник	a, b
13	Rentangle2D_Diagonal	Прямоугольник	a, d
14	Rentangle2D_DiagonalAndCorner	Прямоугольник	d, angle
15	Parallelogram2D	Параллелограмм	a, h
16	Parallelogram2D_Corner	Параллелограмм	a, b, h
17	Parallelogram2D_DiagonalAndCorner	Параллелограмм	d1, d2, angle
18	Rhombus2D	Ромб	a, h
19	Rhombus2D_Diagonal	Ромб	d1, d2
20	Rhombus2D_Corner	Ромб	a, angle
21	Trapezoid2D	Трапеция	a, b, h
22	Trapezoid2D_AllSide	Трапеция	a, b, c, d
23	Trapezoid2D_Diagonal	Трапеция	d1, d2, angle
24	Trapezoid2D_MiddleLine	Трапеция	m, h
25	Circle2D	Круг	r
26	Circle2D_Diameter	Круг	d
27	Circle2D_CircleLength	Круг	L
28	Circle2D_CirclePart	Круг	r, angle
29	CircleHole2D	Кольцо	r, R

30	CircleHole2D_Diameter	Кольцо	d, D
31	Polygon2D_Apophemera	Правильный многоугольник	S, A
32	Polygon2D_Side	Правильный многоугольник	a, n
33	Polygon2D_CircleIn	Правильный многоугольник	r, n
34	Polygon2D_CircleOut	Правильный многоугольник	r, n
35	Hexagon2D_Side	Шестиугольник	a
36	Hexagon2D_CircleOut	Шестиугольник	r
37	Hexagon2D_CircleIn	Шестиугольник	r
38	Hexagon2D_Diagonal	Шестиугольник	d
39	Hexagon2D_ShortDiagonal	Шестиугольник	d
40	Hexagon2D_Perimetr	Шестиугольник	P
41	Octagon2D_Side	Восьмиугольник	a
42	Octagon2D_CircleOut	Восьмиугольник	r
43	Octagon2D_CircleIn	Восьмиугольник	r

**Таблица 3. Объемы**

1	Triangle3D	Тетраэдр	S, h
2	Triangle3D_Equilateral	Правильный тетраэдр	a
3	Cone3D	Конус	r, h
4	Cone3D_Part	Конус	r1, r2, h
5	Square3D	Куб	a
6	Rentangle3D	Прямоугольный параллелепипед	a, b, h
7	Parallelogram3D	Параллелепипед	S, h
8	Cilinder3D	Цилиндр	r, h
9	Hexagon3D	Правильный шестиугольник	a, h
10	Circle3D	Шар	r
11	Circle3D_Diameter	Шар	d

12	Circle3D_CircleLength	Шар	L
13	CircleHole3D	Кольцо	h, r
14	Pyramid3x3D	Пирамида	a, h
15	Pyramid4x3D	Пирамида	a, h
16	Pyramid5x3D	Пирамида	a, h
17	Pyramid6x3D	Пирамида	a, h
18	PyramidNx3D	Пирамида	a, h, n
19	Prism3x3D	Призма	a, h
20	Prism4x3D	Призма	a, h
21	Prism5x3D	Призма	a, h
22	Prism6x 3D	Призма	a, h

**Таблица 4. Площадь поверхности**

1	Square3D_S_Side	Куб	a
2	Square3D_S_Diagonal	Куб	d
3	Rentangle3D_S_Side	Прямоугольник	a, b, h
4	Circle3D_S_Radius	Шар	r
5	Circle3D_S_Diagonal	Шар	d
6	Cone3D_SPart	Конус	r, L
7	Cone3D_SFull	Конус	r, L
8	Cilinder3D_SPart	Цилиндр	r, h
9	Cilinder3D_SFull	Цилиндр	r, h

**Таблица 5. Иные формулы**

1	LengthCircle_Radius	L окружности	r
2	LengthCircle_Diameter	L окружности	d
3	LengthCircle_FromS	L окружности	S
4	Figure2D_Points	Любая фигура	List coordinate
5	Figure2D_Sides	Любая фигура	List sides

## ПРИЛОЖЕНИЕ GeometryLIB\_Expression

### Class ComplexExpression:

- operator +=;
- operator +;
- operator -;
- operator \*;
- operator /;
- Complex\_Addition (ComplexExpression комплексное число); // +
- Complex\_Subtract (ComplexExpression комплексное число); // -
- Complex\_Multiplication (ComplexExpression комплексное число); // \*
- Complex\_Divide (ComplexExpression комплексное число); // /
- Get\_A; // получить действительную часть
- Get\_B; // получить мнимую часть
- Get\_Expression; // получить выражение

### Class VectorExpression:

- operator +=;
- operator +;
- operator -;
- operator \* (перегрузка);

```
VectorExpression operator*(VectorExpression& newVector) { ... }
```

```
VectorExpression operator*(double k) { ... }
```

- Vector\_Addition (VectorExpression вектор); // сложение
- Vector\_Subtract (VectorExpression вектор); // вычитание
- Vector\_Multiplication (перегрузка); // умножение

```
VectorExpression Vector_Multiplication(VectorExpression& newVector) { ... }
```

```
VectorExpression Vector_Multiplication(double k) { ... }
```

```
VectorExpression Vector_VectorMultiplication(VectorExpression& newVector) { ... }
```

- Vector\_Length (длина вектора 1, длина вектора 2, угол между ними); // вне класса
- Vector\_ScalarMultiplication (длина вектора 1, длина вектора 2, угол между ними); //

*вне класса*

- Vector\_Length (перегрузка); // в классе

```
double Vector_Length() { ... }  
double Vector_Length(VectorExpression& newVector) { ... }
```

- Triangle2D\_VectorMultiplication (VectorExpression вектор); // площадь треугольника по векторному произведению

- Equals (VectorExpression вектор); // сравнение векторов
- EqualsMax (VectorExpression вектор); // сравнение на максимум
- EqualsMin (VectorExpression вектор); // сравнение на минимум
- Get\_Cos (VectorExpression вектор); // получить косинус угла между векторами
- Get\_X; // получить координату X
- Get\_Y; // получить координату Y
- Get\_Z; // получить координату Z
- Get\_Expression; // получить XYZ

## BETA -0.10 Version 2.0

**ЧТО НОВОГО?** Были изменены обращения к статическим массивам. Если для работы с ними ранее требовалось указывать их размерность непосредственно в аргументах, то сейчас данная надобность больше не нужна. Теперь многие массивы имеют следующий синтаксис:

```
template<typename T, typename... Args> double Sum(T arg, Args... args){ ... }

template<typename T> double Sum(vector<T> arr){ ... }

template<typename T, size_t N> double Sum(T(&arr)[N]){ ... }

template<typename T, size_t N> double Sum(array<T, N> arr){ ... }
```

Как видно, третья перегрузка Sum больше не содержит **T arr[], int size**. Теперь эта строка заменена на **T(&arr)[N]**. Size\_t N в шаблоне позволяет записывать автономно размерность массива в переменную N, с которой можно работать в логике функции. Таким образом, было сокращено количество аргументов во всех функциях статических массивов.

### **Были добавлены новые функции для FunctionalLIB Array.**

**1) Функция Lgth** – возвращает целочисленное значение размерности массива. Преимущественно, новшества вводят только в работу со статическими массивами, но также применим и с vector и array массивами.

**Шаблон функции выглядит следующим образом:**

```
template<typename T> int Lgth(vector<T>& arr){ ... }

template<typename T, size_t N> int Lgth(T(&arr)[N]){ ... }

template<typename T, size_t N> int Lgth(array<T, N>& arr){ ... }
```

Как видно, принимает только один параметр – обрабатываемый массив.

**Пример применения:**

```
struct MyStruct
{
    vector<string> str{
        "X??1", "DF_21", "*2_k",
        "^1", "Gg%__", "<St&le>",
        "W_w", "J00J00"
    };
};

MyStruct arrStruct;
cout << "Длина массива: " << Lgth(arrStruct.str);
```

**2) Функция Add** – позволяет добавлять к уже существующим и заполненным массивам дополнительные элементы. Возможно замена аналогичной функцией push\_back в случаях использования vector массивов.

**Функция имеет следующий шаблон:**

```
template<typename T> vector<T> Add(vector<T>& arr, T value){ ... }

template<typename T, size_t N> vector<T> Add(T(&arr)[N], T value){ ... }

template<typename T, size_t N> array<T, N + 1> Add(array<T, N>& arr, T value){ ... }
```

Каждая перегруженная функция принимает параметры: тип массива, значение

**Пример применения:**

В зависимости от применяемого типа, может записываться по-разному.

Для статики:

```
typedef int INTEGER;
INTEGER arr[] = {1, 23, 1, 4, 1, 4};
for (auto s : Add(arr, 100))
    cout << s << "\n";
```

Для array<type, size>:

```
array<double, 2> arrDOUBLE = { 0.5, 1.8 };
const int size = sizeof(arrDOUBLE) / sizeof(arrDOUBLE[0]);
array<double, size + 1> result = Add(arrDOUBLE, 2.4);
for (auto s : result)
    cout << s << "\n";
```

Для динамики:

```
vector<int> arr0 = { 0, 1 };
vector<int> arr0_1 = { 0, 1 };

for (int i = 0; i < Lgth(arr0_1); i++)
    Add(arr0, arr0_1[i]);

for (int out : arr0)
    cout << out << " ";
```

**3) Функция LINQ имитации** – функция позволяет сравнивать аргументы и массивы без необходимости создания условных конструкций. Имеет множество ограничений в сравнении с оригиналом из библиотеки NumSharp.

**Шаблоны функции:**

```
template<typename T> bool LINQ_OPERATION(T arg, string operation, T paramIF){ ... }

template<typename T> vector<T> LINQ_OPERATION(vector<T>& arr, string operation, T paramIF){ ... }
```

**Пример использования:**

```
vector<double> result = LINQ_OPERATION(arr0, ">=", 1.5);
result = LINQ_OPERATION(result, "<=", 5.5);
Sort(result);
for (auto s : result)
    cout << s << " ";
```

// сортировка диапазона > 1.5 && < 5.5

## BETA -0.10 Version 3.0

**ЧТО НОВОГО?** Было решено разделить функционал библиотеки на несколько частей. Данное решение было связано с тем, что модули имели противоположное друг от друга назначения (один связан с геометрией, а другой изменяет структуру работы с массивами). **Теперь модули библиотеки подключаются следующим образом:**

```
#include "Geometry.h"
#include "FPFC.h"
```

Geometry.h – пространство имен GeometryLIB\_Figure (без изменений)

FPFC.h – пространство имен FunctionalLIB\_Array (**new!**)

– пространство имен FunctionalLIB\_CustomConsole (**new!**)

*// По сути просто вместо одного заголовочного файла стало два, но без изменения функционала данных компонентов (GeometryLIB\_Figure и FunctionalLIB\_Array).*

В связи с этим было решено переименовать библиотеку под typedPL (типизированный язык программирования), поскольку название Geometry являлось не совсем корректным отображением использования библиотеки (хотя и данное название не совсем подходит – можете порекомендовать более нормальное название).

**ИСПРАВЛЕНО!** Была исправлена работа с типом данных string. Многие методы вызывали сбой при работе с типом данных string. Происходил конфликт с возвращаемым параметром, а также были ошибки, когда не находилась перегрузка с аргументом типа string.

```
_Param_(1)" может быть равен "0": это не соответствует спецификации функции
"std::basic_string<char, std::char_traits<char>, std::allocator<char> >::{ctor}".
```

```
отсутствуют экземпляры перегруженная функция "Fill", соответствующие списку аргументов
"Fill": не найдена соответствующая перегруженная функция
```

**Поэтому были добавлены новые перегрузки для некоторых методов, среди которых:**

- Sort;
- Converse;
- Change;
- Fill;
- FillPart;
- Add;
- LINQ\_OPERATION.

**Новые перегрузки имеют данный шаблон:**

```
string Converse(vector<string>& arr){ ... }

template<size_t N> string Converse(string(&arr)[N]){ ... }

template<size_t N> string Converse(array<string, N>& arr){ ... }
```

Как видно, из динамического массива был убран шаблон как таковой, теперь перегрузка имеет исключительно тип `string`. Для статики и `array` массива в качестве шаблона осталась только длина массива `N`.

## **ДОБАВЛЕНЫ НОВЫЕ ФУНКЦИИ К МОДУЛЮ GEOMETRY!**

Были добавлены две функции, которые позволяют находить площадь любой фигуры, исходя из их координат или длине сторон. Если ранее были предоставлены функции, которые работали исключительно с каким-то типом фигур (самым многофункциональными были функции нахождения правильный многоугольников), то теперь можно находить площадь различных неправильных фигур.

### **Figure2D\_Points**

```
using namespace GeometryLIB_Figure;
cout << Figure2D_Points({
    1,5,
    3,1,
    7,4
});
```

Данная функция возвращает площадь фигуры, рассматривая в качестве параметров координаты вершин фигуры. Важно отметить, что данная функция поддерживает проверку на четность параметров, что позволит избежать ошибок вычислений при нехватке каких-то данных. Также, не обязательно, но желательно использовать формат записи, как показано на рисунке выше, поскольку при наличии многогранных фигур будет сложнее разобрать записи, написанные в одну строку.

### **Figure2D\_Sides**

```
using namespace GeometryLIB_Figure;
cout << Figure2D_Sides({
    10,15,13,17
});
```

Позволяет вычислять площадь разных фигур по значениям сторон фигуры. Может напоминать некоторые функции, предоставленные в прошлых версиях, но имеет расширенные возможности, поскольку данная функция может работать с неограниченным количеством сторон.

## **НОВЫЙ КОМПОНЕНТ К МОДУЛЮ FPFC!**

Был реализован новый модуль (FunctionalLIB\_CustomConsole), который позволит менять некоторые настройки консоли. **Среди главных функций:**

- изменения параметров рамки (размеры, свойства, название);
- изменение параметров текста (его положение, цвет, задний фон).

**Важная особенность:** наличие возможности связывать элементы компонента между собой.

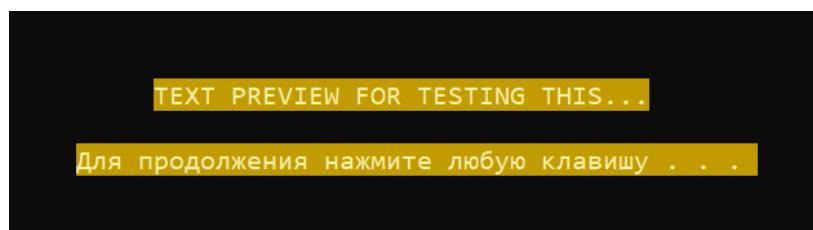
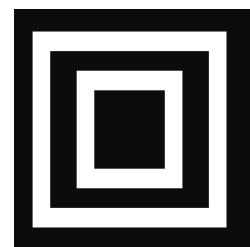
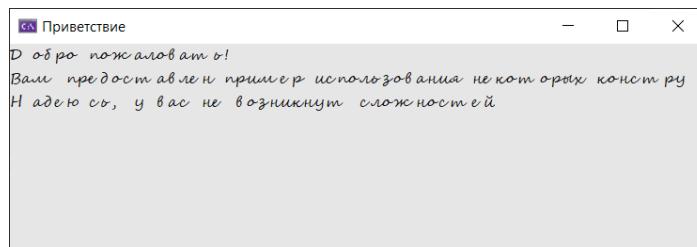
```
static void SETTING(ConsoleSetting config) {  
    config.FontFamily("Times New Roman").Bold();  
    // пример цепочки методов  
  
    bool cfg = false;  
    config.Resize(false).  
        ScrollBarShow(cfg).  
        WindowPanelShow(0);  
    // пример цепочки методов с разными вариациями записи bool  
    // данную запись в столбец можно использовать для удобства чтения  
}
```

Данная возможность появилась благодаря использованию конструкторов класса и функциями, возвращающим значения этого класса, что и позволяет объединять функции между собой. Теперь для подключения к элементам нужно объявлять экземпляр класса ConsoleSetting:

```
ConsoleSetting cfg; // экземпляр класса  
using namespace FunctionalLIB_CustomConsole; // пространство имен
```

*P.S. (Однако, конструкторы не работают с элементами вне класса, поэтому подобная реализация не доступна в FunctionalLIB\_Array).*

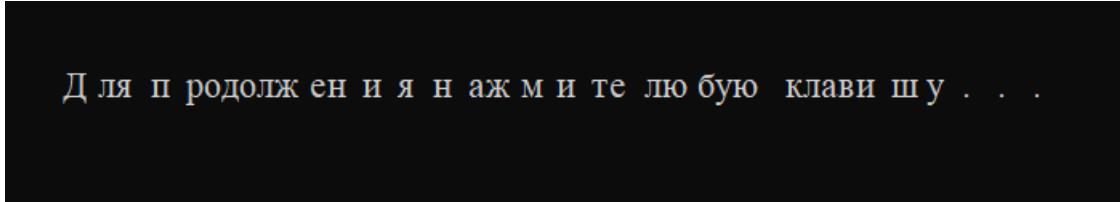
### **Некоторые скриншоты документации:**



Более подробную информацию можно найти в документации и в приложении.

## **БАГИ КОМПОНЕНТА (ДОПОЛНЯЙТЕ И РЕШАЙТЕ)**

### **1. ФОНТЫ... НЕ РАЗБЕГАЙТЕСЬ!**

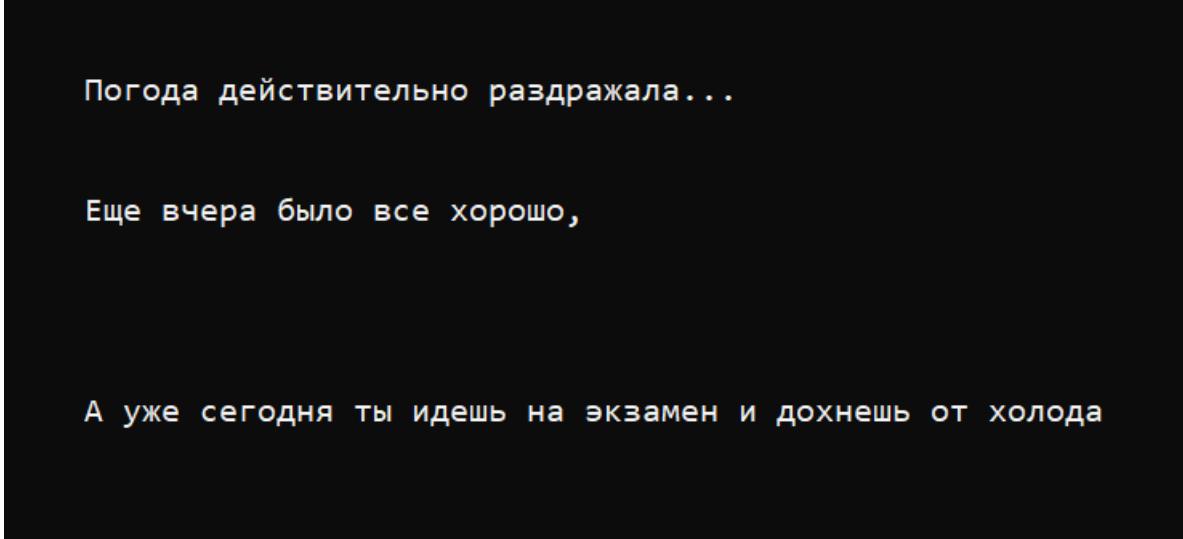


Для продолжения нажмите любую клавишу . . .

При использовании `FontFamily` регулярно происходят подобные выводы. Есть шрифты, в которых данные пробелы минимальны. А есть экземпляры, как показано на картинке выше.

Проблема заключается, скорее всего, в использовании русских символов, поскольку те же предложения на «инглиш лангуаге» отображаются нормально.

### **2. ПРОБЕЛ ВАМ ЧТО – ОТСТУП?**



Погода действительно раздражала...

Еще вчера было все хорошо,

А уже сегодня ты идешь на экзамен и дохнешь от холода

Данный скриншот уже прилагался в документации, а также была озвучена проблема: при вводе строки (независимо от того, что в логике координата Y смещается на +1) появляются отступы между записями. **Данные отступы равнозначны количеству пробелов в предложении** (пробелы именно между символами, если поставить кучу пробелов к ряду – они будут считаться, как 1).

### **3. А «Y» ТО ПОБОЛЬШЕ БУДЕТ**

При использовании равнозначного значения X и Y – отступы по высоте явно превышают длину.

**Возвращаясь к примеру с квадратами:**



// квадрат 15 x 15

Подобное встречается и при использовании других методов. Например, если использовать установку `Size`. Если установить значение по `Y` более 50 – то вряд ли у вас действительно измениться консоль (скорее всего, просто появится горизонтальный `ScrollBar`).

**Данный казус возникает из-за того, что высота символа больше, чем его длина:**



На данной картинке показано два и один символ. Глядя на второй прямоугольник, можно увидеть, как он по высоте в 2 раза больше.

Для устранения, предположительно, придется искать максимально разумный коэффициент, на который будет умножаться переменная `X` в логике метода.

#### 4. ДАЖЕ КОНСОЛЬ СБЕГАЕТ ОТ ТЕБЯ

Проблема возникает при попытке создать плавное переливание цветов фона консоли (может есть и еще подобные «схлопывания», но я находил только этот).

```
ConsoleSetting con;

for (int i = 0; i < 255; i++)
{
    con.BackgroundColor(i, i, i);
    Sleep(50);
} // проблемный код
```

**Поведение консоли:** (будет прикреплено видеофайлом)



WTF.mp4

## BETA -0.10 Version 3.2

### ЧТО НОВОГО В CONSOLE?

Было добавлено несколько новых возможностей в пространство имен FunctionalLIB\_CustomConsole. Появилось несколько методов, позволяющие вытягивать данные из параметров консоли.

```
enum FunctionLIST {
    GetRedValue, // возвращает R, G, B из RGB системы от background консоли
    GetGreenValue,
    GetBlueValue,

    GetConsoleSize_X, // возвращает длину консольного окна по X и Y
    GetConsoleSize_Y,

    GetCursorPosition_X, // возвращает позицию курсора по X и Y
    GetCursorPosition_Y
};
```

Эти функции возвращают определенное значение, поэтому не могут составлять цепочки методов, однако, могут быть их замыкающими:

```
ConsoleSetting con{};
int sizeX = con.Icon("ico.ico").GetConsoleSize_X();
// несмотря на Icon, sizeX будет ссылаться на последний метод GetConsole
```

Соответственно, цепочка закончилась на методе GetConsoleSize\_X и продолжаться не будет.

Помимо этого, было добавлено несколько методов для ConsoleSetting: Icon и TextBG\_Color. Первый метод позволяет изменять икону приложения (не работает с большинством форматов изображений, необходим .ico). TextBG\_Color же позволяет изменять цвет текста, ориентируясь на цвет фона консоли. Данный метод можно использовать, к примеру, для скрытия текста, или для перекраски без использования перегрузок TextColor.

#### **Класс Animation в FunctionalLIB\_CustomConsole:**

Был добавлен новый небольшой класс для анимации некоторых элементов (на данный момент существует только три метода). Подробнее о классе и его методах можно найти в приложении или документации выше.

Пример обращения к классу и использование его методов:

```
string str; cin >> str;

Animation Animation;
Animation.AnimationText(str, 100).ConsoleFadeOut(3000);
    // анимация текста      // анимация консоли
```

## КОНСТАНТЫ ДЛЯ GEOMETRY!

Была добавлена небольшая директива в качестве аналога \_USE\_MATH\_DEFINES с расширенным функционалом.

```
// МАТЕМАТИКА
#define PI 3.14159265358979323846264338327950288 // число ПИ
#define _PI_2_ 6.283185307179586 // число 2 ПИ
#define I = -1 // минимая единица
#define _SQRT_2_ 1.41421356237309504880168872420969808 // корень 2
#define _SQRT_3_ 1.73205080756887729352744634150587237 // корень 3
#define _SQRT_5_ 2.23606797749978969 // корень 5
#define E 2.7182818284590452353602874713526625 // число Эйлера (константа Непера)
#define _LG_2_ 1.44269504088896340736 // логарифм 2
#define _LG_10_ 0.434294481903251827651 // логарифм 10
#define _LN_2_ 0.693147180559945309417232121458 // натуральный LOG 2
#define _LN_10_ 2.30258509299404568401799145468 // натуральный LOG 10
#define M 0.57721566490153286060651209008240243 // Маскерони (постоянная Эйлера)
#define LPS 0.6627434193491815897474209710925290 // предел Лапласа
#define _GOLD_R_ 1.61803398874989484820458683436563812 // золотое сечение
// ВЫСШАЯ МАТЕМАТИКА
#define APERI 1.20205690315959428539973816151144999 // Апери
#define KINKLEN 1.2824271291006226368753425688697917 // постоянная Глейшера – Кинкелина
#define CATALAN 0.915965594177219015054603514932384110774 // постоянная Каталана
#define CAENA 0.64341054629 // константа Каэна
#define DOTTY 0.73908513321516064165531208767387340 // число Дотти
#define DICKMAN 0.624329988543550870992936 // постоянная Голубма (Дикмана)
#define SOLDNER 1.451369234883381050283968485892027 // константа Солднера
#define TREFETENA 0.70258 // константа Эмбри (Трефетена)
#define TWINS 0.66016181584686957392781211001455577 // константа простых близнецов
#define TWINS_B 1.902160583104 // константа простых близнецов по Бруно
#define F 4.66920160910299067185320382046620161 // постоянная Фейгенбаума
#define KHINCHIN 2.685452001065 // постоянная Хинчина
```

## ИСПРАВЛЕНО! Были исправлены некоторые недочеты в GeometryLIB\_Figure.

В частности, были решены проблемы с неправильным счетом некоторых функций из-за отсутствия перевода единиц измерений в радианы (касается функций, где в качестве параметра принимался угол). Таким образом были исправлены все подобные **функции**, а также **было добавлено две новые** Pyramid3x3D (объём треугольной пирамиды), PyramidNx3D (объем пирамиды с неизвестным количеством сторон).

## FUNCTIONAL – ЧТО-ТО НОВОЕ?

Появился новый класс List, который позволяет создавать произвольный динамический массив (то есть массив будет не vector или статический, а list).

### **Синтаксис нового массива:**

```
List<double> MyArray = { 0, 4.2, -5.33, TWINS_B };

List<size_t> SIZE{0,0,1,0};

List<string> _STR_ ({"4", "строчка"});
```

Данный массив содержит в себе многие методы, что используются библиотекой typedPL для векторов и статики.

Важным отличием от предыдущей версии является полное переосмысление структуры. Теперь **методы можно объединять в цепочки**, как это было сделано в ConsoleSetting.

Один из примеров использования:

```
List<double> MyArray = { 0, 4.2, -5.33, TWINS_B };
MyArray.Inversion()
    .Sort()
    .Converse(); // сортировка с поворотом позволяет
    // сделать сортировку по убыванию

for (double Out : MyArray.Union(MyArray)) // дублирование
    cout << Out << " ";
```

Благодаря тому, что данный класс использует иной подход к реализации – **была автономно решена проблема с типом данных string** (смотреть в сноска «ИСПРАВЛЕНО» от BETA -0.10 Version 3.0). Это означает, что не было необходимости в создании излишних string перегрузок.

Также **методы класса List удобнее использовать**. В частности, превосходную работу можно заметить, работая с методами класса в циклах.

К примеру:

```
List<int> arr = { 0,1,5,2,1 };
vector<int> vec = { 0,1,5,2,1 };

for (auto s : arr.Sort())
    cout << s << " "; // пример list массива

for (auto s : Sort(vec))
    cout << s << " "; // пример vector массива // разница в использовании в
циклах
```

В данном примере сортировка векторного массива недоступна, поскольку отсутствует функция begin в диапазоне for. Поэтому сначала придется отсортировать массив вне цикла.

**Появилась возможность использовать операторы** для работы с массивом и его элементами.

Например, операции над всеми элементами:

```
List<int> INT = { 0,10,100 };
INT = INT + 30, INT / 10; // сложение всех на 30
// деление всех на 10 // ответы (3, 4, 13)
```

## **НОВЫЙ КОМПОНЕНТ ДЛЯ СТРОК!**

Было добавлено новое пространство имен FunctionalLIB\_String, который отвечает за работу со строками. В данном компоненте вы можете видоизменять строки, конвертировать их в массивы для дальнейшей работы с ними, а также оперировать новыми функциями, которых нет в стандарте C++ и string.

**Подключение:**

```
#include "FPFC.h"
using namespace FunctionalLIB_String;
```

**Инициализация строки класса String:**

```
using namespace FunctionalLIB_String;

String str = "пример инициализации переменной класса";

String str2("EXAMPLE 2");

string text = "text for 3";
String str3 = text.c_str();

FunctionalLIB_String::String str4;
```

По своей структуре во многом схож с классом List. Некоторые методы напрямую заимствованы у этого класса. Некоторые методы из пространства <string> были упрощены для достижения более комфортной работы с классом string.

При нехватке какого-либо функционала – строки можно свободно конвертировать из String в string - и наоборот.

```
String str_class = "class";
string str_stand("standart");

str_stand = str_class.ToString(); // конвертирование в string
str_class = str_stand.c_str(); // конвертирование в String
```

### **ВЗАЙМОДЕЙСТВИЕ STRING и LIST.**

Несмотря на то, что классы находятся в разных пространствах имен – они неплохо резонируют друг с другом. То есть с ними легко работать, сочетая их друг с другом.

К примеру:

```
List<String> arr = { "str1", "str2", "str3" };
arr.Add({ "str4", "str5" }).Converse(); // перевернет массив
for (String out : arr)
    cout << "Элемент: " << out << "\n";

String str = arr[0] + " " + arr[arr.Count() - 1];
str.Print(); // вывод строки из массива

string str2 = arr[arr.First()].ToString();
// получение string из массива String типа
```

## **НОВЫЙ КОМПОНЕНТ ДЛЯ GEOMETRY!**

Был добавлен новый компонент для работы с выражениями. В частности, вы можете работать с комплексными числами или векторами (на данный момент это все, но в будущем планируется добавление алгоритмов алгебраической и геометрической последовательности, а также новый структур).

Подключение к пространству имен происходит по имени GeometryLIB\_Expression.

Компонент содержит два класса:

- ComplexExpression;
- VectorExpression.

Оба класса содержат перегрузку операторов (хоть и немного кривую), что позволяет производить операции между структурами напрямую.

## **БАГИ? ЧТО, ОПЯТЬ!?**

Еще одна сноска с багами, с которыми пришлось столкнуться при разработке нового компонента и класса List. Здесь также будут собраны проблемы, которые присутствуют в стандарте C++.

### **1. БУЛЕВЫ РЕАЛЬНО БУЛЯТ (РЕШЕНО)**

Проблема заключается в том, что при попытке присвоить к элементу булевого типа какое-либо значение – возникает ошибка.

```
List<String> strArray(10);
strArray[2] = "Okay"; // присвоение к третьему элементу массива

List<bool> boolArray({ true });
boolArray[0] = false; // ошибка в присвоение
```

 C2440 return: невозможно преобразовать  
"std::\_Vb\_reference<std::\_Wrap\_alloc<std::allocator<std::seed\_seq::result\_type>>" в "T &"

Ошибка срабатывает только при использовании индексации для булевого типа данных. К тому же при использовании vector<bool> подобных проблем и ошибок не возникает.

**Причина:** данная ошибка предположительно срабатывает из-за неправильного использования оператора индексации (operator []). Возможно следует изменить конструкцию данного оператора.

```
T& operator[](int index) { return value[index]; } // считывание массива по индексам
```

**Решение:** была изменена основная переменная типа vector на deque (двухсторонняя очередь). Deque во многом похож на вектора, но позволяет добавлять и удалять элементы не только с конца, но и с начала. Проблемой данной ошибки была в том, что bool не могла перейти по ссылке, когда в deque подобной проблемы нет.

## 2. А «СТРИНГ» ТО В «СТРИНГИ» НЕ ХОТИТ (РЕШЕНО)

Проблема возникает при попытке использовать метод `ToString` на таком же типе данных. Метод `ToString` преимущественно используется для конвертации типа без наличия дополнительных аргументов. Но в некоторых случаях он может использоваться, как добавочный элемент (как функция `Add` или operator `+`).

```
List<String> arr = { "s", "t", "r", "i", "n", "g" };
string str = arr.ToString(); // ошибка конвертирования

List<char> arrCHAR = { 's', 't', 'r', 'i', 'n', 'g' };
string str = arrCHAR.ToString(); // без ошибки

String text = "text";
string txt = text.ToString({ "2" }); // ошибка конвертирования
string txt2 = text.ToString({ 222 }); // без ошибки
// ошибка с массивами
```

massivami возникает *только при String*, а базовый *string* выдает без ошибок

Выдает одну из следующих ошибок:

- ✖ C2679 бинарный "+=": не найден оператор, принимающий правый operand типа "const \_Ty" (или приемлемое преобразование отсутствует)
- ✖ C2088 +=: недопустимо для class
- ✖ C2665 "std::to\_string": ни одна перегруженная функция не может преобразовать все типы аргументов

Комментариями показаны, какие блоки выдают ошибки – при добавлении через `ToString` текста – выдается ошибка.

**Причина:** возможно связана с параметром шаблоном `T`, который не является однозначным для конвертирования. Однако с целочисленными данными функция хорошо работает.

**Решение:** была переписана часть с неправильной конвертацией типа. Скорее всего, дело было в том, что конвертация происходила непосредственно в цикле, что приводило к неправильной конвертации. Теперь конвертация в нужный тип происходит сразу при вызове метода, а затем просто складываются строки к строкам.

**ПРОСЬБА:** проверить метод `ToString` на работу с разными типами.

## 3. ВРОДЕ CHAR – А ВРОДЕ И НЕ CHAR (РЕШЕНО)

Нежелательно конвертировать `char` в другие типы данных, поскольку результаты могут быть довольно неожиданными. Рассмотрим пример с тем же `ToString`:

```
string txt2 = text.ToString({'X', 'D', 'z'});
// 1953658213886890 результат // с List<char> работает okay
```

Конвертирование `char` порой может вызывать много вопросов. Также, при работе с операторами – конвертирование `char` довольно проблематично:

```
String a, b;  
  
a = a.operator+(2).operator+(3).operator+(1); // 231  
b = b.operator+'2'.operator+'3'.operator+(1); // 50511  
  
a = '2' + '1'; // 99 // было
```

### исправлено

Если вы посмотрите на перегрузки операторов в приложении (именно с перегрузкой `T`), то вы увидите – что все данные конвертируются в тип `string`. И в первом случае все работает правильно, но при `char` вы сами видите ответ.

```
char a = 3213; // число?  
char b = false; // булева?  
char c = 'text'; // длинное символьное?
```

К тому же использование `char` неоднозначно. Мы можем задавать разные значения и при этом получать разные результаты (в случае примера это «`? {пустота} t»).`

**Причина:** проблема, как и в предыдущем пункте, может заключаться в неправильном конвертировании данных. В случае `ToString` конвертация типа `char` была исправлена.

## 4. КАК ВВОДИТЬ ЭТОТ РУССКИЙ ЛАНГУАГЕ (РЕШЕНО (СМ. `ConsoleSetting`)!?)

Данная проблема относится не столько к библиотеке – сколько к проблемам языка. При попытке ввода русских символов и последующем его выводе происходит следующая ложа:

```
string str;  
cin >> str;  
cout << str; привет? Опять иероглифы!?...  
                                ЇаЇў?в?Для продолжения нажмите любую клавишу . . .
```

С теми же английскими символами такого не происходит. Опять же, говоря о плохой поддержанности стандартом C++ русских символов, можно обратиться к сноске багов из прошлого обновления библиотеке (ФОНТЫ... НЕ РАЗБЕГАЙТЕСЬ!).

**Причина:** загвоздка заключается в использовании кодировки. Было перепробовано множество разных методов использования `setlocale` или похожих по функционалу функций – но результат печальный.

## BETA -0.10 Version 3.5

**ЧТО НОВОГО?** Были добавлены операторы для класса String, которые позволяют работать с классом внутри условий.

**Список операторов:** ==, !=, <, >, <=, >=.

```
string s; cin >> s;
String str = ToTypedPL_String(s);
if (str.Length() > 10)
    cout << "Вы ввели слишком длинное сообщение";
str.Length() <= 5 ? str.Add(0) : str.Remove_LastSymbol(); // пример использования новых
opersatorov для создания условий
```

Помимо этого, произошли небольшие изменения внутри класса List. Был изменен метод Inversion. Теперь данный метод представляет из себя шаблон, работающий только с арифметическими типами данных (int, double, float...).

```
template <typename X = T> // инверсия численных
typename enable_if<is_arithmetic<X>::value, List<X>::type Inversion() {
    for (int i = 0; i < value.size(); i++) {
        if (value[i] != 0)
            value[i] = -value[i];
    }
    return *this;
} // шаблон метода Inversion
```

Ранее могла возникнуть ошибка при использовании метода с типами, которые не являются числовыми типами. Сейчас же данный метод также выдает ошибку при попытке использовать не числовой тип, но компилятор выдает более понятную для пользователя ошибку еще до компиляции:

```
отсутствуют экземпляры шаблон функции "FunctionalLIB_Array::List<T>::Inversion [c T=std::string]", // ошибка записи не числового
соответствующие списку аргументов
```

типа к Inversion

**В ConsoleSetting добавились метод ChangeEncode и функция GetEncode.** ChangeEncode позволяет изменять кодировку консоли (происходит замена с cp1251 на cp866, и наоборот). GetEncode позволит узнать текущую кодировку в случае необходимости. Смена кодировки позволит избежать неправильного отображения русских символов, введенных пользователем, на консольном пространстве.

**Более точное измерение размера консоли.** Теперь вы можете задавать размеры консоли не в символьном эквиваленте, а задавать в качестве параметра пиксели. Для этого появились методы SizePixel, а также группа Get функций для получения размеров в пикселях. Обычный метод Size будет не всегда удобно использовать (поскольку при взаимодействии с FontSize результат всегда будет неправильный (и ряд других причин)), поэтому для более комфортной работы, вы можете использовать SizePixel.

**Переход на C++20** Осуществлен переход со стандарта C++14 на C++20. В связи с переходом на новый, более современный стандарт языка, были исправлены некоторые ошибки, которые возникали в новой версии.

В частности, был изменен алгоритм конвертации типа `string` в `wstring` через `codecvt`, а также исправлена неопределенность для типа `byte` в `Windows.h`.

### **Новый компонент FPFC**

Добавлен компонент для FPFC, который позволяет взаимодействовать с файлами. Это небольшой модуль, который предоставляет базовые функции, которые помогут работать с файлами: перезаписывать информацию, вытягивать данные, создавать свои форматы файлов.

Для подключения пространства имен требуется прописать `using namespace FunctionalLIB_File`. Для явного обращения к методам и функциям компонента можно обойтись и без этой записи, соответственно.

**Завершение поддержки работы с массивами типа `array`, `vector`, статики.** Были удалены все методы для обработки стандартных массивов C++. Это связано с тем, что данный блок кода был мало оптимизирован, а также имел бесполезный функционал в сравнении с классом `List`. Удаленные блоки кода были помечены в документации, как «не поддерживается».