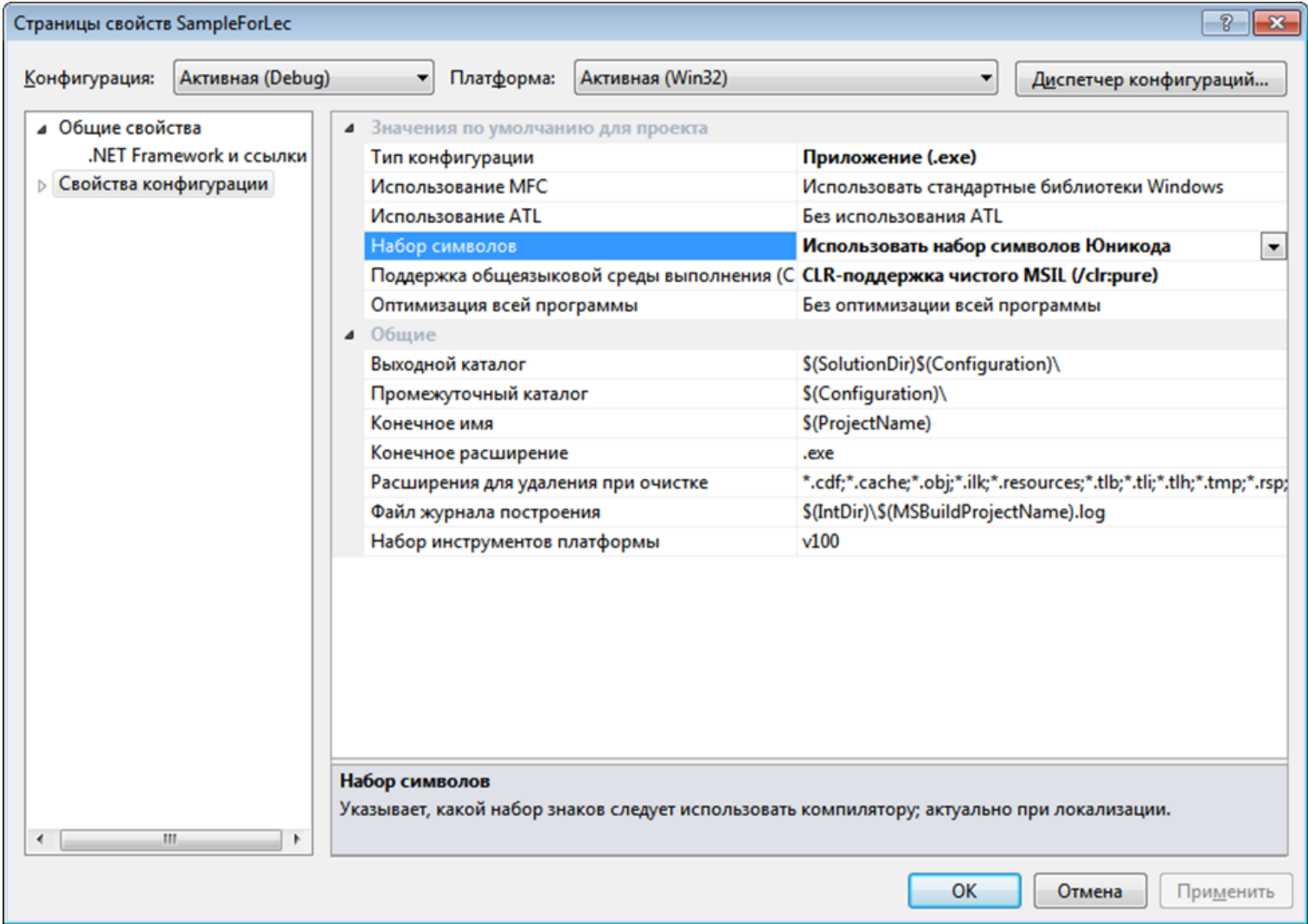


Лекция 8

Символьные и строковые данные в C++

8.1 Символьные и строковые данные, объявление и инициализация

В прошлом семестре мы рассматривали представление символьной информации в памяти ПК и различные системы ее кодировки (ASCII и Unicode). В языке C++ доступны обе эти системы. Выбор той или иной кодировки для конкретного проекта производится на **Странице свойств проекта** (свойство *Набор символов*).



8.1 Символьные и строковые данные, объявление и инициализация

Для хранения одиночных символов в языке C++ используется тип `char`. Константа или переменная такого типа занимает 1 байт в кодировке ASCII и 2 байта в кодировке Unicode. Переменная типа `char` может быть объявлена следующим образом:

```
char symbol;
```

где *symbol* – имя переменной.

При объявлении символьная переменная может быть проинициализирована символьной константой, заключаемой в одинарные кавычки (апострофы), например,

```
char symbol = 'a';
```

8.1 Символьные и строковые данные, объявление и инициализация

В программах могут использоваться символьные массивы, которые объявляются и инициализируются подобно числовым массивам, например,

```
char arr[8] = {'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'};
```

Однако гораздо чаще при работе с текстовыми данными программисту приходится иметь дело с так называемыми *строками* – символьными массивами, последний элемент которых всегда содержит нулевой символ '`\0`' – признак конца строки.

При объявлении строки необходимо учитывать наличие в конце строки нуль-символа, и отводить дополнительный байт под него. Так, при объявлении строки

```
char str[10];
```

следует учитывать, что в ней можно поместить не более 9 символов, так как последний должен быть зарезервирован под нуль-символ.

8.1 Символьные и строковые данные, объявление и инициализация

Строка при объявлении может быть инициализирована начальным значением — строковой константой, состоящей из набора символов, заключенного в двойные кавычки. Например:

```
char str[7] = "Строка";
```

при этом в памяти компьютера она будет выглядеть так:

С	т	р	о	к	а	\0
---	---	---	---	---	---	----

При объявлении строки с одновременной инициализацией не обязательно указывать её размер, например,

```
char str[] = "Строка";
```

В этом случае размер строки определяется автоматически, с учетом добавляемого в конец строки нуль-символа.

8.1 Символьные и строковые данные, объявление и инициализация

Объявлять и инициализировать строки можно также с использованием указателей, например,

```
char *str = "Строка";
```

Во всех ранее рассмотренных примерах **str** – указатель на 1-й символ строки, но в примерах с объявлением в виде массива это *указатель-константа* (его значение не может быть изменено), а в последнем случае – *указатель-переменная*.

8.1 Символьные и строковые данные, объявление и инициализация

Строка может быть объявлена как *именованная константа* с помощью модификатора **const**. При этом следует иметь в виду, что объявление вида:

```
const char * str;
```

говорит о том, что константой является объект, адресуемый указателем **str**, а объявление вида:

```
char * const str;
```

говорит о том, что константой является сам указатель, а объект может быть изменен. Например,

```
const char * str = "Строка";  
str[4] = 'ф';           // ошибка  
str = "Строфа"; // верно  
char * const str = "Строка";  
str[4] = 'ф';           // верно  
str = "Строфа"; // ошибка  
  
const char * const str = "Строка";  
str[4] = 'ф';           // ошибка  
str = "Строфа"; // ошибка
```

8.1 Символьные и строковые данные, объявление и инициализация

Выделение памяти под строку может производиться динамически, с помощью операции **new**, например,

```
char * cp = new char[81];
```

Освобождение выделенной памяти производится с помощью операции **delete**, например,

```
delete[] cp;
```

Современные версии языка C++ располагают рядом специальных *классов* для работы со строками. Использованию одного из них будет посвящена часть сегодняшней лекции и очередной лабораторной работы. Рассмотренные же ранее примеры строк называют строками в стиле C, так как в этом языке отсутствовали иные возможности обработки строк.

8.2 Основные приемы и функции обработки строк в стиле C

8.2.1. Доступ к символам строки

После определения строки указатель на нее содержит адрес первого символа. Доступ к произвольному символу строки может осуществляться как по индексу (нумерация начинается с нуля), указываемому внутри квадратных скобок, так и с использованием адресной арифметики. Например, следующие два выражения являются эквивалентными:

```
char st[] = "String";  
.....  
st[1]           // st[1] содержит символ t  
*(st + 1)       // *(st + 1) содержит символ t
```

К символу строки можно не только получить доступ, но и изменить его значение:

```
char st[] = "String";  
.....  
st[0] = 's';           // Изменение с помощью индекса  
*(st + 1) = 'T';       // Изменение с помощью указателя
```

8.2.1. Доступ к символам строки

Для последовательного доступа к символам строки удобно использовать операторы цикла **while** и **for**. В следующем фрагменте программы сначала определяется размер входной строки **source**, затем выделяется динамическая память под строку-копию **dest** и выполняется посимвольное копирование строки **source** в строку **dest**:

```
//.....  
char *source = "Карл у Клары украл кораллы";  
int n = 0;                                // n – длина строки  
while (*(source+n) != '\0')               // подсчет  
    n++;                                  // длины строки  
char *dest = new char[n+1];               // строка-копия  
for (int i = 0; i <= n; i++)              // посимвольное  
    *(dest+i) = *(source+i);             // копирование  
//.....  
delete[] dest;
```

Здесь следует обратить внимание на то, что в цикле **for** условием окончания цикла является **i <= n**, и в строку **dest** копируются все символы строки **source**, включая нуль-символ.

8.2.1. Доступ к символам строки

Эта же задача может быть решена с использованием адресной арифметики следующим образом:

```
//.....  
char *source = "Карл у Клары украл кораллы";  
int n = 0;                                // длина строки  
char *p = source;                         // указатель на исходную строку  
for (; *p; n++, p++);                     // подсчет длины строки в цикле  
char *dest = new char[n+1];               // строка-копия  
// посимвольное копирование  
for (p = p-n; *p; *dest = *p, dest++, p++);  
*dest = '\0';                             // нуль-символ для конца строки  
dest -= n;                                // указатель dest на начало строки  
//.....  
delete[] dest;
```

8.2.2. Использование системной библиотеки строковых функций

Для работы со строками в стиле C имеется библиотека многочисленных системных функций, прототипы которых определены в заголовочном файле `<string.h>`. Так, например, длина строки `str` без учета нуль-символа может быть определена с помощью функции `strlen`:

```
int n = strlen(str);
```

а копирование строки `str1` в строку `str2` может быть выполнено с использованием функции `strcpy`:

```
strcpy(str2, str1);
```

8.2.2. Использование системной библиотеки строковых функций

С использованием этих функций задача копирования из предыдущего параграфа может быть решена следующим образом:

```
//.....  
char *source = "Карл у Клары украл кораллы";  
int n = strlen(source);  
char *dest = new char[n+1];  
strcpy(dest,source);  
//.....  
delete[] dest;
```

Полный перечень прототипов функций библиотеки можно найти в справочных материалах в интернете. Использование этих функций значительно упрощает обработку строк, и поэтому их надо активно использовать при решении соответствующих задач.

8.2.3. Массивы строк

При решении некоторых задач приходится иметь дело не с отдельными строками, а с *массивами строк*.

Массив строк можно объявить, как двумерный символьный массив, например,

```
char num[5][7] = {"один", "два", "три", "четыре", "пять"};
```

Здесь первая размерность [5] указывает на количество строк в массиве, а вторая [7] – на максимальную длину строки с учетом нуль-символа. Объявленный массив в памяти компьютера будет иметь следующий вид:

о	д	и	н	\0	\0	\0
д	в	а	\0	\0	\0	\0
т	р	и	\0	\0	\0	\0
ч	е	т	ы	р	е	\0
п	я	т	ь	\0	\0	\0

8.2.3. Массивы строк

Для ссылки на одну из строк массива можно использовать имя массива с первым индексом, например `num[3]`. Однако удобнее работать с массивом строк как с массивом указателей:

```
char *num = { "один", "два", "три", "четыре", "пять"};
```

При таком объявлении резервируется память под 5 указателей `char *`, и они инициализируются адресами строковых констант из списка. Это экономнее, так как не расходуется лишняя память для коротких строк.

При динамическом выделении памяти под массив строк последовательность действий та же, что и при работе с числовыми двумерными массивами.

8.2.4. Виртуальный тип данных TCHAR

В рассмотренных выше примерах для объявления символьных и строковых переменных использовался тип данных **char**. Однако этот тип данных может быть использован лишь при однобайтовой кодировке **ASCII**. Если в настройках проекта установлен набор символов **Unicode**, то символьные и строковые данные должны объявляться с типом **wchar_t**.

Чтобы не переписывать программы при изменении кодировки символов, в C++ с помощью директивы **typedef** определен виртуальный тип данных **TCHAR**, являющийся надстройкой над типами данных **char** и **wchar_t**. Это определение находится в заголовочном файле **<tchar.h>** вместе с многочисленными определениями и так называемыми *макросами*. Включив заголовочный файл **<tchar.h>** в свой проект и используя в своих функциях тип **TCHAR** вместо **char**, вы сделаете свои приложения независимыми от установленной для конкретного проекта кодировки.

Следует иметь в виду, что при использовании типа **TCHAR** вместо рассмотренных в 8.2.2. библиотечных строковых функций следует использовать их аналоги, определённые в заголовочном файле **<tchar.h>**. Так, например, вместо функции **strcpy** используется функция **wscpy_s**.

8.3 Строки CLR и класс *System::String*

При работе с проектами, созданными в среде *Visual Studio* в шаблоне *CLR Windows Forms*, текстовые поля элементов управления формы передаются в пользовательские функции в виде строк класса *System::String*. Чтобы избежать лишних громоздких преобразований таких строк в рассмотренные выше строки в стиле C, будем далее изучать и применять методы обработки строк именно этого класса.

Строковый тип в *CLR System::String* относится к ссылочным (управляемым) типам и предназначен для хранения строк текста переменной длины. Особенностью значений этого типа является тот факт, что эти значения хранятся в динамической памяти и являются неизменяемыми. При попытке изменения значения строки, в действительности в динамической памяти создается новый ее экземпляр с измененными данными, что не влияет на значение исходной строки.

Текст строки хранится в виде неизменяемой, доступной только для чтения, последовательности объектов *System::Char* в коде *Unicode*. Максимальный размер строки класса *String* в памяти составляет 2 Гб или более 1 миллиарда символов.

Номер позиции объекта *Char* в строке определяется *индексом*, который является неотрицательным числом и равен нулю для первой позиции в строке. Индексы используют различные методы класса *String* для поиска отдельных символов и подстрок в строке, их копирования и сравнения.

8.3.1. Объявление и инициализация строк класса *String*.

Основные свойства класса

Так как строковый тип класса **String** относится к ссылочным типам, объявление строки этого класса похоже на объявление указателя и отличается лишь тем, что вместо символа ***** используется символ **^**. Например,

```
String ^str ; // Объявление строки str
```

При объявлении строка может быть инициализирована, например,

```
String ^s1 = "873"; // s1 = "873"
```

Инициализация пустой строкой может быть выполнена двумя способами: строковой константой и с использованием свойства **Empty**, например,

```
String ^str1 = "";  
String ^str2 = String::Empty;
```

8.3.1. Объявление и инициализация строк класса *String*. Основные свойства класса

Длина строки может быть определена с использованием свойства **Length**, например,

```
String ^s1 = "abcdef";  
int len;  
len = s1->Length; // len = 6  
s1 = "";  
len = s1->Length; // len = 0
```

Обратите внимание на синтаксис ссылки на свойство строки (->).

Присвоить строке значение другой строки можно как при инициализации, так и в операторе присваивания:

```
String ^s1 = "My String";  
String ^s2 = s1;           // s2 = "My String"  
  
// ИЛИ  
String ^s1 = "My String";  
String ^s2;  
s2 = s1;                   // s2 = "My String"
```

8.3.1. Объявление и инициализация строк класса *String*. Основные свойства класса

Определить, равны ли две строки между собой, можно путем их непосредственного сравнения в операторе **if**. Например,

```
String ^s1 = "Hello!";  
String ^s2 = "Hello!";  
int res;  
if (s1 == s2)  
    res = 1;  
else  
    res = 0;
```

Строки класса **String** могут объединяться в массивы строк, при этом используется класс **array**, пример работы с которым будет приведен далее.

8.3.2. Основные методы класса *String* для работы со строками

Для сравнения двух строк в лексикографическом порядке может быть использован метод **Compare**, возвращающий значение типа **int** в зависимости от результата сравнения: **-1**, если **s1<s2**; **0**, если **s1=s2**; **1**, если **s1>s2**, например,

```
String ^s1 = "Hello!";  
String ^s2 = "Hello";  
int res;  
res = s1->Compare(s1,s2); // res = 1  
res = s2->Compare(s2,s1); // res = -1  
res = s2->Compare(s1,s2); // res = 1  
s2 = s1;  
res = s1->Compare(s1,s2); // res = 0  
res = s1->Compare(s2,s1); // res = 0
```

Обратите внимание на то, что в качестве текущей строки (т.е. строки слева от знака ссылки **->**) при вызове метода может быть использована любая из сравниваемых строк.

8.3.2. Основные методы класса *String* для работы со строками

Для сцепления (конкатенции) двух строк может быть использован метод **Concat**:

```
String ^s1 = "Hello ";  
String ^s2 = "world!";  
String ^s3;  
s3 = s1->Concat(s1,s2); // s3 = "Hello world!"
```

Как и в случае с методом **Compare**, при вызове метода может быть использована любая из сцепляемых строк. Результат сцепления зависит от порядка аргументов метода, например,

```
String ^s1 = "Hello ";  
String ^s2 = "world!";  
String ^s3;  
s3 = s1->Concat(s2,s1); // s3 = "world!Hello "
```

8.3.2. Основные методы класса *String* для работы со строками

Сцепление строк может быть произведено и без использования метода **Concat**, с помощью операции конкатенции (знак операции +):

```
String ^s1 = "Hello ";  
String ^s2 = "world!";  
String ^s3;  
s3 = s1 + s2;  // s3 = "Hello world!"
```

Копирование строки может быть выполнено с помощью метода **Copy**, например,

```
String ^s1 = "MyString";  
String ^s2;  
s2 = s1->Copy(s1);    // s2 = "My String"  
  
// Однако проще обойтись обычным присваиванием  
  
String ^s1 = "My String";  
String ^s2;  
s2 = s1;              // s2 = "My String"
```

8.3.2. Основные методы класса *String* для работы со строками

Вставка подстроки в строку производится с помощью метода **Insert**. Метод имеет два параметра: первый из них – номер (индекс) позиции в строке, начиная с которой делается вставка (нумерация начинается с 0); второй – вставляемая подстрока, например:

```
String ^s1 = "123789";  
String ^s2;  
s2 = s1->Insert(3, "456"); // s2 = "123456789"  
s1 = "123789";  
s2 = s1->Insert(0, "456"); // s2 = "456123789"
```

Превышение позиции вставки длины строки приводит к ошибке компиляции с выдачей соответствующего сообщения.

8.3.2. Основные методы класса *String* для работы со строками

Поиск и возвращение индекса первого вхождения подстроки в данную строку выполняется с помощью метода **IndexOf**. Если заданная подстрока найдена, метод возвращает позицию ее первого вхождения в строку, иначе возвращается **-1**.

Метод имеет перегруженные варианты реализации. В первом варианте метод имеет один единственный параметр – подстроку для поиска, и поиск ведется с начала строки. Во втором варианте добавляется второй параметр – номер начальной позиции строки, с которой начинается поиск. В третьем варианте используется еще один, третий параметр – количество позиций для поиска. Использование метода во всех трех вариантах иллюстрируется следующим фрагментом программы:

```
String ^s1 = "Hello world!";  
int index;  
index = s1->IndexOf("wor"); // index = 6  
index = s1->IndexOf("ab"); // index = -1  
index = s1->IndexOf("wor",7); // index = -1 – поиск из позиции 7  
index = s1->IndexOf("wor",0); // index = 6 – поиск из позиции 0  
// поиск идет от заданного индекса (3)  
// и проверяется заданное количество символов (5)  
index = s1->IndexOf("wo",3,5); // index = 6  
index = s1->IndexOf("wor",3,5); // index = -1
```

8.3.2. Основные методы класса *String* для работы со строками

Поиск и возвращение индекса последнего вхождения подстроки в данную строку выполняется с помощью метода **LastIndexOf**. Метод отличается от **IndexOf** лишь тем, что поиск ведется с конца строки. Например,

```
String ^s1 = "text-text-text";  
int index;  
index = s1->LastIndexOf("text"); // index = 10
```

8.3.2. Основные методы класса *String* для работы со строками

Методы **PadLeft** и **PadRight** *дополняют строку до заданной длины* слева или справа, соответственно, пробелом или заданным символом. Если заданная длина меньше либо равна длине исходной строки, то добавление не производится.

Методы имеют два перегруженных варианта реализации. В первом варианте методы имеют один единственный параметр – длину результирующей строки. Во втором варианте добавляется второй параметр – символ, которым дополняется строка. Использование методов в обоих вариантах иллюстрируется следующим фрагментом программы:

```
String ^s1 = "abc";  
String ^s2;  
s2 = s1->PadLeft(5); // s2 = "  abc"  
s2 = s1->PadLeft(2); // s2 = "abc"  
s2 = s1->PadLeft(5, '*'); // s2 = "***abc"  
  
s2 = s1->PadRight(3); // s2 = "abc"  
s2 = s1->PadRight(8); // s2 = "abc      "  
s2 = s1->PadRight(8, '*'); // s2 = "abc*****"  
  
s2 = (s1->PadLeft(6, '*'))->PadRight(9, '*'); // s2 = "***abc***"
```

8.3.2. Основные методы класса *String* для работы со строками

Удаление заданного количества символов из строки производится методом **Remove**. Метод имеет два перегруженных варианта реализации. В первом варианте единственный параметр задает номер позиции, с которой начинается удаление символов до конца строки. Во втором варианте добавляется параметр, в котором задается максимальное количество удаляемых символов. Использование метода в обоих вариантах иллюстрируется следующим фрагментом программы:

```
String ^s1 = "0123456789";  
String ^s2;  
s2 = s1->Remove(3);    // s2 = "012"  
s2 = s1->Remove(3,2);  // s2 = "01256789"
```

8.3.2. Основные методы класса *String* для работы со строками

Замена всех вхождений в строку одной подстроки на другую выполняется методом **Replace**. В первом параметре метода задается первая (заменяемая) подстрока, во втором – вторая (заменяющая) подстрока. Если первый параметр содержит исходную строку, то производится ее замена на подстроку во втором параметре. Например,

```
String ^s1 = "Эти слова разделены двумя пробелами";  
String ^s2, ^s3;  
s2 = s1->Replace("  ", " "); // Два пробела всюду заменены на один  
s3 = s2->Replace(s2, "Эти слова разделены одним пробелом");  
// s3 = "Эти слова разделены одним пробелом"
```

8.3.2. Основные методы класса *String* для работы со строками

Метод **Substring** служит для *выделения подстроки* из строки. Метод имеет два перегруженных варианта реализации. В первом варианте единственный параметр задает номер позиции, с которой начинается выделение символов до конца строки. Во втором варианте добавляется параметр, в котором задается количество выделяемых символов. Использование метода в обоих вариантах иллюстрируется следующим фрагментом программы:

```
String ^s1 = "Automobile";  
String ^s2;  
s2 = s1->Substring(4);    // s2 = "mobile"  
s2 = s1->Substring(0,4); // s2 = "Auto"
```

8.3.2. Основные методы класса *String* для работы со строками

Метод **Split** возвращает строковый массив, содержащий подстроки текущей строки (слова), разделенные элементами заданного массива символов Unicode. Например, в следующем фрагменте программы сначала объявляется символьный массив **d1m**, содержащий символы–разделители слов: пробел, запятую и тире. Затем с использованием метода **Split** заданная строка **str** преобразуется в строковый массив **words**, состоящий из отдельных слов текста в строке **str**. Количество элементов массива равно **10**.

Оба массива являются объектами класса **array** – этого требует метод **Split**.

```
String ^str = "Напролёт целый год-гололёд, Будто нет ни весны, ни лета";  
array <TCHAR>^ d1m = {' ', ',', '-', ' '};  
array <String>^ words = str->Split(d1m);  
int len = words->Length;          // len = 10
```

В результате выполнения метода массив **words** будет заполнен следующими строковыми значениями:

```
words[0] = "Напролёт", words[1] = "целый", words[2] = "год",  
words[3] = "гололёд", words[4] = "Будто", words[5] = "нет",  
words[6] = "ни", words[7] = "весны", words[8] = "ни", words[9] = "лета"
```

8.3.2. Основные методы класса *String* для работы со строками

Метод **Join** *сцепляет* элементы массива строк, помещая их в одну строку и разделяя заданным символом. Таким образом, этот метод по выполняемой функции противоположен методу **Split**. Например,

```
array <String^>^ words={"Карл", "у", "Клары", "украл", "кораллы"};  
String ^str = str->Join(" ", words);
```

В результате выполнения метода строка **str** будет инициализирована следующим значением:

"Карл у Клары украл кораллы"

8.3.2. Основные методы класса *String* для работы со строками

Методы **ToLower** и **ToUpper**, не имеющие параметров, возвращают текущую строку, представленную, соответственно, строчными или прописными буквами.

Метод **Trim**, не имеющий параметров, удаляет из текущей строки все начальные и конечные пробелы.

Для преобразования строк класса **String** в арифметические типы данных и обратно можно воспользоваться методами **Parse** и **ToString**, иллюстрируемыми следующими фрагментами программы:

```
int i;
String ^s1 = "280";
i = i.Parse(s1);    // i = 280
i = i.Parse("-32"); // i = -32

i = 350;
s1 = i.ToString(); // s1 = "350"

double x;
s1 = "12,87";      // Разделитель – запятая!
x = x.Parse(s1);   // x = 12.87

x = -3.568;
s1 = x.ToString(); // s1 = "-3.568"
```

8.3.2. Основные методы класса *String* для работы со строками

Метод `TryParse` позволяет перед преобразованием строки в арифметические типы проверить допустимость строкового представления числа. Если число не является допустимым, то метод возвращает значение **false**. Это позволяет проанализировать возвращаемое логическое значение для недопущения ошибки преобразования недопустимого значения. Например, в следующем фрагменте программы производится ввод из текстового поля формы `txtX` значения вещественной переменной `x` типа **double**. Если поле `txtX` имеет недопустимый формат (например, точку в качестве разделителя целой и дробной части), то метод вернет значение **false**, и оператор `if` выведет сообщение об ошибке.

```
double x;  
if (!Double.TryParse(txtX.Text, x))  
    . . . Сообщение об ошибке и выход
```

При работе с данными типа **float** в вызове метода должен использоваться класс **Single**, при работе с данными типа **int** — класс **Int32**.

8.4 Типовые алгоритмы обработки строк и примеры их реализации

К типовым алгоритмам обработки строк можно отнести алгоритмы решения следующих задач:

- определение количества символов в строке при заданных условиях;
- замена или вставка символов в строке;
- удаление символов в строке;
- анализ символов на принадлежность к группе;
- подсчет количества заданных фрагментов текста;

Пример 8.4–1. Разработать функцию, которая подсчитывает, сколько раз заданный символ встречается в заданной строке.

```
int Sample1(String ^str, String ^ch)
{
    int n = 0;
    for (int i = 0; i < str->Length; i++)
        if (str->Substring(i,1) == ch) n++;
    return n;
}
```

Параметрами функции являются строковые переменные **str** и **ch**, содержащие заданную строку и заданный символ. В цикле **for** происходит выделение очередного символа с помощью метода **Substring**. Если выделенный символ совпадает с заданным, то значение счетчика **n** увеличивается на **1**. По окончании цикла накопленное значение **n** возвращается оператором **return**.

Пример 8.4-2. Разработать функцию, которая в заданной строке заменяет каждый пробел на заданный символ.

```
String ^Sample2(String ^str, String ^ch)
{
    str = str->Replace(" ", ch);
    return str;
}
```

Параметрами процедуры являются строковые переменные **str** и **ch**, содержащие заданную строку и заданный символ. Замена производится с помощью метода **Replace**.

Пример 8.4-3. Разработать функцию, которая из заданной строки удаляет все символы, совпадающие с заданным.

```
String ^Sample3(String ^str, String ^ch)
{
    int index;
    while ((index=str->IndexOf(ch)) >= 0)
        str = str->Remove(index, 1);
    return str;
}
```

Параметрами функции являются строковые переменные **str** и **ch**, содержащие заданную строку и заданный символ. В цикле **while** с помощью метода **IndexOf** происходит поиск позиции первого символа строки **index**, совпадающего с заданным. Пока такой символ находится, он удаляется методом **Remove**. Функция работает корректно даже в том случае, когда все символы строки совпадают с заданным. При этом она возвращает пустую строку.

Пример 8.4-4. Разработать функцию, которая подсчитывает, сколько раз заданная подстрока входит в заданную строку.

```
int Sample4(String ^str, String ^substr)
{
    int i = 0, j = 0, k = 0, n = substr->Length;
    do
    {
        j = str->IndexOf(substr, i);
        if (j >= 0)
        {
            k++;
            i = j + n;
        }
    }
    while (j >= 0);
    return k;
}
```

Пример 8.4-4. Разработать функцию, которая подсчитывает, сколько раз заданная подстрока входит в заданную строку.

Параметрами функции являются строковые переменные `str` и `substr`, содержащие заданную строку и заданную подстроку. В цикле `do-while` с использованием метода `IndexOf` ищется позиция `j` очередного вхождения подстроки в строку. Если она найдена (`j >= 0`), то счетчик вхождений `k` увеличивается на `1`, а начальная позиция `i` для следующего поиска смещается от `j` на длину подстроки `n`. Как только очередное вхождение не найдено (`j < 0`), происходит выход из цикла и возврат значения счетчика вхождений `k`.

Пример 8.4-5. Разработать функцию, которая подсчитывает количества четных и нечетных цифр в заданной строке.

```
void Sample5(String ^str, int &n1, int &n2)
{
    String ^Digit = "0123456789";
    String ^ch;
    int k;
    n1 = 0; n2 = 0;
    for (int i = 0; i < str->Length; i++)
    {
        ch = str->Substring(i,1);
        if (Digit->IndexOf(ch) >= 0)
        {
            k = k.Parse(ch);
            k % 2 == 0 ? n2++ : n1++;
        }
    }
}
```

Пример 8.4-5. Разработать функцию, которая подсчитывает количества четных и нечетных цифр в заданной строке.

Входным параметром функции является строковая переменная **str**, содержащая заданную строку. Выходными параметрами являются целые переменные **n1** и **n2**, передаваемые по ссылке. В них по завершению выполнения функции будут содержаться подсчитанные количества нечетных и четных цифр соответственно. В цикле **for** с помощью метода **Substring** выделяется очередной символ строки **ch**. Если этот символ входит в строку **Digit**, то есть является цифрой, то он преобразуется в целое числовое значение **k** с помощью метода **Parse**. Затем условная операция проверяет четность **k** и соответствующий счетчик **n1** или **n2** увеличивается на **1**.

8.5 Пример выполнения лабораторного задания

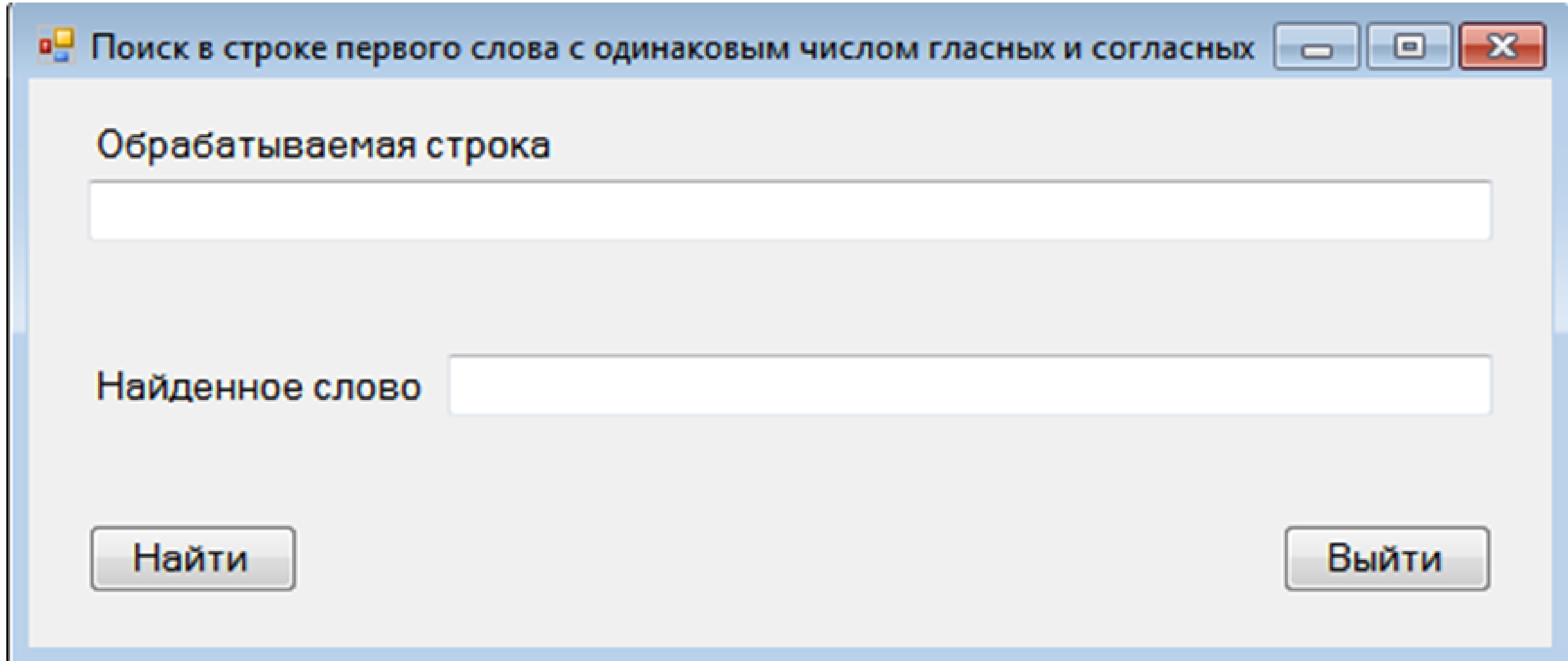
1. Задание. В заданной строке, состоящей из слов на русском языке, разделенных одним или несколькими пробелами, точками, запятыми и тире, найти и вывести первое в строке слово с равным количеством гласных и согласных букв.

2. Формализация задачи. Из условия задания следует, что его выполнение распадается на следующие этапы:

- 1) Ввод исходной строки.
- 2) Удаление из строки лишних символов с тем, чтобы все слова были разделены ровно одним пробелом.
- 3) Формирование из строки массива слов.
- 4) Поиск в массиве слов первого слова с равным количеством гласных и согласных букв.
- 5) Вывод найденного слова.

8.5 Пример выполнения лабораторного задания

3. Конструирование формы.



Поиск в строке первого слова с одинаковым числом гласных и согласных

Обрабатываемая строка

Найденное слово

Найти

Выйти

8.5 Пример выполнения лабораторного задания

3. Конструирование формы.

Форма содержит 6 объектов:

- 1) текстовое поле **txtString** для ввода обрабатываемой строки;
- 2) текстовое поле **txtWord** для вывода найденного слова;
- 3) кнопку **cmdFind** для запуска процедуры поиска;
- 4) кнопку **cmdExit** для выхода из приложения;
- 5) надписи **lblString** и **lblWord**, комментирующие назначение текстовых полей.

8.5 Пример выполнения лабораторного задания

4. Программный код проекта. Событийные процедуры

```
private: System::Void cmdFind_Click(System::Object^ sender, System::EventArgs^ e)
{
    String^ source = txtString->Text;
    if (source->Length==0)
    {
        MessageBox::Show("Введите строку", "Ошибка",
            MessageBoxButtons::OK, MessageBoxIcon::Error);
        txtString->Focus();
        return;
    }
    String^ res = Find(source);
    if (res->Length > 0)
        txtWord->Text = res;
    else
        MessageBox::Show("Слово не найдено. Введите другую строку", "Сообщение",
            MessageBoxButtons::OK, MessageBoxIcon::Information);
    txtString->Focus();
    txtString->SelectionStart = 0;
}
```

8.5 Пример выполнения лабораторного задания

4. Программный код проекта. Событийные процедуры

```
private: System::Void cmdExit_Click(System::Object^ sender, System::EventArgs^ e)
{
    this->Close();
}
private: System::Void Form1_Load(System::Object^ sender, System::EventArgs^ e)
{
    txtString->Text = "";
    txtWord->Text = "";
}
private: System::Void txtString_TextChanged(System::Object^ sender, System::EventArgs^ e)
{
    txtWord->Text = "";
}
```

8.5 Пример выполнения лабораторного задания

4. Программный код проекта. Событийные процедуры

Обратите внимание: в данном проекте реализован подход, отличный от применявшегося в предыдущих лабораторных работах. Ранее операции ввода–вывода реализовывались в соответствующих функциях, размещенных в отдельном файле. Здесь же ввод *единственного* исходного данного (обрабатываемой строки) и вывод *единственного* результата (найденного слова) выполняются непосредственно в событийной процедуре `cmdFind_Click`. В данном случае это вполне оправдано, и такой подход имеет право на существование наряду с применявшимся ранее.

8.5 Пример выполнения лабораторного задания

4. Программный код проекта. Заголовочный файл find.h

```
#include <tchar.h>
using namespace System;
String^ Find(String^);
String^ DelSpace(String^);
String^ GetWord(array<String^>^);
int GetVowel(String^);
```


8.5 Пример выполнения лабораторного задания

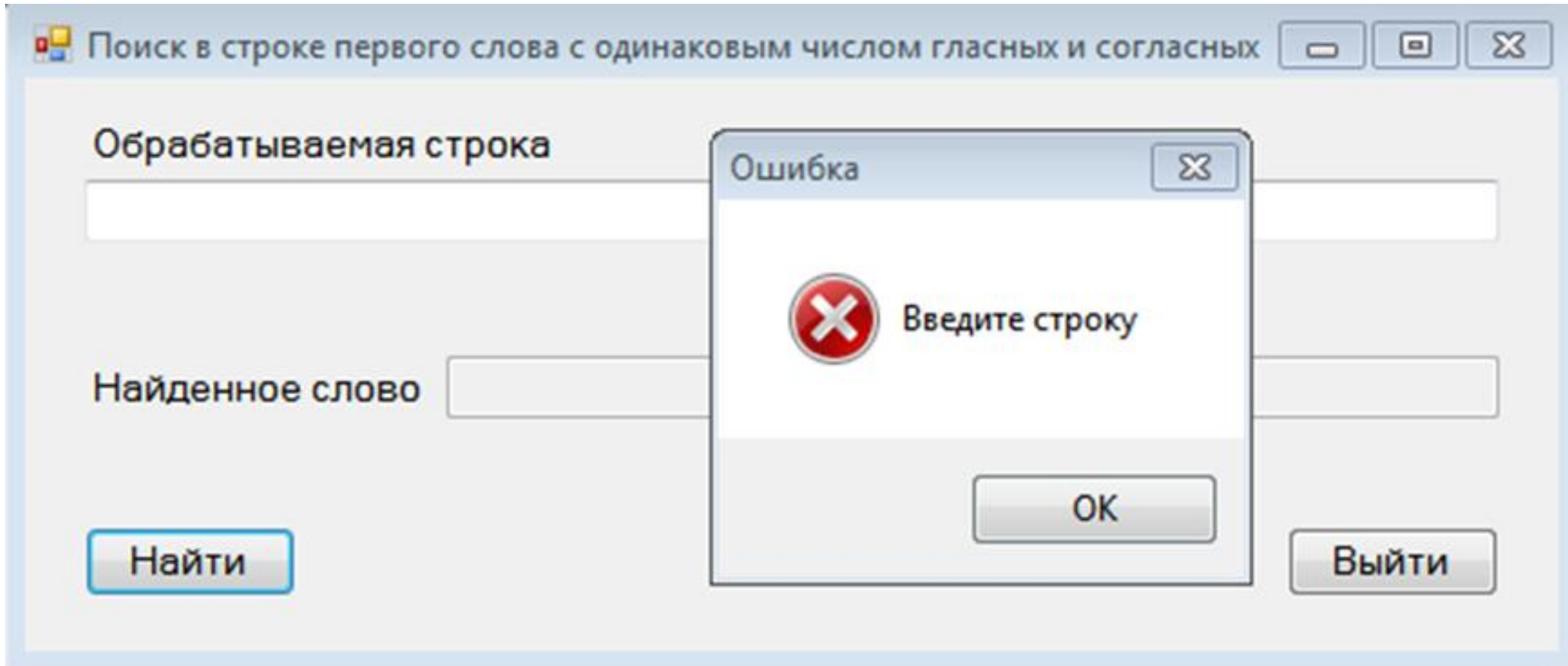
4. Программный код проекта. Тексты функций в файле Find.cpp

```
// Функция поиска в массиве первого слова с заданным условием
String^ GetWord(array<String^>^ arr)
{
    String^ Word = "";           // Объявление и инициализация строки с результатом поиска
    for each (String^ s in arr)   // Цикл перебора всех слов в массиве
    // Слово должно прежде всего иметь четную длину,
    // а количество гласных букв должно составлять половину длины слова
        if (s->Length % 2 == 0 && GetVowel(s) == s->Length/2)
        {
            Word = s;             // Найденное слово копируется в Word
            break;                // Досрочный выход из цикла
        }
    return Word;                  // Возврат найденного слова или пустой строки
}

// Функция подсчета количества гласных букв в слове
int GetVowel(String^ str)
{
    String^ Vowel = "aeёиоуыэюяАЕЁИОУЫЭЮЯ"; // Строка с гласными буквами
    int Qty = 0;                               // Счетчик гласных букв
    for (int i = 0; i < str->Length; i++)       // Цикл по буквам слова
    // Если очередная буква входит в строку гласных букв, увеличиваем счетчик на 1
    if (Vowel->IndexOf(str->Substring(i,1)) >= 0) Qty++;
    return Qty;                                // Возврат количества гласных букв в слове
}
```

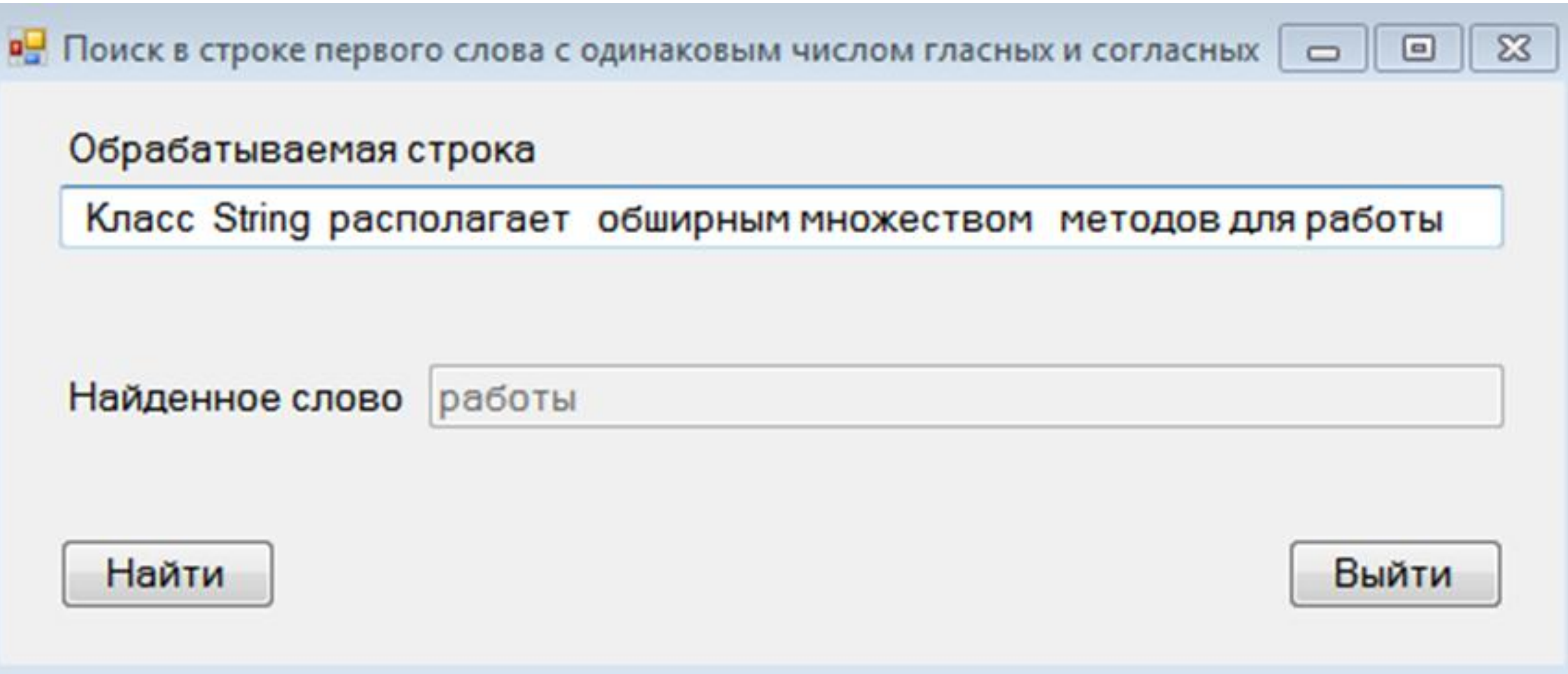
8.5 Пример выполнения лабораторного задания

5. Результаты работы приложения при пустой исходной строке



8.5 Пример выполнения лабораторного задания

5. Результаты работы приложения при найденном слове



Поиск в строке первого слова с одинаковым числом гласных и согласных

Обрабатываемая строка

Класс String располагает обширным множеством методов для работы

Найденное слово

работы

Найти

Выйти

8.5 Пример выполнения лабораторного задания

5. Результаты работы приложения при ненайденном слове

