1.8. Символьные строки

Wednesday, May 13, 2020 21:59

Язык C не имеет нативной поддержки строк по сравнению, например, с Delphi. Здесь работа со строками ведется через символьные массивы и функции для работы со строками из разных библиотек.

В этом уроке мы разберем:

- понятие символьных массивов, в т.ч. массивов переменной длины
- экранирование символов (escape characters)
- создание структур с символьными массивами
- различные операции со строками

Вспомним пройденное

Со строками мы встретились еще в первом уроке, когда писали программу Hello World:

```
printf ("Hello, World!\n");
```

Здесь функции **printf()** передается строка `"Hello, World!\n"`.

Двойные кавычки (**"**) нужны для определения границ строки, чтобы символы внутри этих границ воспринимались компилятором именно как символы, а не как ключевые слова. Кстати, можно использовать двойные кавычки и внутри строки, – компилятор поймет, что мы хотим. Но об этом позже, там есть свои особенности.

Когда мы знакомились с типом данных **char**, мы узнали, что этот тип может хранить только *один* символ. Чтобы присвоить значение переменной типа **char**, используются одинарные кавычки:

```
plusSign = '+';
```

То есть, между одинарными и двойными кавычками есть разница, о которой нужно помнить. Запись

```
plusSign = "+";
```

работать не будет.

Символьные массивы

Для работы с переменными, которые должны хранить больше одного символа обычно используются **массивы символов**.

В уроке 1.5 (*массивы*) мы задавали символьный массив **word**:

```
char  word [] = { 'H', 'e', 'l', 'l', 'o', '!' };
```

Так как мы не задали длину массива, компилятор автоматически подсчитает количество символов и создаст массив из 6 элементов.

Чтобы вывести на экран содержимое массива `word`, нужно по очереди вывести каждый символ, используя формат **%c**.

На основе вышесказанного можно построить различные функции для работы со строками. Наиболее популярные операции над строками включают объединение двух строк (*concatenate*), копирование содержимого строки в другую строку, извлечение части строки (*substring*) и проверку равенства двух строк (т.е. они должны содержать одинаковые символы в одинаковом порядке).

Напишем функцию **concat()**, которая будет объединять две строки:

```
concat (result, str1, n1, str2, n2);
```

где **str1** и **str2** – символьные массивы, которые будут объединены, а **n1** и **n2** соответственно –

количество символов в этих массивах. Так мы сможем объединять массивы любой длины.

```c
// Function to concatenate two character arrays

#include <stdio.h>

void  concat (char  result[], const char  str1[], int  n1,
                              const char  str2[], int  n2)
{
    int  i, j;

    // copy str1 to result

    for ( i = 0;  i < n1;  i++ )
            result[i] = str1[i];

    // copy str2 to result

    for ( j = 0;  j < n2;  j++ )
            result[n1 + j] = str2[j];
}

int main (void)
{
    void   concat (char  result[], const char  str1[], int  n1,
                                   const char  str2[], int  n2);
    const  char   s1[5] = { 'T', 'e', 's', 't', ' '};
    const  char   s2[6] = { 'w', 'o', 'r', 'k', 's', '.' };
    char   s3[11];
    int    i;

    concat (s3, s1, 5, s2, 6);

    for ( i = 0;  i < 11;  i++ )
        printf ("%c", s3[i]);

    printf ("\n");

    return 0;
}
```

Программа выдаст:

```
Test works.
```

Обратите внимание, что массив **s3**, который хранит результаты соединения двух массивов, должен быть достаточных размеров, иначе при попытке записи в несуществующий элемент массива программа будет работать непредсказуемо.

Строки переменной длины

В нашей функции **concat()** нам пришлось следить за количеством символов в массиве. А что если нам нужно провести много разных операций над строкой? Запутаешься следить за тем, где какой длины могут быть строки. Рассмотрим способ обработки строк, которые не будут зависеть от их длины.

Способ заключается в добавлении **спецзнака** в конец каждой строки. Таким образом функция-обработчик сможет понять, что она дошла до конца строки, когда она дойдет до этого символа.

В языке С, спецзнак, обозначающий конец строки, называется **нулевым символом** (*null character*) и

записывается как **'\0'**. Например, выражение

```
const char  word [] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
```

задает массив из семи символов, и последний из них – нулевой.
прим.: вспомните, что в языке C символ обратной косой черты [\] является специальным и не считается как отдельный символ; таким образом '\0' будет занимать один элемент массива, а не два, потому что компилятор преобразовывает '\0' в один нулевой символ.

Начнем с простого – напишем функцию **stringlength()**, которая будет определять длину строки с использованием нулевого символа.

```
// Function to count the number of characters in a string

#include <stdio.h>

int  stringLength (const char  string[])
{
    int  count = 0;

    while ( string[count] != '\0' )
        ++count;

    return count;
}

int main (void)
{
    int   stringLength (const char  string[]);
    const char  word1[] = { 'a', 's', 't', 'e', 'r', '\0' };
    const char  word2[] = { 'a', 't', '\0' };
    const char  word3[] = { 'a', 'w', 'e', '\0' };

    printf ("%i   %i   %i\n", stringLength (word1),
              stringLength (word2), stringLength (word3));

    return 0;
}
```

Программа выдаст:

```
5   2   3
```

При объявлении функции **stringlength()** мы в качестве аргумента указали ключевое слово **const**, потому что функция не изменяет массив, а просто считает количество элементов.

Инициализация и вывод строк на экран

Вернемся к нашей функции **concat()** и перепишем ее, чтоб она работала со строками произвольной длины. Функция теперь не должна принимать длины входных строк; она будет принимать только соединяемые массивы и массив, в который нужно записать результат.

Прежде чем приступить к написанию программы, рассмотрим несколько полезных фич, которые помогут упростить работу со строками.

Первое – можно инициализировать массив символов, не указывая каждый символ в кавычках отдельно, а просто указать все элементы сразу, строкой:

```
char  word[] = { "Hello!" };
```

Нулевой символ **'\0'** тоже автоматически добавится в конец строки, его указывать не нужно.

Можно и без фигурных скобок:

```
    char  word[] = "Hello!";
```

Две записи выше будут эквивалентны уже известной нам записи

```
    char  word[] = { 'H', 'e', 'l', 'l', 'o', '!', '\0' };
```

Если мы явно указываем размер массива, нужно убедиться, что в массиве будет достаточно места под нулевой символ. Так, в выражении

```
    char  word[7] = { "Hello!" };
```

программе хватит места, чтобы записать в массив нулевой символ.

А в выражении

```
    char  word[6] = { "Hello!" };
```

нулевой символ в массив не поместится и записан не будет. И компилятор на это никак не пожалуется. А попытка записи в несуществующий элемент массива может закончиться вылетом программы с ошибкой. Так что нужно избегать подобных случаев.

Кстати, функция **printf()** тоже использует нулевой символ, чтобы определить, что нужно закончить вывод символов. Мы ведь ей тоже на вход подаем строку и компилятор тоже в ее конец дописывает нулевой символ.

Вторая фича – удобный вывод строк с использованием формата **%s**. Этот формат позволяет вывести строку целиком, при условии что она заканчивается нулевым символом. Таким образом, выражение

```
    printf ("%s\n", word);
```

выведет на экран весь массив **word** целиком.

Наконец мы можем приступить к переписыванию функции **concat()**. Здесь нужно обратить внимание на несколько вещей:

- Так как мы больше не получаем на вход количество символов, функция должна будет самостоятельно определить, когда закончить обработку строки.
- Нулевой символ в конце массива **str1** не должен быть скопирован в выходной массив, потому что другие функции, которые будут с ним работать, посчитают, что на этом строка заканчивается и не будут читать строку дальше. Но в конце массива **str2** нулевой символ нужно сохранить, потому что на этом массиве итоговая строка завершается.

```
 1 #include <stdio.h>
 2
 3 int main (void)
 4 {
 5     void  concat (char  result[], const char  str1[], const char  str2[]);
 6     const char  s1[] = { "Test " };
 7     const char  s2[] = { "works." };
 8     char  s3[20];
 9
10     concat (s3, s1, s2);
11
12     printf ("%s\n", s3);
13
14     return 0;
15 }
16
17 // Function to concatenate two character strings
18
19 void concat (char  result[], const char  str1[], const char  str2[])
```

```
{
    int  i, j;

    // copy str1 to result

    for ( i = 0;  str1[i] != '\0';  i++ )
        result[i] = str1[i];

    // copy str2 to result

    for ( j = 0;  str2[j] != '\0';  j++ )
        result[i + j] = str2[j];

    // Terminate the concatenated string with a null character

    result [i + j] = '\0';
}
```

Помните, что индекс массива начинается с нуля, т.е. если есть массив **string** с **n** количеством символов, то чтобы прочитать последний символ строки, нужно написать **`string[n-1]`**. А нулевой символ будет храниться в **`string[n]`**. То есть, *при создании массива для строки нужно оставлять место под нулевой символ: создавать массив размером на 1 элемент больше, чем в нем будет символов*, т.е. размер массива должен быть **n + 1**, где **n** – количество символов в строке.

Возвращаясь к программе, мы выделили массиву **s3** 20 элементов. Это просто сделано для избыточности, чтобы соединенные между собой **str1** и **str2** гарантированно в него поместились. Благодаря нулевому символу, на экран выводятся только заполненные нами элементы массива.

Проверка двух строк на равенство

Для сравнения двух строк мы не можем использовать оператор равенства вроде

```
if ( string1 == string2 )
  ...
```

потому что этот оператор может быть применен только к простым типам переменных вроде **float**, **int** или **char**. К сложным типам данных вроде структур и массивов этот оператор неприменим.

Чтобы проверить равенство двух строк, нужно последовательно проверять символы в строке, один за другим. Если функция проверки наткнется на нулевой символ в обеих строках одновременно, и к этому моменту все предыдущие символы пройдут проверку на равенство, значит строки одинаковые.

Напишем функцию **`equalStrings()`**, которая будет на вход получать две строки и возвращать значение **true** или **false**.

```
// Function to determine if two strings are equal

#include <stdio.h>
#include <stdbool.h>

bool equalStrings (const char  s1[], const char  s2[])
{
    int  i = 0;
    bool areEqual;

    while ( s1[i] == s2 [i]  && s1[i] != '\0' &&  s2[i] != '\0' )
        i++;

    if ( s1[i] == '\0'  &&  s2[i] == '\0' )
        areEqual = true;
    else
```

```
17          areEqual = false;
18
19      return areEqual;
20  }
21
22
23  int main (void)
24  {
25        bool  equalStrings (const char  s1[], const char  s2[]);
26        const char  stra[] = "string compare test";
27        const char  strb[] = "string";
28
29        printf ("%i\n", equalStrings (stra, strb));
30        printf ("%i\n", equalStrings (stra, stra));
31        printf ("%i\n", equalStrings (strb, "string"));
32
33        return 0;
34  }
```

Программа выдаст:

```
0
1
1
```

Внимательно проследите за индексами массивов, с которыми работает программа. Цикл завершается как только он натыкается на нулевой символ в одной из строк или в обеих строках. Значение **i** будет равно индексу элемента массива, в котором был найден нулевой символ.
После обнаружения нулевого символа, программа проверяет, есть ли нулевой символ в обеих массивах. Если это так, значит массивы равны.

В этой программе показано еще одно интересное свойство, которое основано на особенностях работы указателей в языке С: мы можем в качестве аргумента передать не заранее заданный символьный массив, а просто написать строку в самом выражении:

```
printf ("%i\n", equalStrings (strb, "string"));
```

Мы разберем это подробнее в следующем уроке.

Кстати, функцию **equalStrings()** можно использовать прямо в выражениях:

```
if  ( equalStrings (string1, string2) )
  ...
```

Ввод строк с клавиатуры

Существуют разные функции для чтения строк, в разных библиотеках. Мы рассмотрим работу с функцией **scanf()**. В ней для чтения строки можно использовать формат данных **%s**, который будет читать вводимую строку, пока не будет достигнут символ **пробела**, **табуляции**(Tab) или символ **конца строки** (Enter), – чтение заканчивается на любом из них.

Таким образом, чтение строки в массив будет выглядеть вот так:

```
char  string[81];

scanf ("%s", string);
```

Обратите внимание, что в коде отсутствует знак **&** перед **string**. Это опять-таки особенности работы с указателями, которые мы разберем в следующем уроке. Пока что примите как данное, что при чтении строк знак **&** ставить не нужно.

Если мы вызовем функцию **scanf()** и напишем, например, **"Gravity"**, то все эти символы будут сохранены в массив. Но если мы напишем **"iTunes playlist"**, то будет записано только слово **"iTunes"**, потому что знак **пробела** прерывает дальнейшее чтение строки. Если мы снова вызовем функцию **scanf()**, она перезапишет массив словом **"playlist"**, потому что функция **scanf()** всегда продолжает чтение с того места, где она остановилась.
Я уже имел с этим проблемы в упражнениях к предыдущих уроках и мне приходилось очищать клавиатурный буфер, чтобы scanf() не читал введенные ранее символы.

Если мы будем читать несколько строк подряд:

```
    scanf ("%s%s%s", s1, s2, s3);
```

то, если мы напишем несколько слов через пробел, например **"mobile app development"**, то в каждый массив будет записано по слову: в **s1** будет **"mobile"**, в **s2** будет **"app"** и в **s3** будет **"development"**.

Также функция **scanf()** автоматически добавляет нулевой символ в конец каждого массива. Так, при вводе **"Gravity"** в массива будет занято не 7, а 8 элементов массива, – в **string[7]** будет записан **нулевой символ**.

Напишем программу, которая наглядно покажет нам работу функции **scanf()** со строками.

```
 1 //  Program to illustrate the %s scanf format characters
 2
 3 #include <stdio.h>
 4
 5 int main (void)
 6 {
 7         char  s1[81], s2[81], s3[81];
 8
 9         printf ("Enter text:\n");
10
11         scanf ("%s%s%s", s1, s2, s3);
12
13         printf ("\ns1 = %s\ns2 = %s\ns3 = %s\n", s1, s2, s3);
14         return 0;
15 }
```

В итоге получим:

```
Enter text:
smart phone
apps

s1 = smart
s2 = phone
s3 = apps
```

Как видим, функция **scanf()** разделила слова на три разных массива, – первый раз после **пробела** и второй раз после **символа переноса строки**.

Если мы напишем больше 80 символов, не вводя пробел, табуляцию или новую строку, то первый массив переполнится. Это приведет либо к вылету программы, либо ее непредсказуемой работе. К сожалению, функция **scanf()** никак не может определить, какой длины строку ей передают и за этим нужно следить самостоятельно. Но есть способ предотвратить переполнение массива при помощи отсечения лишних знаков. Это делается добавлением допустимого количества знаков между **%** и **s**:

```
    scanf ("%80s%80s%80s", s1, s2, s3);
```

Таким образом, если мы передадим функции больше 80 символов, будут записаны только первые 80. И не забываем оставлять место под нулевой символ, что мы тут и сделали (у нас массив из 81 элементов).

Ввод одного символа

В стандартной библиотеке С если несколько функций для чтения и вывода как единичных символов, так и строк. Для чтения одного символа можно использовать функцию **`getchar()`**. Если мы вызовем функцию несколько раз подряд, она последовательно будет читать знаки со входа. То есть, если мы напишем **"abc"** и нажмем Enter, то первый вызов функции вернет **`'a'`**, второй – **`'b'`**, третий – **`'c'`**. А четвертый вызов вернет **`'\n'`**.

Может возникнуть вопрос, зачем нужна функция ввода одного символа, когда можно использовать **`scanf()`** и формат **`%c`**? Да, ничто не мешает использовать **`scanf()`**, но **`getchar()`** немного уменьшает количество писанины и улучшает читабельность кода, – сразу видно, что мы принимаем только один символ. И еще этой функции не нужно передавать никакие аргументы.

В некоторых случаях работы с текстом нужно прочитать предложение или несколько предложений. Обычно предложение сначала помещают в буфер, а потом проводят над ним дальнейшие операции. В этом случае вызвать **`scanf()`** с форматом **`%c`** не получится, потому что он начнет пытаться распихивать отдельные слова по отдельным массивам, а нам нужно упаковать все предложение в один массив.

Для этого есть функция **`gets()`**, она предназначена как раз для захвата всей строки целиком и прекращает считывать строку только по достижению символа новой строки (т.е. по нажатию на **Enter**).

прим.: часто компиляторы ругаются на эту функцию, мол, она устаревшая и опасная. Некоторые компиляторы вообще отказываются компилировать код. Дело в том, что она создает возможность атаки с переполнением буфера, – то же самое, о чем мы говорили насчет переполнения массива при вводе слишком длинной строки в `scanf()`. Но там мы реализовали защиту от переполнения, а тут защиту никак не реализуешь. Так что все-таки стоит избегать. Даже гугл сразу в живом поиске выдает "why gets function is dangerous".

Попытаемся самостоятельно реализовать функцию с подобным функционалом, напишем функцию **`readLine()`**, которая в качестве аргумента будет принимать массив, в который должна быть записана считанная строка. Ну и не забываем, что не нужно записывать символ новой строки, по которому мы будем завершать функцию.

```c
#include <stdio.h>

int main (void)
{
    char  line[81];
    void  readLine (char  buffer[]);

    readLine (line);
    printf ("%s\n\n", line);

    return 0;
}

// Function to read a line of text from the terminal

void  readLine (char  buffer[])
{
    char  character;
    int   i = 0;

    do
    {
        character = getchar ();
        buffer[i] = character;
        i++;
    }
    while ( character != '\n' );
```

```
28
29      buffer[i - 1] = '\0';
30  }
```

Попробуем ввести строку с пробелами:

```
This is a sample line of text.
This is a sample line of text.
```

Защиту от переполнения массива здесь нетрудно реализовать. Достаточно добавить второй аргумент функции – число символов, по достижению которых цикл **do** закончится. Еще хорошо бы уведомить пользователя, какие данные от него ожидаются:

```
    printf("Enter a line of text, up to 80 characters. Hit enter when done:\n");
```

Мы так уже делали в упражнениях к предыдущему уроку; показывали пользователю, в каком формате вводить дату и время.

Для следующего примера представим, что мы пишем текстовый редактор и нужно подсчитывать количество слов в документе. Для этого нужно будет написать функцию **countWords()**, которая будет получать на вход строку и возвращать количество слов в ней. Для простоты, словами будем считать любую последовательность алфавитных букв. Разделять слова будем по всем символам, которые не являются алфавитными буквами. Может сначала показаться, что нужно искать пробелы и символы новой строки, но часто люди забывают ставить пробелы после знаков препинания, например. К тому же, такой подход более прост и при этом более гибкий.

```
1  #include <stdio.h>
2  #include <stdbool.h>
3
4  //  Function to determine if a character is alphabetic
5  bool isAlphabetic (const char  c)
6  {
7      if  ( (c >= 'a'  &&  c <= 'z') || (c >= 'A'  &&  c <= 'Z') )
8         return true;
9      else
10       return false;
11 }
12
13 // Function to count the number of words in a string
14 int  countWords (const char  string[])
15 {
16     int   i, wordCount = 0;
17     bool  lookingForWord = true, alphabetic (const char  c);
18
19     for ( i = 0;  string[i] != '\0';  i++ )
20         if ( isAlphabetic(string[i]) )
21         {
22             if ( lookingForWord )
23             {
24                 ++wordCount;
25                 lookingForWord = false;
26             }
27         }
28         else
29            lookingForWord = true;
30
31     return wordCount;
32 }
33
34 int main (void)
```

```
35 {
36     const char  text[] = "And here we go... again.";
37     int   countWords (const char  string[]);
38
39     printf ("%s - words = %i\n", text, countWords (text));
40
41     return 0;
42 }
```

Программа выдаст:

```
And here we go... again. - words = 5
```

Рассмотрим функцию **countWords()** поподробнее.

Функция начинает работу с поиска алфавитных букв и как только натыкается на первую букву, увеличивает на 1 счетчик слов **wordCount** и выставляет значение **False** флагу **lookingForWord** (т.е. функция нашла слово и в текущий момент его не ищет).
Это приведет к тому, что если следующий символ тоже будет буквой (то есть, мы все еще находимся *внутри* слова), то функция каких действий не выполнит и просто перейдет к следующему символу. И так пока не дойдет до символа, который не является буквой. В этом случае флагу **lookingForWord** будет выставлено значение **True** и программа дальше продолжит искать слова.

Для наглядности нарисуем таблицу, в которую запишем, как будут изменяться переменные функции по ходу чтения строки. Попытайтесь самостоятельно проработать ход выполнения программы.

i	string[i]	wordCount	lookingForWord
		0	true
0	'W'	1	false
1	'e'	1	false
2	'l'	1	false
3	'l'	1	false
4	','	1	true
5	' '	1	true
6	'h'	2	false
7	'e'	2	false
8	'r'	2	false
9	'e'	2	false
10	' '	2	true
11	'g'	3	false
12	'o'	3	false
13	'e'	3	false
14	's'	3	false
15	'.'	3	true
16	'\0'	3	true

Нулевая строка (Null string)

Рассмотрим более приближенный к практике вариант применения функции **`countWords()`**. В этот раз мы будем использовать функцию **`readLine()`**, чтобы создать пользователю возможность ввести несколько строк текста. После этого программа подсчитает количество слов и выведет результат.

Для гибкости мы не будем ограничивать количество вводимых строк, вместо этого мы создадим пользователю возможность "сказать" программе, что он закончил ввод текста. Один из вариантов – заканчивать ввод по двойному нажатию на **Enter**. При втором нажатии на **Enter** кроме символа новой строки функция **`readLine()`** ничего не получит и поэтому **вернет только нулевой символ**.

Строка, которая не содержит никаких символов кроме одного нулевого, называется **нулевой строкой**. С ней будут корректно работать все функции для работы со строками: **`stringLength()`** будет возвращать нулевую длину, **`concat()`** не будет присоединять никакие символы, даже **`equalStrings()`** правильно покажет равенство двух нулевых строк.

Если нужно задать нулевую строку, это делается парой двойных кавычек:

```
char  buffer[100] = "";
```

Обратите внимание, что внутри кавычек ничего быть не должно, даже пробела.

Для закрепления пройденного материала напишем программу, которая будет считывать строку с клавиатуры и подсчитывать количество символов в ней.

```
#include <stdio.h>
#include <stdbool.h>

bool alphabetic (const char  c)
{
    if  ( (c >= 'a'  &&  c <= 'z') || (c >= 'A'  &&  c <= 'Z') )
      return true;
    else
      return false;
}

void  readLine (char  buffer[])
{
    char  character;
    int   i = 0;

    do
    {
        character = getchar ();
        buffer[i] = character;
        i++;
    }
    while ( character != '\n' );

    buffer[i - 1] = '\0';
}

int  countWords (const char  string[])
{
    int   i, wordCount = 0;
    bool  lookingForWord = true, alphabetic (const char  c);

    for ( i = 0;  string[i] != '\0';  i++ )
        if ( alphabetic(string[i]) )
        {
            if ( lookingForWord )
            {
```

```
38                  ++wordCount;
39                  lookingForWord = false;
40              }
41          }
42          else
43              lookingForWord = true;
44
45      return wordCount;
46  }
47
48  int main (void)
49  {
50      char  text[81];
51      int   totalWords = 0;
52      int   countWords (const char  string[]);
53      void  readLine (char  buffer[]);
54      bool  endOfText = false;
55
56      printf ("Type in your text.\n");
57      printf ("When you are done, double press 'ENTER'.\n\n");
58
59      while ( ! endOfText )
60      {
61          readLine (text);
62
63          if ( text[0] == '\0' )
64              endOfText = true;
65          else
66              totalWords += countWords (text);
67      }
68
69      printf ("\nThere are %i words in the above text.\n",  totalWords);
70
71      return 0;
72  }
```

Попробуем ввести текст и посмотрим, что получится:

```
Type in your text.
When you are done, press 'ENTER'.

Wendy glanced up at the ceiling where the mound of lasagna loomed
like a mottled mountain range. Within seconds, she was crowned with
ricotta ringlets and a tomato sauce tiara. Bits of beef formed meaty
moles on her forehead. After the second thud, her culinary coronation
was complete.

There are 48 words in the above text.
```

Здесь мы ввели 5 строк, каждая из них была обработана функцией **countWords()**, после чего подсчитанное ею количество слов в строке прибавлялось к счетчику общего количества слов. Когда программа встречает нулевую строку(то есть, первый ее символ – нулевой), она ничего не передает функции **countWords()**, а выставляет флагу **endOfText** значение **True**, из-за чего условие цикла перестает выполняться и цикл завершается.

Экранирование символов (Escape characters)

Я уже вскользь упоминал, что список специальных символов с обратным слэшем (****) не заканчивается на символе новой строки и нулевом символе. Таких символов достаточно много и так как для их вызова мы используем спецсимвол, они называются **экранируемыми символами** (*escape characters*).

Приведем их список.

Escape Character	Name
\a	Audible alert
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\"	Double quotation mark
\'	Single quotation mark
\?	Question mark
\nnn	Octal character value *nnn*
\unnnn	Universal character name
\Unnnnnnnn	Universal character name
\xnn	Hexadecimal character value *nn*

Первые 7 символов списка выполняют связанное с ними действие. Например, **'\a'** заставляет компьютер проиграть звук ошибки (*Windows Foreground.wav*). Символ удаления **'\b'** удалит одну букву в терминале (по аналогии с кнопкой **Backspace**). Символ горизонтальной табуляции **'\t'** помогает легко формировать данные в колонки, распределяя их на расстояние в 8 символов между первыми знаками соседних значений. То есть, выражение

```
    printf ("%i\t%i\t%i\n", 22, 333, 4444);
```

выдаст

```
22      333     4444

```

Чтобы напечатать символ обратного слэша, нужно написать два слэша подряд:

```
    printf ("\\t is the horizontal tab character.\n");
```

Чтобы вставить символ двойных кавычек, перед кавычками нужно написать обратный слэш:

```
    printf ("\"Hello,\" he said.\n");
```

```
"Hello," he said.
```

Чтобы присвоить одинарную кавычку переменной типа **char**, нужно перед ней написать обратный слэш:

```
    c = '\'';
```

Когда нужно, чтоб компилятор не спутал строку с **триграфом** (поговорим о них в будущих уроках), нужно вызвать экранированный символ знака вопроса: **'\?'**.

Последние четыре записи в таблице показывают, как можно вывести **любой** символ, использовав его код в кодировке **ASCII**.
Например в **'\nnn', nnn** – код символа из трех цифр в восьмеричном формате. А в **'\xnn', nn** – код символа из двух цифр в шестнадцатиричном формате. То есть, знак вопроса (**?**) можно вывести, набрав **\077** или **\x3f**. *Плохой пример, но лучше не придумал, надо исправить.*

Отсюда же, кстати, растут ноги и у нулевого символа **'\0'**. Он потому и называется нулевым, потому что его значение 0. Это свойство даже можно использовать в логических выражениях:

```
    while ( string[count] )
        ++count;
```

Такой цикл будет продолжаться до тех пор, пока не будет достигнут нулевой символ.

Всего в кодировке **ASCII** 256 символов. Первые 32 символа являются служебными, первые 128 одинаковые для всех систем. Вторая половина символов зависит от языка системы, например, у меня появляются русские символы.

Если нужно вывести символ из кодировки, где символы представлены не 8-битным, а 16- или 32-битным значением, есть операторы **\u**, который требует число и 4-х шестнадцатиричных цифр и **\U**, который требует число из 8-и шестнадцатиричных цифр. Цифры берутся из таблицы **универсального набора символов** (*universal character name*), т.е. это символы **Unicode**. Например, **'\u03A8'** задаст символ Ψ.

C99 and C++ disallow the hexadecimal values representing characters in the basic character set (base source code set) and the code points reserved by ISO/IEC 10646 for control characters.

The following characters are also disallowed:

- *Any character whose short identifier is less than 00A0. The exceptions are 0024 ($), 0040 (@), or 0060 (').*
- *Any character whose short identifier is in the code point range D800 through DFFF inclusive.*

Больше о строках-константах

Если поместить обратный слэш в конец строки и перейти на новую строку, то компилятор С проигнорирует все, что будет написано в первой строке после слэша. Это часто помогает, если нужно задать очень длинную строку или нужно объявить макрос (о макросах поговорим в следующих уроках).

```
    char  letters[] =
          { "abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

Если мы напишем как показано выше, компилятор выдаст ошибку **missing terminating " character**. Для продолжения строки нужно поставить слэш:

```
    char  letters[] =
          { "abcdefghijklmnopqrstuvwxyz\
ABCDEFGHIJKLMNOPQRSTUVWXYZ" };
```

Также важно понимать, что отступы в редакторе кода – тоже символы и нужно продолжать текст *с самого начала строки*, без отступов, если вы не хотите, чтобы они попали в строку.

Еще один способ переноса длинной строки – использование нескольких пар двойных кавычек. То есть, если мы зададим строки **"one" "two" "three"**, они будут записаны как **"onetwothree"**.

Код будет выглядеть так:

```
    char  numbers[] =
         { "one"
           "two"
           "three"};
```

Таким же образом, при попытке вызвать три функций **printf()** подряд,

```
    printf ("Programming in C is fun\n");
    printf ("Programming"  " in C is fun\n");
    printf ("Programming"  " in C"  " is fun\n");
```

компилятор объединит их в одну строку и вызовет **printf()** один раз. Кстати, в данном случае он еще и одинаковые фразы выведет, потому что пробелы между парами кавычек игнорируются.

Строки, структуры и массивы

Допустим, нам нужно написать программу-словарь. Пользователь набирает слово и программа ищет перевод и/или значение слова в базе данных. Сначала надо разобраться, в каком формате лучше хранить данные словаря. Первое что может прийти в голову (с текущим уровнем знаний) – структура из слова и его определения:

```
struct  entry
{
    char  word[15];
    char  definition[50];
};
```

Эта структура состоит из 14-буквенного слова и 49-буквенного определения этого слова (не забываем, что нулевой символ занимает один элемент массива).

Ну и нужно создать массив таких структур, потому что в здравом уме никто не будет выделять по переменной на каждое слово:

```
struct entry  dictionary[100];
```

Да, 100 слов – не так много и если мы делаем нормальный словарь, в нем скорее всего будет 100 000+ слов и для работы с ними будет нужен более сложный подход, чем мы здесь рассматриваем, как минимум нужно будет хранить словарь отдельным файлом, а не помещать в память программы. Но пока остановимся на массиве структур.

Со структурой данных словаря определились, теперь надо определиться с его организацией. Обычно слова идут в алфавитном порядке, мы тоже так сделаем.

Теперь можно перейти к написанию программы. Зададим функцию, **`lookup()`** которая будет искать введенное слово в базе данных словаря. Если слово найдено, функция вернет индекс элемента массива, в котором оно записано, в противном случае она вернет **`-1`**, тем самым сообщая о неудачном поиске. В итоге вызов функции будет выглядеть вот так:

```
entry = lookup (dictionary, word, entries);
```

где **`dictionary`** – массив с базой данных, **`word`** – искомое слово, **`entries`** – количество элементов в базе данных.

Также мы можем воспользоваться написанной нами ранее функцией **`equalStrings()`**, которая будет проверять совпадение введенного слова со словами в базе данных.

```
 1 // Program to use the dictionary lookup program
 2
 3 #include <stdio.h>
 4 #include <stdbool.h>
 5
 6 struct  entry
 7 {
 8     char   word[15];
 9     char   definition[50];
10 };
11
12 bool equalStrings (const char  s1[], const char  s2[])
13 {
14     int  i = 0;
15     bool areEqual;
16
17     while ( s1[i] == s2 [i]  &&
18                 s1[i] != '\0' &&  s2[i] != '\0' )
19         i++;
20
21     if ( s1[i] == '\0'  &&  s2[i] == '\0' )
22         areEqual = true;
23     else
24         areEqual = false;
25
```

```
26      return areEqual;
27  }
28
29  // function to look up a word inside a dictionary
30
31  int  lookup (const struct entry  dictionary[], const char  search[],
32                const int  entries)
33  {
34      int  i;
35      bool equalStrings (const char s1[], const char s2[]);
36
37      for ( i = 0;  i < entries;  i++ )
38          if ( equalStrings (search, dictionary[i].word) )
39              return i;
40
41      return -1;
42  }
43
44  int main (void)
45  {
46      const struct entry  dictionary[100] =
47        { { "aardvark", "A burrowing African mammal"        },
48          { "abyss",    "A bottomless pit"                  },
49          { "acumen",   "Mentally sharp; keen"              },
50          { "addle",    "To become confused"                },
51          { "aerie",    "A high nest"                       },
52          { "affix",    "To append; attach"                 },
53          { "agar",     "A jelly made from seaweed"         },
54          { "ahoy",     "A nautical call of greeting"       },
55          { "aigrette", "An ornamental cluster of feathers" },
56          { "ajar",     "Partially opened"                  } };
57
58      char  word[10];
59      int   entries = 10;
60      int   entry;
61      int   lookup (const struct entry  dictionary[], const char  search[],
62                    const int  entries);
63
64      printf ("Enter word: ");
65      scanf ("%14s", word);
66      entry = lookup (dictionary, word, entries);
67
68      if ( entry != -1 )
69          printf ("%s\n", dictionary[entry].definition);
70      else
71          printf ("Sorry, the word %s is not in my dictionary.\n", word);
72
73      return 0;
74  }
```

Функция **lookup()** последовательно перебирает элементы массива, передавая слова на сравнение функции **equalStrings()**, пока та не найдет совпадение. Если все слова будут перебраны и совпадение не будет найдено, то **lookup()** вернет **-1** и программа выдаст сообщение, что такого слова нет в словаре.

Более оптимальный метод поиска

Мы решили задачу в лоб: последовательно перебрали все слова в массиве. С маленькими массивами так можно делать, но если нам нужно будет искать среди сотен тысяч слов, тормоза станут заметной проблемой. Решить эту проблему можно, воспользовавшись тем, что у нас слова в базе данных отсортированы в алфавитном порядке.

Это дает нам возможность использовать алгоритм **бинарного поиска**.

Представим себе, что мы играем в угадайку. Ведущий придумывает число от 1 до 99 и нам нужно отгадать число за наименьшее количество попыток. При каждой попытке ведущий говорит нам, угаданное нами число меньше, больше или совпадает с загаданным. Допустим, если мы начнем угадывать с 50, ответ "больше" или "меньше" сузит круг поиска со 100 позиций до 49 позиций. Дальше процесс повторяется: допустим, если на ответ "50" нам сказали "больше", то нам нужно искать от 51 до 100 включительно. Срединой будет 75, проверяем это число. И так пока не дойдем до загаданного числа. Такой алгоритм в среднем работает быстрее других алгоритмов поиска.

Перепишем функцию **lookup()** под новый алгоритм. Также вместо функции **equalStrings()** нам нужно будет написать функцию **compareStrings()**, которая будет возвращать **-1**, если сравниваемое слово выше по алфавиту, **0**, если слова равны и **1**, если сравниваемое слово ниже по алфавиту. То есть,

```
compareStrings ("alpha", "beta");
```

вернет значение **-1**, а выражение

```
compareStrings ("zioty", "yucca");
```

вернет **1**.

Основная функция **main()** из предыдущей программы никак не изменяется.

```
// Dictionary lookup program

#include <stdio.h>

struct  entry
{
char  word[15];
char  definition[50];
};

// Function to compare two character strings

int  compareStrings (const char  s1[], const char  s2[])
{
int  i = 0, answer;

while ( s1[i] == s2[i] && s1[i] != '\0'&& s2[i] != '\0' )
    i++;

if ( s1[i] < s2[i] )
    answer = -1;               /* s1 < s2  */
else if ( s1[i] == s2[i] )
    answer = 0;                /* s1 == s2 */
else
    answer = 1;                /* s1 > s2  */

return answer;
}

// Function to look up a word inside a dictionary

int  lookup (const struct entry  dictionary[], const char  search[],
         const int  entries)
{
int  low = 0;
int  high = entries - 1;
int  mid, result;
int  compareStrings (const char  s1[], const char  s2[]);

while  ( low <= high )
{
    mid = (low + high) / 2;
```

```
43          result = compareStrings (dictionary[mid].word, search);
44
45          if ( result == -1 )
46              low = mid + 1;
47          else if ( result == 1 )
48              high = mid - 1;
49          else
50              return mid;   /* found it */
51      }
52
53      return -1;            /* not found */
54      }
55
56      int main (void)
57      {
58      const struct entry  dictionary[100] =
59        { { "aardvark", "A burrowing African mammal"        },
60          { "abyss",    "A bottomless pit"                  },
61          { "acumen",   "Mentally sharp; keen"              },
62          { "addle",    "To become confused"                },
63          { "aerie",    "A high nest"                       },
64          { "affix",    "To append; attach"                 },
65          { "agar",     "A jelly made from seaweed"         },
66          { "ahoy",     "A nautical call of greeting"       },
67          { "aigrette", "An ornamental cluster of feathers" },
68          { "ajar",     "Partially opened"                  } };
69
70      char  word[10];
71      int   entries = 10;
72      int   entry;
73      int   lookup (const struct entry  dictionary[], const char  search[],
74                    const int  entries);
75
76      printf ("Enter word: ");
77      scanf ("%14s", word);
78      entry = lookup (dictionary, word, entries);
79
80      if ( entry != -1 )
81          printf ("%s\n", dictionary[entry].definition);
82      else
83          printf ("Sorry, the word %s is not in my dictionary.\n", word);
84
85      return 0;
86      }
```

Разберем программу. Цикл в функции **compareStrings()** такой же как и у **equalStrings()**, то есть цикл останавливается либо когда доходит до **нулевого символа**, либо если замечает **разницу между словами**. Таким образом мы пропускаем одинаковые начала слов и проверяем первые различающиеся символы в словах. Так как в кодировке **ASCII** (да и во всех других популярных кодировках) коды букв **возрастают** с алфавитным порядком, мы можем использовать обычный оператор сравнения, т.е. если одна буква **ниже** второй по алфавиту, то сравнение покажет "**меньше**". И наоборот. Если же все символы слов прошли проверку на равенство и одновременно закончились на нулевой символ, значит они одинаковые.

Функция **lookup()** имеет переменные **low** и **high** – текущие границы поиска.
Сначала границы расположены от первого до последнего элемента базы данных словаря (**high = entries - 1**, потому что нумерация элементов массива начинается с нуля).
Далее вычисляется середина и определяется направление изменения границ угадывания (в меньшую или большую сторону, в зависимости от того, что выдаст **compareStrings()**). Затем снова вычисляется середина. И так пока не будет найдено совпадение или пока границы не пересекутся (что означает, что совпадений не найдено). Обратите внимание, что деление здесь целочисленное, т.е. дробный остаток будет отсечен.

Операции над символами

Символы нередко используются в арифметических операциях и операциях сравнения. Поэтому важно понимать, как язык C работает с символами.

Когда символ обрабатывается в каком-либо выражении, язык C его конвертирует в целочисленное значение **int** и продолжает в таком виде с ним работать. В уроке 1.4 (*принятие решений*) мы использовали выражение

```
c >= 'a'  &&  c <= 'z'
```

чтобы определить, относится ли символ **c** к строчным буквам.

Как я уже говорил, строчные и прописные буквы в кодировке **ASCII** расположены последовательно, без лишних символов между ними и их коды также последовательно возрастают (см. таблицу ниже).

Symbol	Decimal	Binary
A	65	01000001
B	66	01000010
C	67	01000011
D	68	01000100
E	69	01000101
F	70	01000110
G	71	01000111
H	72	01001000
I	73	01001001
J	74	01001010
K	75	01001011
L	76	01001100
M	77	01001101
N	78	01001110
O	79	01001111
P	80	01010000
Q	81	01010001
R	82	01010010
S	83	01010011
T	84	01010100
U	85	01010101
V	86	01010110
W	87	01010111
X	88	01011000
Y	89	01011001
Z	90	01011010

Symbol	Decimal	Binary
a	97	01100001
b	98	01100010
c	99	01100011
d	100	01100100
e	101	01100101
f	102	01100110
g	103	01100111
h	104	01101000
i	105	01101001
j	106	01101010
k	107	01101011
l	108	01101100
m	109	01101101
n	110	01101110
o	111	01101111
p	112	01110000
q	113	01110001
r	114	01110010
s	115	01110011
t	116	01110100
u	117	01110101
v	118	01110110
w	119	01110111
x	120	01111000
y	121	01111001
z	122	01111010

Таблица алфавитных букв и соответствующих им десятичных и двоичных кодов ASCII.

Переоформить и добавить цифры.

То есть, выражение выше сравнивает значение **c** со значением **'a'**, которое в **ASCII** имеет код **97**, затем с **'b'**, у которого код **98** и так далее до конца.

Таким образом, эквивалентным выражением будет

```
c >= 97  &&  c <= 122
```

так как код символа **'z'** – **122**. Но первый вариант все же предпочтительнее, так как его легче понять.

Кстати, код символа можно вывести, если представить его как тип **int**:

```
printf ("%i\n", 'a');
```

Программа выдаст **97**.

А еще можно проводить над символами математические операции:

```
c = 'a' + 1;
printf ("%c\n", c);
```

Так как мы увеличиваем код буквы на единицу, то получим следующую по списку букву, то есть **'b'**.

На практике не так часто приходится заниматься такими странными вещами как прибавление цифр к буквам, но пример выше подводит нас к удобному способу переводить цифры от **'0'** до **'9'** в соответствующие им цифры в целочисленном виде, т.е. от **0** до **9**.

Напомню, что в языке С, например, **'0'** – это не то же самое, что **0**. В кодировке **ASCII** символ **'0'** имеет код **48**, его также можно узнать, воспользовавшись строкой выше. Таким образом, если мы вычтем из конвертируемой цифры символ **'0'**, то в результате получим ее целочисленное значение. То есть,

`'0' - '0' = 48 - 48 = 0`

или

`'5' - '0' = 53 - 48 = 5,`

потому что код **ASCII** символа **'5'** равен **53**.

В других кодировках результат скорее всего будет такой же, даже несмотря на то, что коды символов могут отличаться от кодов **ASCII** (потому что они скорее всего будут просто смещены в бóльшую или меньшую сторону).

Напишем функцию **strToInt()**, которая будет переводить число в формате строки в число типа **int**. Предполагая, что вводимое число не будет выходить за допустимые пределы этого типа, естественно.

```
#include <stdio.h>

// Function to convert a string to an integer
int  strToInt (const char  string[])
{
    int  i, intValue, result = 0;

    for  ( i = 0; string[i] >= '0' && string[i] <= '9'; i++ )
    {
        intValue = string[i] - '0';
        result = result * 10 + intValue;
    }

    return result;
}

int main (void)
{
    int  strToInt (const char  string[]);

    printf ("%i\n", strToInt("245"));
    printf ("%i\n", strToInt("100") + 25);
    printf ("%i\n", strToInt("13x5"));

    return 0;
}
```

Получим:

```
245
125
13
```

Программа последовательно перебирает символы в строке, отнимая от них **'0'**. Для присоединения каждой последующей цифры, нужно домножать текущее число на 10, чтобы сместить цифры влево. Также обратите внимание на последний результат: цикл остановился как только наткнулся на символ, который не является цифрой.
Программу желательно доработать: она не умеет обрабатывать отрицательные числа и у нее нет нормальной защиты от некорректных данных, например если подать в функцию **"xxx"**, она вернет **0**.

На этом мы заканчиваем урок. В стандартных библиотеках C существует много функций, которые мы здесь не рассмотрели, но написали свои реализации этих функций для лучшего понимания принципов работы со строками и символами. Например:

- **strlen()**, которая подсчитывает длину строки
- **strcmp()**, которая сравнивает две строки
- **strcat()**, которая соединяет две строки
- **atoi()**, которая преобразовывает строку в число типа **int**
- **isupper()**, **islower()**, **isalpha()** и **isdigit()**, которые проверяют, относится ли символ к прописным, строчным буквам, алфавиту и цифрам соответственно.

На практике программы желательно писать с использованием именно этих функций, а не изобретать велосипед и писать свои функции, как мы делали в этом уроке (исключительно в учебных целях!).

Источники:

1. Stephen Kochan – Programming in C (4th Edition); chapter 9
2. IBM Knowledge Center – The Unicode standard (C++ only)
   https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_73/rzarg/unicode_standard.htm

Упражнения

2. *Why could you have replaced the while statement of the equalStrings() function of Program 9.4:*

```
    while ( s1[i] == s2[i]  &&  s1[i] != '\0' &&  s2[i] != '\0' )
```

with the statement

```
    while ( s1[i] == s2[i]  &&  s1[i] != '\0' )
```

to achieve the same results?

Потому что первая строка рано или поздно дойдет до нулевого символа, а тест на равенство будет так или иначе провален, если найдется хоть одна пара несоответствующих символов. Но мне такой подход меньше нравится, что если в первой строке по ошибке не окажется нулевого символа? В первом случае цикл смогла бы завершить вторая строка, а так цикл может зависнуть.

3. *The countWords() function from Programs 9.7 and 9.8 incorrectly counts a word that contains an apostrophe as two separate words. Modify this function to correctly handle this situation. Also, extend the function to count a sequence of positive or negative numbers, including any embedded commas and periods, as a single word.*

Не вижу смысл менять функцию `countWords()`, когда можно поменять функцию `alphabetic()`.

```
1    bool alphabetic (const char  c)
2    {
3        if  ( (c >= 'a'  &&  c <= 'z') || (c >= 'A'  &&  c <= 'Z') ||
4              (c >= '\"' &&  c <= '.') || (c >= '0'  &&  c <= '9') )
5          return true;
```

```
        else
            return false;
    }
```

Просто расширил диапазон символов, которые считаются частью слова.

4. *Write a function called `substring()` to extract a portion of a character string. The function should be called as follows:*

```
substring (source, start, count, result);
```

where `source` is the character string from which you are extracting the substring, `start` is an index number into `source` indicating the first character of the substring, `count` is the number of characters to be extracted from the source string, and `result` is an array of characters that is to contain the extracted substring.

```
// Function to extract a portion of a character string
void subString (const char  source[], int start, int count, char  result[])
{
    int i;

    if (count <= 0)
        printf("Error: Less than 1 characters to copy.\n");
    else if (count > 80)
        printf("Error: Maximum copy length exceeded.\n");
    else {
        for (i = 0; i <= count - 1; i++) { // because array indices start from 0
            if (source[start + i] == '\0') {
                printf("Warning: End of string reached, %i of %i characters \
were copied.\n", i + 1, count);
                break;
            }
            else
                result[i] = source[start + i];
        }

        result[i] = '\0'; // terminate the string with null character
    }

    return;
}
```

Просто посимвольное копирование участка одной строки в другую строку, не забывая о том, что индексы массива начинаются с нуля и не забыв прилепить нулевой символ в конец второй строки.

5. *Write a function called findString() to determine if one character string exists inside another string. The first argument to the function should be the character string that is to be searched and the second argument is the string you are interested in finding. If the function finds the specified string, have it return the location in the source string where the string was found. If the function does not find the string, have it return –1.*

```
// Function to determine if one string exists in another
int findString (const char  s1[], const char  s2[])
{
    int  i, j;
    bool areEqual = false;

    for (i = 0; s1[i] != '\0'; i++) {
        if (s1[i] == s2[0]) {
            areEqual = true;

            for (j = 1; s2[j] != '\0'; j++)
                if (s1[i + j] != s2[j]){
                    areEqual = false;
                    break;
```

```
                    }
                }

                if (areEqual)
                    return i;
            }

            return -1;
        }
```

Функция последовательно сравнивает символы проверяемой строки с первым символом искомой строки. Если функция находит совпадение, то запускается цикл, который проверяет все последующие символы до конца искомой строки. Если обнаруживается несовпадение, цикл прерывается и поиск продолжается. Если совпадение оказалось полным, функция возвращает стартовую позицию поиска.

6. *Write a function called `removeString()` to remove a specified number of characters from a character string. The function should take three arguments: the source string, the starting index number in the source string, and the number of characters to remove.*

   Пришлось попотеть, но все ошибки были по невнимательности, нужно было просто все подробно расписать на бумаге, что куда должно двигаться и все сразу стало ясно.

```
// Function to remove a portion of a character string
void removeString (char source[], int start, int count)
{
    int i;

    if (count <= 0)
        printf("Error: Less than 1 characters to delete.\n");
    else {
        for (i = start; source[i] != '\0'; i++)
            source[i] = source[i + count];

        if (i < (count + start)) {
            printf("Warning: End of string reached, %i of %i \
characters were deleted.\n", (i - start), count);
            source[start] = '\0';
        }
    }

    return;
}
```

Функция смещает все элементы, которые должны остаться после удаления так, чтобы они начинались с того места, где началось "удаление" заданной части текста. Если же было задано удалить больше символов, чем есть в строке, нас защитит проверка каждого следующего элемента строки на нулевой символ: цикл прервется по его достижению. И количество удаленных символов можно посчитать, исходя из места, где программа остановилась.

7. *Write a function called `insertString()` to insert one character string into another string. The arguments to the function should consist of the source string, the string to be inserted, and the position in the source string where the string is to be inserted.*

   Снова пришлось расписывать, потому что в уме тяжело держать что куда должно двигаться.

```
// Function to insert a string into another string
void insertString (char s1[], const char s2[], int start)
{
    int i;
    int length1, length2;
    const int maxLength = 23; // max string length
    int end;

```

```
    // Determine the length of both strings
    for (i = 0; s1[i] != '\0'; i++) {}
    length1 = i;
    for (i = 0; s2[i] != '\0'; i++) {}
    length2 = i;

    // Error handling routines
    end = length1 + length2;

    if (start > length1) {
        printf("Error: Starting position is further than initial \
string end.\n");
        return;
    }

    if ( (end - 1) > maxLength ) {
        printf("Warning: End of string was reached. Some text \
might be lost.\n");
        s1[maxLength + 1] = '\0';
    }

    // Copying routines
    for (i = end; i >= start; i--){
        if (i > maxLength) // If out of string bounds...
            continue; // ...do nothing
        else
            s1[i] = s1[i - length2];
    }

    for (i = 0; i < length2; i++) { // Don't copy the null character
        if ((i + start) > maxLength) {
            return;                 // Exit on string overflow
        }
        else
            s1[i + start] = s2[i];
    }

    return;
}
```

Функция начинает с конца строки и перетаскивает символы вправо, чтобы освободить место под вставляемую строку, параллельно проверяя, не работает ли она с несуществующими элементами массива. Потом вставляет строку, снова проверяя, что она работает в разрешенном диапазоне.

8. *Using the findString(), removeString(), and insertString() functions from preceding exercises, write a function called replaceString() that that replaces `s1` inside `source` with the character string `s2`.*

Здесь я познал важность инициализации переменных. Программа работала неправильно, потому что в функции `findString()` не была инициализирована переменная `areEqual`.

```
// Function to find and replace the string
void replaceString (char source[], const char s1[], const char s2[])
{
    void insertString (char s1[], const char s2[], int start);
    int findString (const char  s1[], const char  s2[]);
    void removeString (char source[], int start, int count);

    int i;
    int position, length;

    for (i = 0; s1[i] != '\0'; i++) {}
    length = i;

    position = findString (source, s1);

```

```
16          if (position < 0) {
17              printf("Error: Searched string not found.\n");
18              return;
19          }
20
21          removeString(source, position, length);
22          insertString(source, s2, position);
23
24          return;
25      }
```

В самой функции нет ничего сложного, не думаю, что нужно давать какие-то пояснения.

9. *You can extend even further the usefulness of the replaceString() function from the preceding exercise if you have it return a value that indicates whether the replacement succeeded, which means that the string to be replaced was found inside the source string.*

```
1       // Function to find and replace the string
2       bool replaceString (char source[], const char s1[], const char s2[])
3       {
4           void insertString (char s1[], const char s2[], int start);
5           int findString (const char  s1[], const char  s2[]);
6           void removeString (char source[], int start, int count);
7
8           int i;
9           int position, length;
10          bool result = true;
11
12          for (i = 0; s1[i] != '\0'; i++) {}
13          length = i;
14
15          position = findString (source, s1);
16
17          if (position < 0) {
18              result = false;
19              return result;
20          }
21          else
22              result = true;
23
24          removeString(source, position, length);
25          insertString(source, s2, position);
26
27          return result;
28      }
```

У меня уже был реализован похожий функционал, так что достаточно было заменить вывод предупреждения на возврат переменной типа bool.

10. *Write a function called dictionarySort() that sorts a dictionary into alphabetical order.*

Функция основана на функции сортировки численных массивов, которую я писал в предыдущем уроке (урок 1.6 (пользовательские функции), упражнение 13). Так что бонусом у нас есть возможность выбирать направление сортировки (a – z или z – a).

```
1       // Function to sort the dictionary
2       void  sort (struct entry  a[], int  entries, int direction)
3       {
4           int  i, j, k;
5           struct entry temp;
6           int val1, val2;
7
8           for ( i = 0;  i < entries - 1;  i++ ) {
9               val1 = 0;
```

```
            for (k = 0; a[i].word[k] != '\0'; k++)
                val1 += a[i].word[k];

            for ( j = i + 1;  j < entries;  j++ ) {
                val2 = 0;
                for (k = 0; a[j].word[k] != '\0'; k++)
                    val1 += a[j].word[k];

                if ((direction == 1 && (val1 > val2)) ||
                    (direction == -1 && (val1 < val2))) {
                    temp = a[i];
                    a[i] = a[j];
                    a[j] = temp;
                }
            }
        }
    }
```

Здесь для определения позиции слова используется то, что коды ASCII возрастают с ростом положения буквы в алфавите. Функция суммирует коды всех букв в слове и сравнивает их с суммой кодов других слов.

11. *Extend the strToInt() function so that if the first character of the string is a minus sign, the value that follows is taken as a negative number.*

Лень делать защиту от неверных данных, нужно быстрее изучать указатели.

```
// Function to convert a string to an integer
int  strToInt (const char  string[])
{
    int  i, intValue, result = 0;
    bool isNegative = false;

    for  ( i = 0; (string[i] >= '0' && string[i] <= '9') || string[i] == '-'; i++ )
    {
        if (string[i] == '-') {
            isNegative = true;
            continue; // so '-' code isn't treated as number
        }

        intValue = string[i] - '0';
        result = result * 10 + intValue;
    }

    if (isNegative)
        result = -result;

    return result;
}
```

Алгоритм прост и уже пояснялся в уроке. Я просто добавил проверку на наличие знака " - " и инверсию знака рассчитанного числа при его обнаружении.

12. *Write a function called strToFloat() that converts a character string into a floating-point value.*

Мне кажется, говнокод вышел. Никакие защиты тоже не делал.

```
    // Function to convert a string to an integer
    double  strToFloat (const char  string[])
    {
        int  i, j, floatPosition = 0, powFraction = 1;
        int intValue, integer = 0;
        double fraction = 0;
        double result = 0;
        bool isNegative = false;
```

```
 9          bool isFractional = false;
10
11          for  ( i = 0; (string[i] >= '0' && string[i] <= '9') ||
12                        string[i] == '-' || string[i] == '.'; i++ )
13          {
14              if (string[i] == '-') {
15                  isNegative = true;
16                  continue; // so '-' code isn't treated as number
17              }
18
19              if (string[i] == '.') {
20                  isFractional = true;
21                  floatPosition = i;
22                  continue;
23              }
24
25              if (!isFractional) {
26                  intValue = string[i] - '0';
27                  integer = integer * 10 + intValue;
28              }
29              else {
30                  intValue = string[i] - '0';
31                  fraction = fraction * 10 + intValue;
32              }
33
34          }
35
36          for ( j = 1; j < (i - floatPosition); j++ )
37              powFraction *= 10;
38
39          result = integer + fraction / powFraction;
40
41          if (isNegative)
42              result = -result;
43
44          return result;
45      }
```

Тип `float` не обладает достаточной точностью, чтобы правильно делить на большие числа, пришлось работать с типом `double`. Функция сначала считает целую часть, потом по обнаружению точки считает дробную часть (в формате целого числа). Затем дробная часть делится на 10 в степени количества чисел в дробной части и прибавляется к целой части.

13. *Write a function called uppercase() that converts all lowercase characters in a string into their uppercase equivalents.*

Как-то совсем примитивно по сравнению с предыдущими упражнениями.

```
 1      // Function to convert a string to an integer
 2      void  uppercase (char  string[])
 3      {
 4          int i;
 5
 6          for ( i = 0; string[i] != '\0'; i++ )
 7              if (string[i] >= 'a' && string[i] <= 'z')
 8                  string[i] = string[i] - 'a' + 'A';
 9
10          return;
11      }
```

14. *Write a function called intToStr() that converts an integer value into a character string. Be certain the function handles negative integers properly.*

Поленился подробно все расписать на бумаге – просидел 4 часа за дебагом.

```
1    // Function to convert a string to an integer
2    void  intToStr (int  number, char result[])
3    {
4        int i = 0;
5        int length = 0;
6        int temp[11] = { 0 }; // because int can't hold more that 10 numbers
7        bool isNegative = false;
8
9        if (number < 0) {
10           isNegative = true;
11           i++;
12
13           number = -number;
14       }
15
16       // get the number into buffer array and count number of digits i
17       do {
18           temp [i] = number % 10;
19           number /= 10;
20           i++;
21       }
22       while (number != 0);
23
24       result [i] = '\0'; // write null character at the end of the string
25       i--;               // move pointer before the null character
26       length = i;
27       if (isNegative) {
28           length++;      // compensate the offset created by minus character
29           result[0] = '-'; // and add the minus character to the beginning
30       }
31
32   // since number in array is written backwards, we have to go from end to start
33       for ( ; i >= 0; i--) {
34           if (isNegative && i == 0)
35               return;
36
37           result[length - i] = temp[i] + '0';
38       }
39
40       return;
41   }
```

Функция использует идею из упражнения 5 урока 1.4 (принятие решений).

Используя оператор `% 10` мы вычленяем последнюю цифру из числа и вставляем ее в первый элемент буферного массива. Затем мы удаляем эту цифру делением числа на 10 (помним про то, что при целочисленном делении отсекается дробная часть). И в конце увеличиваем переменную-счетчик цифр. Таким образом в конце цикла do мы имеем общее количество цифр и массив, хранящий цифры числа в обратном порядке.

Так как индексация массива начинается с нуля, счетчик будет находиться там, где должен стоять нулевой символ, заканчивающий строку. Вписываем нулевой символ и смещаемся на шаг влево.

Если конвертируемое число положительное, то длина строки и текущее значение указателя будут совпадать. Тогда цикл, читающий буферный массив задом наперед начнет заполнять выходной массив с 0-го элемента, что нам и нужно.

Если конвертируемое число отрицательное, то указатель будет равен количеству цифр + 1, потому что добавляется знак минуса. А длину строки нужно приравнять к указателю и добавить еще одну единицу, чтобы заполняющий массив цикл начал с 1-го элемента, а не с 0-го, т.е. нужно сделать так, чтоб `( элементы – указатель ) = 1`.