ШАГ компьютерная академия

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ С

Урок №14

Программирование на языке C

Содержание

1. Работа со строками в C++

Синтаксис объявления строковых массивов и их инициализация.

В прошлых уроках, Вы познакомились с различными видами массивов. Однако сегодня мы более детально рассмотрим еще один тип массивов — строковый, так как этот тип заслуживает особого внимания. Строки предназначены для ввода, обработки и вывода символьной информации.

Строковая константа — это последовательность из нуля или более символов, заключенных в кавычки. Кавычки не являются частью строковой константы, а служат только для ее ограничения.

Строки представляются в виде массива элементов типа char. Это означает, что символы строки можно представить расположенными в соседних ячейках памяти — по одному символу в ячейке. Но массив символов — не всегда строка!

В качестве примера рассмотрим следующую строку: «Символьная строка». Кавычки не являются частью строки. Они вводятся для того, чтобы отметить ее начало и конец, то есть играют ту же роль, что и апострофы в случае одиночного символа (каждая ячейка — 1 байт).

Необходимо отметить, что на рисунке последним элементом массива является символ '\0'. Это нуль-символ: в языке С он используется для того, чтобы отмечать конец строки. Нуль-символ — не есть цифра 0; он не выводится на печать и в таблице кодов ASCII имеет номер 0. Наличие нуль-символа означает, что количество ячеек массива должно быть, по крайней мере на одну больше, чем количество символов, которые необходимо размещать в памяти.

Не следует путать символьную константу со строкой, содержащей один символ: 'X' — это не то же самое, что "X". В первом случае — это отдельный символ. Во втором случае — это строка, состоящая из одного символа (буквы X) и символа конца строки '\0'.

Правила инициализации строковых массивов.

Приведем пример инициализации строкового массива:

```
#include <iostream>
using namespace std;
int n=5;
// Инициализация строкового массива.

char line[5] = { 'C','a','t','!','\0' };
void main ()
{
       cout << "Word: ";
       for (int i=0; i<n; i++)
          cout << line[i];
}
```

Вышеописанный пример не слишком удобен для создания длинных строк. Кроме того, вывод строки побуквенно в цикле выглядит довольно странно, не так ли? Для символьных массивов существует специальный способ

инициализации. Вместо фигурных скобок и запятых можно прямо использовать строку символов, заключенных в кавычки. При этом в описании не обязательно задавать размеры массива, поскольку компилятор «сам» определяет его длину, подсчитывая число начальных значений. А операция вывода cout настроена таким образом, что достаточно указать только имя строкового массива и он тут же отобразится на экране.

```
#include <iostream>
using namespace std;
int n=5;
char line[] = "Cat!"; // Инициализация строкового массива.

void main ()
{
      cout << "Word: ";
      // Показ на экран строкового массива.
      cout <<line;
}
```

2. Взаимосвязь строк и указателей

Мы с Вами уже обсуждали тему указателей в одном из уроков, и Вы должны помнить, как тесно они связаны с массивами. Поэтому сегодня мы так же не сможем обойти указатели стороной. В программе доступ к строке осуществляется с помощью указателя на символ. Если описать переменную message как:

```
char *message;
```

то в результате выполнения оператора

```
message = "and bye!";
```

message станет указателем на строку. Обратите внимания, что оператор cin нельзя будет применять к такому указателю.

```
#include <iostream>
using namespace std;
char *message;
char privet[] = "and bye!";
char *pr = privet;
void main ()
{
      message = "Hello";
      cout << " " << message << " " << pr << "\n";
      int i = 0;
      while  (*(pr+i)!='\0')
      {
         cout<< *(pr+i++) << " ";
      }
}
```

Использование указателей часто применяется при работе с массивом строк. В этом случае к каждой строке можно обратиться с помощью указателя на его первый символ. Это удобно тем, что для перестановки двух строк, расположенных в неправильном порядке, фактически достаточно переставить указатели в массиве указателей, а не сами строки.

Рассматривается функция **month_name()**, которая возвращает указатель на строку, содержащий имя n-го месяца. Это типичная задача для использования строкового массива.

Функция **month_name()** содержит локальный массив строк и при обращении к ней возвращает указатель на нужную строку.

В описании массива указателей на символы **name[]** инициализатором является просто список строк. Символы i-й строки помещаются в определенное место памяти, а указатель на ее начало хранится в элементе **name[i]**. Поскольку размер массива **name** не указан, компилятор сам подсчитывает количество инициализаторов и соответственно устанавливает правильное число.

```
#include <iostream>
using namespace std;
const int n=15;
void main ()
{
      char *month_name(int);
      /* ------------- */
      for (int i=0; i < n; i++)
         cout << "Month number " << i << " - " <<  month_name(i) <<"\n";
 }
/* ------------------------------------------ */
char *month_name (int k)  /* Название k-го месяца */
```

```
{
      static char *name[] = {
                              "none","January",
                              "February","March","April",
                              "May","June","July","August",
                              "September","October","November",
                              "December"
                             };
      return (k<1||k>12)?name[0]:name[k];
}
```

Результат работы программы:

```
Month number 0 - none
Month number 1 - January
Month number 2 - February
Month number 3 - March
Month number 4 - April
Month number 5 - May
Month number 6 - June
Month number 7 - July
Month number 8 - August
Month number 9 - September
Month number 10 - October
Month number 11 - November
Month number 12 - December
Month number 13 - none
Month number 14 - none
Press any key to continue
```

3. Функции работы со строками из библиотеки обработки строк

Здесь мы перечислим основные функции, предназначенные для работы со строками. Большинство прототипов этих функций (если не оговорено особо) находится в заголовочном файле **string.h**.

int getchar(); — Возвращает значение символа (если он есть), который пользователь набрал на клавиатуре. После ввода символа нужно нажать клавишу **Enter**. Заголовочный файл — **stdio.h**

int getch(); — Аналогично предыдущему, только символ на экране не отображается. Используется чаще для организации задержки выполнения программы. Заголовочный файл — **conio.h**

int putchar(int c); — Выводит символ c на экран. В случае успеха возвращает сам символ c, в противном случае — **EOF**. Заголовочный файл — **stdio.h**

char *gets(char *s); — Читает символы, включая пробелы и табуляции, до тех пор, пока не встретится символ новой строки, который заменяется нулевым символом. Последовательность прочитанных символов запоминается в области памяти, адресуемой аргументом **s**. В случае успеха возвращает аргумент **s,** в случае ошибки — нуль. Заголовочный файл — **stdio.h**

int puts(const char *s); — Выводит строку, заданную аргументом **const char *s**. Заголовочный файл — **stdio.h**

char *strcat(char *dest, const char *scr); — Объединяет исходную строку **scr** и результирующую строку **dest**, присоединяя первую к последней. Возвращает **dest**.

char *strncat(char *dest, const char *scr, int maxlen); — Объединяет **maxlen** символов исходной строки **scr** и результирующую строку **dest**, присоединяя часть первой к последней. Возвращает **dest**.

char *strchr(const char *s, int c); — Ищет в строке **s** первое вхождение символа **c**, начиная с начала строки. В случае успеха возвращает указатель на найденный символ, иначе возвращает нуль.

char *strrchr(const char *s, int c); — Аналогично предыдущему, только поиск осуществляется с конца строки.

int strcmp(const char *s1, const char *s2); — Сравнивает две строки. Возвращает отрицательное значение, если **s1<s2;** нуль, если **s1==s2;** положительное значение, если **s1>s2**. Параметры — указатели на сравниваемые строки.

int stricmp(const char *s1, const char *s2); — Аналогично предыдущему, только сравнение осуществляется без учета регистра символов.

int strncmp(const char *s1, const char *s2, int maxlen); — Аналогично предыдущему, только сравниваются первые **maxlen** символов.

int strnicmp(const char *s1, const char *s2, int maxlen); — Аналогично предыдущему, только сравниваются первые **maxlen** символов без учета регистра.

int strcspn(const char *s1, const char *s2); — Возвращает длину максимальной начальной подстроки строки **s1**, не содержащей символов из второй строки **s2**.

int strlen(const char *s); — Возвращает длину строки

s — количество символов, предшествующих нулевому символу.

char *strlwr(char *s); — Преобразует все прописные (большие) буквы в строчные (малые) в строке **s**.

char *strupr(char *s); — Преобразует все строчные (малые) буквы в прописные (большие) в строке s.

char *strnset(char *s, int c, int n); — Заполняет строку **s** символами **c**. Параметр **n** задает количество размещаемых символов в строке.

char *strpbrk(const char *s1, const char *s2); — Ищет в строке **s1** первое вхождение любого символа из строки **s2**. Возвращает указатель на первый найденный символ или нуль — если символ не найден.

char *strrev(char *s); — Изменяет порядок следования символов в строке на обратный (кроме завершающего нулевого символа). Функция возвращает строку **s**.

char *strset(char *s, int c); — Заменяет все символы строки **s** заданным символом **c**.

int strspn(const char *s1, const char *s2); — Вычисляет длину максимальной начальной подстроки строки **s1**, содержащей только символы из строки **s2**.

char *strstr(const char *s1, const char *s2); — Ищет в строке **s1** строку **s2**. Возвращает адрес первого символа вхождения строки **s2**. Если строка отсутствует — возвращает нуль.

char *strtok(char *s1, const char *s2); — Делит исходную строку **s1** на лексемы (подстроки), разделенные одним или несколькими символами из строки **s2**.

double atof(const char *s); — Преобразует строку **s** в число с плавающей точкой типа **double**.

Заголовочный файл — **math.h**

int atoi(const char *s); — Преобразует строку **s** в число типа **int**. Возвращает значение или нуль, если строку преобразовать нельзя. Заголовочный файл — **stdlib.h**

long atol(const char *s); — Преобразует строку **s** в число типа **long**. Возвращает значение или нуль, если строку преобразовать нельзя. Заголовочный файл — **stdlib.h**

char *itoa(int value, char *s, int radix); — Преобразует значение целого типа **value** в строку **s**. Возвращает указатель на результирующую строку. Значение **radix** — основание системы счисления, используемое при преобразовании (от 2 до 36). Заголовочный файл — **stdlib.h**

4. Работа со строками в С. Примеры

Предыдущая тема урока была посвящена строковым функциям, эта тема посвящается работе с ними.

Определение длины строк.

Длина строки определяется просто. Для этого нужно передать строковый указатель функции **strlen()**, которая возвратит длину строки, выраженную в символах. После объявления

```
char *c = "Any old string....";
int len;
```

следующий оператор установит переменную **len** равной длине строки, адресуемой указателем **c**:

```
len = strlen(c);
```

Пример использования функции strlen().

```
#include <stdio.h>
#include <string.h>
#include <iostream>
using namespace std;
const int MAXLEN=256;
void main()
{
        char string[MAXLEN]; /* Место для 255 символов. */
        cout << "Input string:: ";
        gets(string);
        cout <<"\n";/* Начать новую строку. */
        cout << "String: " << string << "\n";
        cout << "Length = " << strlen(string);
}
```

Здесь определяется строковая переменная с именем **string** для приема ввода от функции **gets()**. После того как вы введете строку, программа передаст переменную **string** функции **strlen()**, которая вычислит длину строки в символах.

В функцию **strlen()** можно передавать и другие виды строк. Например, вы можете определить и инициализировать символьный буфер следующим образом:

```
char buffer[128] = "Copy in buffer";
```

Затем используйте функцию strlen() для установки целой переменной len, равной числу символов в литеральной строке, скопированной в буфер:

```
int len;        /* Определить целую переменную. */
len = strlen(buffer); /* Вычислить длину строки. */
```

Копирование строк.

Оператор присваивания для строк не определен. Если c1 и c2 — символьные массивы, вы не сможете скопировать один в другой следующим образом:

```
c1 = c2; //???
```

Но если **c1** и **c2** объявить как указатели типа **char** *, компилятор согласится с этим оператором, но вряд ли вы получите ожидаемый результат. Вместо копирования символов из одной строки в другую оператор **c1 = c2 скопирует указатель c2 в указатель c1**, перезаписав, таким образом, адрес в **c1**, потенциально потеряв информацию, адресуемую указателем.

```
char*c1 = new char [10];
char*c2 = new char [10];
c1=c2;// память выделенная под c1 - потерялась
```

Чтобы скопировать одну строку в другую, вместо использования оператора присваивания вызовите функцию копирования строк **strcpy()**. Для двух указателей **c1** и **c2** типа **char *** оператор

```
strcpy(c1, c2);
```

копирует символы, адресуемые указателем **c2**, в память, адресуемую указателем **c1**, включая завершающие нули. И только на вас лежит ответственность за то, что принимающая строка будет иметь достаточно места для хранения копии.

Аналогичная функция **strncpy()** ограничивает количество копируемых символов. Если источник **(source)** и приемник **(destination)** являются указателями типа **char *** или символьными массивами, то оператор

```
strncpy(destination, source, 10);
```

скопирует до 10 символов из строки, адресуемой указателем **source**, в область памяти, адресуемую указателем **destination**. Если строка **source** имеет больше 10 символов, то результат усекается. Если же меньше — неиспользуемые байты результата устанавливаются равными нулю.

Примечание: Строковые функции, в имени которых содержится дополнительная буква n, объявляют числовой параметр, ограничивающий некоторым образом действие функции. Эти функции безопаснее, но медленнее, чем их аналоги, не содержащие букву n. Программные примеры

содержат следующие пары функций: **strcpy() и strncpy(), strcat() и strncat(), strcmp() и strncmp().**

Конкатенация строк.

Конкатенация двух строк означает их сцепление, при этом создается новая, более длинная строка. При объявлении строки

```
char original[128] = "Test ";
```

оператор

```
strcat(original, " one, two, three!");
```

превратит значение первоначальной строки original в «Test one, two, three!»

При вызове функции **strcat()** убедитесь, что первый аргумент типа **char** * инициализирован и имеет достаточно места, чтобы запомнить результат. Если **c1** адресует строку, которая уже заполнена, а **c2** адресует ненулевую строку, оператор **strcat(c1, c2)**; перезапишет конец строки, вызвав серьезную ошибку.

Функция strcat() возвращает адрес результирующей строки (совпадающий с ее первым параметром) и может использоваться как каскад нескольких вызовов функций:

```
strcat(strcat(c1,c2),c3)
```

Следующий пример показывает, как можно использовать функцию **strcat()** для получения в одной строке фамилии, имени и отчества, хранящихся отдельно, например, в виде полей базы данных. Введите фамилию, имя и отчество. Программа сцепит введенные вами строки и отобразит их как отдельную строку.

```
#include <iostream>
#include <string.h>
using namespace std;
void main()
{
        //Резервирование места для ввода трех строк.
        char *fam = new char[128];
        char *im = new char[128];
        char *otch = new char[128];
        //Ввод данных.
        cout << "Enter" << "\n";
        cout << "\tSurname: ";
        cin >> fam;
        cout << "\tName: ";
        cin >> im;
        cout << "\tLastname: ";
        cin >> otch;
         //Резервирование места для результата.
        //Нужно учесть два пробела и результирующий
        //нулевой символ.
        char *rez=new char[strlen(fam)+strlen(im)+strlen(otch)+3];
        //"Сборка" результата.
        strcat(strcat(strcpy(rez,fam)," "),im);
        strcat(strcat(rez," "),otch);
        //Возврат памяти в кучу.
        delete [] fam;
        delete [] im;
        delete [] otch;
        //Вывод результата.
        cout << "\nResult: " << rez;
        delete [] rez;
}
```

Приведенная программа демонстрируют важный принцип конкатенации строк: всегда инициализируйте первый строковый аргумент. В данном случае символьный массив **rez** инициализируется вызовом функции **strcpy()**, которая вставляет **fam** в **rez**. После этого программа добавляет пробелы и две другие строки — **im** и **otch**. Никогда не вызывайте функцию **strcat()** с не инициализированным первым аргументом.

Если вы не уверены в том, что в строке достаточно места для присоединяемых подстрок, вызовите функцию

strncat(), которая аналогична функции **strcat()**, но требует числового аргумента, определяющего число копируемых символов. Для строк **s1** и **s2**, которые могут быть либо указателями типа **char** *, либо символьными массивами, оператор

```
strncat(s1, s2, 4);
```

присоединяет максимум четыре символа из **s2** в конец строки **s1**. Результат обязательно завершается нулевым символом.

Существует один способ использования функции **strncat()**, гарантирующий безопасную конкатенацию. Он состоит в передаче функции **strncat()** размера свободной памяти строки-приемника в качестве третьего аргумента. Рассмотрим следующие объявления:

```
const int MAXLEN=128
char s1[MAXLEN] = "Cat";
char s2[] = "in hat";
```

Вы можете присоединить **s2** к **s1**, формируя строку **«Cat in hat»**, с помощью функции **strcat()**:

```
strcat(s1, s2);
```

Если вы не уверены, что в **s1** достаточно места, чтобы запомнить результат, используйте альтернативный оператор:

```
strncat(s1, s2, (MAXLEN-1)-strlen(s1));
```

Этот способ гарантирует, что **s1** не переполнится, даже если **s2** нужно будет урезать до подходящего размера. Этот оператор прекрасно работает, если **s1** — нулевая строка.

Часто программам приходится выполнять поиск в строках отдельных символов или подстрок, особенно при проверке имен файлов на заданное расширение. Например, после того как пользователю предложили ввести имя файла, проверяется, ввел ли он расширение .**TXT**, и если это так, то выполняется действие, отличное от того, какое было бы выполнено для расширения .**EXE**.

Возможно, вы также захотите отвергнуть все расширения, кроме определенного, что поможет вам предотвратить ошибки, вызванные загрузкой файла данных не желаемого типа.

Поиск символов.

Пример использования функции **strchr()**.

```
#include <iostream>
#include <string.h>
#include <stdio.h>
using namespace std;
void main()
{
  char *filename = new char[128];
  cout << "Enter name of file: ";
  gets(filename);
  cout << "\nName of file: " << filename << "\n";
  if (strchr (filename,'.'))
     cout << "Name has extension" << "\n";
  else
     strcat (filename,".TXT");
  cout << "Name of file: " << filename << "\n";
  delete [] filename;
}
```

Данная программа находит расширение в имени файла, выполняя поиск точки среди символов введенной строки. (В имени файла может быть только одна точка, которая должна предшествовать расширению, если оно имеется.) Ключевым в этой программе является оператор

```
if (strchr (filename,'.'))
     cout << "Name has extension" << "\n";
else
     strcat (filename,".TXT");
```

Выражение **strchr (filename,'.')** возвращает указатель на символ точки в строке, адресуемой указателем **filename**. Если такой символ не найден, функция **strchr()** возвращает нуль. Поскольку ненулевые значения означают «истину», вы можете использовать функцию **strchr()** в качестве возвращающего значения «истина»/«ложь». Вы также можете применить функцию **strchr()** для присваивания указателя на подстроку, начинающуюся с заданного символа. Например, если **p** — указатель, объявленный как **char ***, и указатель **filename** адресует строку **TEST.TXT**, то результат действия оператора **p=strchr(filename, '.');** показан на рисунке

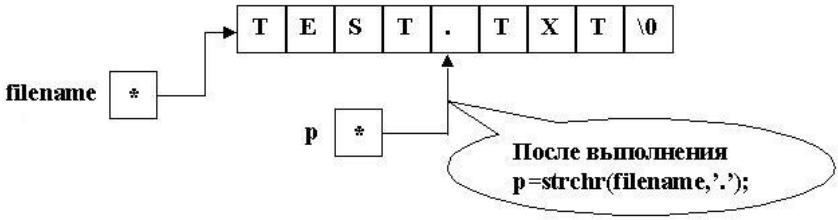

Рисунок демонстрирует еще один важный момент, связанный с адресацией указателем не полной строки, а ее части — подстроки. Такими указателями следует пользоваться с большой осторожностью. На рисунке показана только одна строка, **TEST.TXT**, оканчивающаяся нулевым байтом, но два указателя — **filename** и **p**. Указатель **filename** адресует полную строку. А указатель p адресует

подстроку внутри того же набора символов. Строковые функции не заботятся о байтах, которые предшествуют их первому символу. Поэтому оператор

```
puts(p);
```

отображает подстроку .**TXT** так, будто она полная строковая переменная, а не часть другой строки.

В программировании на C нет ничего необычного в использовании многих указателей, адресующих подстроки одной и той же полной строки. Но строка, показанная на рисунке, расположена в куче, поэтому оператор

```
delete [] p;
```

пытаясь тем самым освободить подстроку, адресуемую указателем **p**, что, несомненно, приведет к разрушению кучи, вызвав ошибку, относящуюся к разряду трудно обнаруживаемых.

Функция **strchr()** отыскивает первое появление символа в строке. Объявления и операторы

```
char *p;
char s[]="Abracadabra";
p = strchr(s,'a');
```

присваивают указателю **p** адрес первой строчной буквы 'a' в строке «Abracadabra».

Функция **strchr()** рассматривает завершающий нуль строки как значащий символ. Приняв во внимание этот факт, можно узнать адрес конца строки. Учитывая предыдущие объявления, оператор

```
p = strchr(s,0);
```

установит указатель p равным адресу подстроки «bra» в конце строки «Abracadabra».

Поиск подстрок.

Кроме поиска символов в строке, вы также можете поохотиться и за подстроками. Этот пример демонстрирует этот метод. Данная программа аналогична предыдущей, но устанавливает расширение файла **.TXT**.

```
#include <iostream>
#include <string.h>
#include <stdio.h>
using namespace std;
void main()
{
  char *filename = new char[128],*p;
  cout << "Enter name of file: ";
  gets(filename);
  cout << "\nName of file: " << filename << "\n";
  strupr(filename);
  p = strstr (filename,".TXT");
  if (p)
     cout << "Name has extension" << "\n";
  else
   { p = strchr (filename,'.');
     if (p)
        *p=NULL; //Удалить любое другое расширение.
     strcat (filename,".TXT");
   }
  cout << "Name of file: " << filename << "\n";
  delete [] filename;
}
```

Эта программа создает имя файла, которое обязательно заканчивается расширением .TXT. Чтобы определить, есть ли в имени файла это расширение, программа выполняет оператор

```
p = strstr (filename,".TXT");
```

Подобно **strchr()**, функция **strstr()** возвращает адрес подстроки или нуль, если искомая строка не найдена. Если же цель будет обнаружена, указатель **p** установится равным ее адресу, в данном примере — адресу точки в подстроке **.TXT**. Поскольку расширение может быть введено и строчными буквами, программа выполняет оператор

```
strupr(filename);
```

чтобы перед вызовом функции **strstr()** преобразовать буквы оригинальной строки в прописные.

Пример также демонстрирует способ усечения строки в позиции заданного символа или подстроки. Здесь вызывается функция **strstr()**, чтобы установить указатель p равным адресу первой точки в строке **filename.** Если результат этого поиска не нулевой, то выполнится оператор, который запишет вместо точки нулевой байт:

```
*p = NULL;
```

Тем самым будет присоединен новый конец строки в том месте, где раньше находилось расширение файла. Теперь строка готова к добавлению нового расширения путем вызова функции **strcat().**

5. Примеры задач на новый материал

В данном разделе урока, мы приготовили для Вас несколько интуитивно понятных примеров с использованием указателей.

Программа для замены в слове X всех букв «а» на сочетание «ky».

```
#include <string.h>
#include <stdio.h>
void main ()
{
        /*
                k - переменная для прохода по оригинальному массиву
                i - переменная для прохода по результативному массиву
                n - длина оригинального массива
        */
        int k=0,i=0, n;
        /*
                x1 - оригинальный массив
                x2 - результативный массив (больше в два раза, на
                        случай, если оригинальный весь заполнен буквами 'a')
                px1 - указатель для перемещения по оригинальному массиву
                px2 - указатель для перемещения по результативному массиву
        */
        char x1[40],x2[80],*px1,*px2;

        // Запрос на ввод оригинального массива
        puts( "Enter word (max 39 letters) ");
        gets(x1);

        /*
        записываем адреса начала
        оригинального  и результативного
        массивов в указатели
        */
        px1 = x1;
        px2 = x2;

        /*
        вычисляем реальную длину
```

```
        оригинального массива
        */
        n = strlen(x1)+1;

        // цикл поэлементно перебирает
        // оригинальный массив
        while  (k<n)
        {
                // если значение текущего элемента
                // не совпадает с 'a'
                if (*(px1+k)!='a')
                {
                        // копируем текущий элемент
                        // в результативный массив
                        *(px2+i) = *(px1+k);
                        // переходим к следующим элементам
                        i++;
                        k++;
                }
                // если значение текущего элемента
                // совпадает с 'a'
                else
                {
                        // записываем символ 'k' в текущую
                        // позицию результативного массива
                        *(px2+i) = 'k';
                        // записываем символ 'y' в следующую
                        // позицию результативного массива
                        *(px2+i+1) = 'y';
                        // переходим к следующему элементу
                        // оригинального массива
                        k++;
                        // "перепрыгиваем" через один элемент
                        // результативного массива
                        i += 2;
                }
        }
        // демонстрируем результативный массив
        puts(x2);
}
```

Программа для замены всех сочетаний «ky» в слове X на букву «a».

```
#include <string.h>
#include <stdio.h>
void main ()
{
   /*
```

```
        k - переменная для прохода по оригинальному массиву
        i - переменная для прохода по результативному массиву
        n - длина оригинального массива
*/
int k=0,i=0, n;
/*
        x1 - оригинальный массив
        x2 - результативный массив
        px1 - указатель для перемещения по оригинальному массиву
        px2 - указатель для перемещения по результативному массиву
*/
char x1[40],x2[40],*px1,*px2;

// Запрос на ввод оригинального массива
puts( "Enter word (max 39 letters) ");
gets(x1);

/*
записываем адреса начала
оригинального  и результативного
массивов в указатели
*/
px1 = x1;
px2 = x2;

/*
вычисляем реальную длину
оригинального массива
*/
n = strlen(x1)+1;

// цикл поэлементно перебирает
// оригинальный массив
while (k<n)
{
        // проверяем, если два символа
        // в текущей позиции оригинального
        // массива не совпадают с буквосочетанием "ky"
        if (strncmp((px1+k),"ky",2)!=0)
        {
                // просто копируем один символ
                // из текущей позиции в
                // результативный массив и
                // передвигаемся на символ вперед
                *(px2+i++) = *(px1+k++);
        }
        // если же два символа
        // в текущей позиции оригинального
        // массива совпадают с буквосочетанием "ky"
        else
        {
```

```
                        // записываем в результативный массив
                        // символ 'a', и переходим на один
                        // символ вперед. В оригинальном переходим
                        // на два символа вперед
                        *(px2+i++) = 'a';
                        k += 2;
                }
        }
        // демонстрируем результативный массив
        puts(x2);
}
```

Программа, удваивающая каждую букву слова X.

```
#include <string.h>
#include <stdio.h>
void main ()
{
        // n - длина оригинального массива *2
        int n;

        /*
                x1 - оригинальный массив
                x2 - результативный массив (больше в два раза)
                px1 - указатель для перемещения по оригинальному массиву
                px2 - указатель для перемещения по результативному массиву
        */
        char x1[40],x2[80],*px1,*px2;

        // Запрос на ввод оригинального массива
        puts( "Enter word (max 39 letters) ");
        gets(x1);
        /*
        записываем адреса начала
        оригинального  и результативного
        массивов в указатели
        */
        px1 = x1;
        px2 = x2;
        /*
        вычисляем двойную длину
        оригинального массива
        */
        n = 2*strlen(x1);

        // записываем в последний элемент
        // результативного массива '\0'
    *(px2+n) = '\0';
```

```
        // цикл поэлементно перебирает
        // оригинальный массив
    while  ((*px1)!='\0')
    {
                // записываем значение из текущей позиции
                // оригинального массива в текущую позицию
                // результативного массива, в последнем
                // переходим на один элемент вперёд
                *px2++ = *px1;
                // записываем значение из текущей позиции
                // оригинального массива в текущую позицию
                // результативного массива, в обоих массивах
                // переходим на один элемент вперёд
                *px2++ = *px1++;
    }
        // демонстрируем результативный массив
        puts(x2);
}
```

6. Домашнее задание

1. Пользователь вводит строку с клавиатуры в фиксированный массив. Необходимо проверить, сколько элементов массива теперь занято и сколько свободно.

2. Показать на экран с m по n символов строки, введенной пользователем и записать данный отрезок в другой массив. (m и n также вводятся пользователем)

3. Удалить с m по n символов, перезаписать строку и показать ее на экран.

4. Пользователь вводит отдельно строку и символ, необходимо показать на экран номера по порядку всех совпадений (нумерация с единицы).

5. Пользователь вводит отдельно строку и символ, необходимо показать на экран номер только последнего совпадения (нумерация с единицы).