ЛЕКЦИЯ 12.

Строки

Строка – конечная последовательность символов. Количество символов в строке называется её длиной (текущей длиной). Допустимы строки нулевой длины.

- Основные операции над строками:
- поиск символов в строке;
- замена символов в строке;
- поиск, замена, удаление подстрок;
- вставка в строку новой подстроки;
- сцепление (конкатенация) двух строк.

В результате выполнения этих операций длина строки может измениться!

Средства C++ для работы со строками

Для работы со строками символов язык C++ предоставляет две возможности:

- работу с массивами данных типа **char** (функции для работы с такими данными описаны в файлах **<string.h>** или **<cstring>**
- класс **string**, описанный в файле **<string>**

Работа со строками фиксированной длины

Если длина строки известна, не равна нулю и не изменяется, для её хранения можно использовать массив соответствующего размера.

Пример: номер зачётной книжки всегда состоит из 7 символов

```
char nz[7];
nz[0] = '1'; nz[1] = '0'; nz[2] = '2'; nz[3] = '3';
nz[4] = '1'; nz[5] = '6'; nz[6] = '9';
…
for (int i=0; i<7; i++)
  cout << nz[i];
```

Неудобства такого подхода – строку можно обрабатывать только посимвольно!

Нуль-терминированные строки

Нуль-терминированные строки – строки, в которых символ с кодом 0 является признаком конца строки и не входит в её состав.

Нуль-терминированные строки используются:

- в строковых константах;
- при вводе-выводе;
- в стандартных функциях, описанных в файле **<string.h>**

Примеры строковых констант:

```
"T"
"This is \”string \”.”
"This is character string"
"Это строковая константа"
"A" "1234567890" "0" "$"
```

\" внутри строки означают, что в строку входит символ ".

Символы строки записываются в последовательные ячейки памяти. В конце каждой строки компилятор помещает нулевой символ '\0'. Строковые константы размещаются в статической памяти. Вся фраза в кавычках является указателем на место в памяти, где записана строка (аналогично имени массива).

Определение и инициализация строк

Строка объявляется как массив символов:

```
char *str;
```

или

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

При инициализации строки размерность массива лучше не указывать.

```
char *str="This is a string";
char str[]="This is a string“;
```

Такой вид инициализации является краткой формой стандартной инициализации массива

```
char  str[]={’T’,’h’,  ’i’,  ’s’,  ’  ’,   ’i’,  ’s’,  ’  ’,
’a’,’  ’  , ’s’, ’t’, ’r’, ’i’, ’n’,’g’,’\0’};
```

Без символа 0 имеем обычный массив символов, а не строку.

Как и для других массивов, имя str является указателем на первый элемент массива:

```
str == &str[0], *str == 'T', *(str+1) == str[1] == 'h'
```

Массив и указатель: различия

```
char h1[ ] =
"Программирование на языке Си++";
char *h2 =
"Программирование на языке Pascal";
```

Основное отличие состоит в том, что указатель **h1** является **константой,** в то время как указатель **h2 - переменной**.

В первом случае создается массив из 31 символа в статической памяти (по

одному на каждый символ строки +1 на ’\0’).

h1 – синоним адреса первого элемента массива, т.е. **&h1[0].**

Нельзя изменить **h1,** так как это означало бы изменение адреса массива в памяти.

Можно использовать **h1+1** (выбор следующего символа строки), но нельзя использовать выражение **++ h1.**

В случае с указателем также создается в статической памяти массив для строки (из 33 символов). Но кроме этого выделяется еще место для переменной - указателя **h2**. Сначала эта переменная указывает на начало строки, но ее значение может изменяться: **++h2** будет указывать на второй символ строки.

Использование операции сложения с указателем для выбора следующих символов строки допустимо для обоих вариантов:

```
for(int i=0;i<16;i++)
    printf("%c",*(h1+i));
printf("\n");

for(i=0;i<16;i++)
    printf("%c",*(h2+i));
printf("\n");
```

В результате получаем

Программирование
Программирование

Операцию увеличения можно использовать **только для указателя:**

```
while ((*h2) != '\0')
    printf("%c",*(h2++));
    //Программирование на языке Си++
```

Если нужно заменить **h2** на **h1,** можно так: **h2=h1;**

Но **h1 = h2;** - запрещенная конструкция .

Ситуация аналогична **x = 5** или **5 = x**.

Левая часть оператора присваивания должна быть именем переменной.

Инструкция **h2 = h1** не уничтожит строку про язык Си, а только изменит адрес, записанный в **h2**.

Инструкции

```
h1[13] ='C';     и  *(h1+8)='C';
```

позволяют изменять элементы первой строки.

Строка встроенного типа может считаться пустой в двух случаях: если указатель на строку имеет нулевое значение (тогда у нас вообще нет никакой строки) или указывает на массив, состоящий из одного нулевого символа (то есть на строку, не содержащую ни одного значимого символа).

```
// pc1 не адресует никакого массива символов
char *pc1 = 0;
// pc2 адресует нулевой символ
const char *pc2 = "";
```

Ввод-вывод строк

1. Использование операторов >> и <<

```
const int MAX=80;
char name[MAX];
cout<<" Привет, как вас зовут?"<<endl;
cin>>name;
cout<<"Хорошее имя ">>name>>endl;
```

Оператор >> вводит символы до тех пор, пока не будет нажата клавиша ввода, затем в строку дописывается символ '\0'. Таким образом в **name** можно ввести до 79 символов.

Оператор << выводит символы до конечного '\0'.

Что произойдет, если ввести более 79 символов?

Так как в C++ нет проверки выхода индекса за границы массива, можно получить непредсказуемые результаты.

Однако существует способ защиты от ввода лишних символов:

```
…
cin >> setw(MAX) >> name;
```

Метод setw определяет максимальное количество символов, которое можно ввести, включая '\0'. Таким образом в **name** можно ввести до 79 символов.

2. Использование метода get

Если в предыдущем примере введем строку, содержащую пробелы внутри, то введется только первое слово, ибо оператор **<<** считает пробел нулевым символом.

```
// ввод строки с пробелами
cin.get(name,MAX);
```

Если в нужно ввести в строку несколько строк с клавиатуры, то нужно задать 3-ий аргумент – признак конца последней строки

```
// ввод нескольких строк
// $-признак конца ввода
cin.get(name,MAX,$);
```

3. *Использование функций gets и puts*

```
char *gets (char *s);//ввод строки
```

Функция читает символы до тех пор, пока не встретится символ новой строки ‘\n’, который создается при нажатии клавиши ввода. Функция берет все символы до ‘\n’ (не включая его) и присоединяет к ним символ ‘\0’.

```
int puts(const char *s);//вывод строки
```

```
gets(name);
puts(name);
```

При неудачном завершении работы функция **gets** возвращает значение **NULL.**
Поэтому можно проверять правильность ввода и определять конец ввода (EOF) :

```
while (gets(name)!=NULL)…
```

При неудачном завершении работы функция **puts** возвращает значение **EOF,** иначе – ненулевое число.

puts - записать строку в стандартный файл вывода **stdout.**
Прекращает запись, когда находит нуль-символ. Далее этот нуль-символ заменяется на символ новой строки.

Обработка строк

Доступ к строке осуществляется с помощью указателя типа char.

Указатель указывает на соответствующий строке массив символов. Даже когда пишем строковый литерал, например

```
const char *st=”English\n”;
```

компилятор помещает все символы строки в массив и затем присваивает переменной **st** адрес первого элемента массива. Для перебора символов строки используется адресная арифметика, например так:

```
While (*st++) {…}
```

Функции определения длины строки:

```
int string_length (const char *st)
{int length_st=0;
 if (st)
     while (*st++)
       ++length_st;
 return length_st;}
/*********************************/
int str_length (const char *st)
```

```
{int length_st=0;
 if (st)
   for (int i=0; st[i]!='\0';
    length_st++, i++);
 return length_st;}
```

Функции для работы со строками из заголовочного файла <string.h>

`char* strcat(char *s1, const char *s2)` (сцепление строк).
Функция добавляет s2 к s1 и возвращает s1. В конец результирующей строки добавляется нуль-символ Функция не проверяет, умещается ли результат в первом массиве.

`char* strchr(char *s, int ch)`
Функция возвращает указатель на первое вхождение символа ch в строку s, если его нет, то возвращается NULL.

`char* strrchr(const char *s, int ch)`
Ищет последнее вхождение ch в s.

`int strcmp(const char *s1, const char *s2)`
Функция сравнивает строки и возвращает отрицательное (если s1 меньше s2), нулевое (если s1 равно s2) или положительное (если s1 больше s2) значение. Строки сравниваются лексикографически.

`char* strcpy(char *s1, const char *s2)`
Функция копирует s2 в s1 и возвращает s1.

`size_t strlen(const char *s)`
Функция возвращает длину строки (без учета символа завершения строки).

`char* strncat(char *s1, const char *s2, size_t n)`
Функция добавляет не более n символов из s2 к s1 и возвращает s1. Первый символ s2 пишется на место завершающего нуль-символа строки s1. Если длина строки s2 меньше n, переписываются все символы s2. К строке s1 добавляется нуль-символ. Если строки перекрываются, поведение не определено.

`int strncmp(const char *s1, const char *s2, size_t n)`
Функция сравнивает первую строку и первые n символов второй строки и

возвращает отрицательное (если s1 меньше s2), нулевое (если s1 равно s2) или положительное (если s1 больше s2) значение.

`char* strncpy(char *s1, const char *s2, size_t n)`

Функция копирует не более n символов из s2 в s1 и возвращает s1. Если длина исходной строки превышает или равна n, нуль-символ в конец строки s1 не добавляется. В противном случае строка дополняется нуль-символами до n-го символа. Если строки перекрываются, то результат не предсказуем.

`size_t strspn(const char *s1, const char *s2)`

Функция возвращает индекс первого символа из s1, отсутствующего в s2.
Примеры :

```
strspn(”abracadabra”, ”abr”) // результат - 4
strspn(”abracadabra”, ”abc”) // результат - 2
strspn(”abracadabra”, ”abcdr”)//результат - 11
```

`char* strstr(char *s1, const char *s2)`

Функция выполняет поиск первого вхождения подстроки s2 в строку s1. В случае удачного поиска, возвращает указатель на элемент из s1, с которого начинается s2, и NULL в противном случае.

`char* strtok(const char *str1, const char *str2)`

Функция выделения из строки s1 лексем, разделенных любым из множества символов, входящих в строку s2, возвращает указатель на следующую лексему, или NULL, если больше нет лексем.

При первом вызове в качестве str1 передается адрес строки. В последующих вызовах для этой строки в качестве str1 передается NULL. После нахождения лексемы функция strtok вместо разделителя записывает "\0".

Общие замечания по использованию стандартных функций

- Все функции, изменяющие строки, не проверяют правильность выделения памяти. Ошибки не вызывают никаких сообщений компилятора!
- MS Visual Studio считает большинство перечисленных функций небезопасными (deprecated) и предлагает свои, более надёжные функции(например, strcpy_s вместо strcpy). Однако это может привести к непереносимости программы на другие компиляторы!

Особенности работы с нуль-терминированными строками

При формировании строки для совместимости со стандартными средствами языка нужно правильно размещать признак конца строки. Так, результат вывода следующей программы не определен:

```
char s[20];
s[0]=1;
cout << strlen(s) << endl;
```

Другой особенностью работы со строками является отсутствие каких-либо проверок на действительную длину строки. Так, написав

```
char *s;
cin >> s;
```

скорее всего, что-то куда-то введем… Ошибка может обнаружиться гораздо позднее, и совсем не там, где ее совершили…

Еще одна типичная ошибка в работе со строками связана с тем, что присваивание двух строковых переменных в Паскале и С трактуется по-разному. Для иллюстрации этого приведем фрагмент программы на Паскале и попытаемся дословно перевести ее в С.

В Паскале:

```
Var
 S1, S2: string;
…
readln(S1);
S2 := S1;
…
// изменяем S1
…
S1 := S2;
// S1 восстановлена …
```

В С:

```
char *S1 = new char[255],
*S2 = new char[255];
…
cin >> S1;
S2 = S1; // а надо было strcpy(S2, S1);
…
// изменяем S1
…
S1=S2;
// увы, S1 не восстановлена …
```

Пример 13.1. *Слова в строке разделены одним или более пробелами.В начале и в конце строки пробелы могут отсутствовать. Подсчитать количество слов.*

```
char str3[]=" Слова  в  строке разделены одним или
более пробелами. Подсчитать количество слов";
int count=0,i;
if (str3)
{for (i=0; str3[i]!='\0';  i++)
  if ((str3[i]!=' ')
      &&((i==0)||(str3[i-1]==' ')))
          count++;
  cout<<"count="<<count<<endl;
 }
```

Пример 13.2. *Подсчитать, сколько раз в строке встречается заданная подстрока.*

```
…
char A[]=" Дана строка и подстрока. Подсчитать, сколько
  раз в строке   встречается заданная подстрока";
char B[]="строка";
int count1=0;
char* s, *s1;
puts(A);
int n=strlen(B);    //Длина подстроки
s1=A;               //адрес начала строки
while((s=(char*)strstr(s1,B))!=NULL)
 //Пока в строке есть подстроки:
{
    s1=s; //сохраняем адрес удаляемого слова
          //отсюда дальше будем искать следующее слово
    while(*(s+n)!=NULL)          //Сдвигаем строку влево
    {
      *s=*(s+n);
      s++;
    }
    *s=NULL; //Ограничиваем ее
    count1++; //число подстрок
}
//Вывод результата
printf("%d раза\n", count1);
```

…

Пример 13.3. ***Ввести строку длиной не более 100 символов и перевернуть ее (т.е. вместо строки «мама мыла раму» получить строку «умар алым амам»).***

Обратите внимание на особенность ввода строк, которые могут содержать пробелы.

```
#include <iostream>
#include <cstring>
using namespace std;
int main (void) {
char *s = new char[101], c;
cin.getline(s, 101);
// cin >> s; - неверно!
int len = strlen(s);
if (len>0)
 {
    for (int i=0; i<len-1; i++, len--)
    {
      c=s[len-1];
      s[len-1]=s[i];
      s[i]=c;
    }
 }
 cout << s << endl;
 return 0;
}
```

Пример 13.4. **Разделить строки на лексемы (выделить и распечатать все слова).**

```
char string[] = "A string\tof ,,
 tokens\nand some  more tokens"; //строка
char seps[]= " ,\t\n"; //разделители
char *token;
printf( "Tokens:\n" );
// поиск первого слова:
token = strtok(string, seps);
while(token != NULL)
   {printf(" %s\n", token);
     // выбор следующего слова:
    token = strtok(NULL, seps);
```

```
    }
```

Пример 13.5.

/*с ошибкой*/

```
/* попробуйте удалить "BAZA" из "BABAZAZASSS"  */
void delstr(const char*A,const char*B)
{//функция удаления подстроки
 char* s;
 int n=strlen(B); //Длина подстроки
 while((s=(char*)strstr(A,B))!=NULL)
                  //Пока в строке есть заданные подстроки:
 {  while(*(s+n)!=NULL)
                  //Сдвигаем всю строку влево
    {             //стирая найденную подстроку
     *s=*(s+n);
      s++;
    }
   *s=NULL;       //Ограничиваем ее
 }}
```

/*Исправленный вариант*/

```
void delstr_new(char*A,const char*B)
{
 char* s,*s1;
 int n=strlen(B);          //Длина подстроки
 s1=A;        //адрес начала строки
 while((s=(char*)strstr(s1,B))!=NULL)
              //Пока в строке есть заданные подстроки:
 {
   s1=s;      //сохраняем адрес удаляемого слова
              //отсюда дальше будем искать следующее слово
   while(*(s+n)!=NULL)      //Сдвигаем всю строку влево
   {                        //стирая найденную подстроку
     *s=*(s+n);
     s++;
   }
   *s=NULL;                 //Ограничиваем ее
 }
```

КОНЕЦ ЛЕКЦИИ

Материал для самостоятельного изучения

Примеры на тему "Указатели и строки"

Большинство операций языка Си, имеющих дело со строками, работают с указателями.

1. Рассмотрим бесполезную, но поучительную программу:

```
/* Указатели и строки */
void main( )
{
 char *mesg = "Message";
 char *copy;
 copy = mesg;
 printf("%s\n",copy);
 printf("X = %s; значение = %u;
      &X = %u\n",mesg,mesg,&mesg);
 printf("X = %s; значение = %u;
      &X = %u\n",copy,copy,&copy);
}
```

Это **не копирование** строки.

На выходе получим:

Message

X = Message; значение = 4290432; &X = 1244960

X = Message; значение = 4290432; &X = 1244948

Третьим элементом в каждой строке является &X, т. е. адрес X.

Указатели `mesg` и `copy` записаны в ячейках 124948 и 124960, соответственно.

Второй элемент `значение=`. Это сам указатель . Значением указателя является адрес, который он содержит.

Оператор `copy = mesg;` создает второй указатель, ссылающийся на ту же самую строку. Сама строка не копируется.

2. /* Строки в качестве указателей */

```
void main( )
{
 printf("%s, %u, %c\n", "This ",
```

```
        "character", *"string");
 printf("%s", "This\n\tstring\n");
}
```

Вот что получится на экране:

```
This , 4281976, s
This
    string
```

Почему?

Формат **%s** выводит строку **"This ".**

Формат **%u** выводит целое без знака, но так как строка **"character",** является указателем, то выдается его значение, являющееся адресом первого символа строки.

По формату **%c** выводится первый символ строки **"string"** (ибо ***"string" -** разыменование указателя **"string").**

Примеры использования функций

Пример 1:

```
/* сцепить строки */
#include <string.h>
void main(void)
{
   char destination[25]=”This”;
   char *blank = " ", *c = "C++";
   strcat(destination, blank);
   strcat(destination, c);
   printf("%s\n", destination);
}
```

Пример 2:

```
#include <string.h>
void main(void)
{  char destination[25];
   char *source = "structured ";
   strcpy(destination, "programming");
   strncat(destination, source, 11);
}
```

Пример 3:

```
#include <string.h>
#include <stdio.h>
void main(void)
{  char *buf1 = "aaa", *buf2 = "bbb",;
   int ptr;
   ptr = strcmp(buf2, buf1);
   if (ptr > 0)
    printf("buf2 is greater than buf1\n");
   else
    printf("buf2 is less than buff1\n"); }
```

Пример 4:

```
int strncmp(char *str1, const char *str2, size_t n);

void  main(void)
{  char *buf1 = "aaabbb",
                *buf2 = "bbbccc";
   int ptr;
   ptr = strncmp(buf2,buf1,3);
   if (ptr >0)
    printf("buf2 is greater than buf1\n");
   else
    printf("buf2 is less than buf1\n");}
```

Пример 5:

```
void main(void)
{ char string[10];
  char *str1 = "abcdefghi";
  strcpy(string, str1);
  printf("%s\n", string);}
```

Пример 6:

```
void main(void)
{  char string[10];
```

```
   char *str1 = "abcdefghi";
   strncpy(string, str1, 3);
   string[3] = '\0';
   printf("%s\n", string);}
```

Пример 7:

```
#include <stdio.h>
#include <string.h>
void main(void)
{  char *string = "Borland International";
   printf("%d\n", strlen(string));}
```

Пример 8:

```
void main(void)
{ char string[20];char *ptr,c = 'r';
  strcpy(string,"This is a string");
  ptr = strchr(string, c);
  if (ptr)
    printf("The character %c is at
     position: %d\n", c, ptr);
  else
    printf("The character was not found\n");}
```

Пример 9:

```
void main(void)
{char string[20];
 char *ptr, c = 'r';
 strcpy(string, "This is a string");
 ptr = strrchr(string, c);
…
}
```

strpbrk - найти в строке str1 первое появление любого из множества символов, входящих в строку str2 (исключая завершающий знак NULL)

strpbrk возвращает указатель на найденный в str1 символ или NULL-указатель, если знаков из str2 в str1 нет.

Пример 10:

```
char* strpbrk(const char *str1, const char *str2);
char str[80]="123 abc";
char *st1, *st2;
st1=strpbrk(str,"bca");
st2=strpbrk(str,”1bca");
    //st1= abc
    //st2 =123 abc
```

Пример 11:

```
#include <stdio.h>
#include <string.h>
void main(void) {
  char *string1 = "abcdemnopqrstu";
  char *string2 = "onm";
  int *ptr;
  ptr = strpbrk(string1, string2);
  if (ptr)
     printf(“found firstcharacter:%c\n",
         ptr);
  else
      printf("didn't find character in set\n");}
```

strspn - определить длину подстроки (отрезка) строки str1, содержащего символы из множества, входящих в строку str2

```
size_t strspn(const char *str1, const char *str2 );
```

Пример 12:

```
void main(void)
{char *string1 = "1234567890";
   char *st1 = "123DC8", *st2="123D45C8";
   char *st3 = "123DC584176";
   int len1,len2, len3;
   len1 = strspn(string1, st1);
   len2 = strspn(string1, st2);
   len3 = strspn(string1, st3); }
 //   len1=3 len2=5 len3=8
```

strcspn - определить длину начальной части отрезка строки str1, не содержащего символы строки str2

size_t strcspn(const char *str1, const char *str2);

Пример 13:

```
void main(void)
{   char *string1 = "1234567890";
    char *string2 = "747DC8";
    char *string3 = "7471DC8";

    int len1,len2;
    len1 = strcspn(string1, string2);
    len2 = strcspn(string1, string3);
    // len1=3 len2=0
```

strstr - Находит первое вхождение str2 в str1

```
char* strstr(const char *str1, const char *str2);
```

Находит первое вхождение str2 в str1 (без "\0"). В случае успеха возвращает указатель на найденную подстроку, в случае неудачи – 0.

Если str1 указывает на строку нулевой длины, то возвращает указатель на str1.

```
char str[80]=”123 abc 456”;
printf (”%s\n”,strstr(str,”abc”));
//печать: abc 456
```

Пример 14:

```
void main(void)
{   char input[16] = "abc,d,ef";
    char *p;
    p = strtok(input, ",");
    if (p) printf("%s\n", p);
    p = strtok(NULL, ",");
    while(p != NULL)
    {printf(" %s\n", p);
      // выбор следующего слова:
     p = strtok(NULL, ”,”);
    }
}
```

Функции для работы с памятью

void *memchr(const void *str, int c, size_t n);

Ищет первое вхождение c в n байтах str.

int memcmp(conct void *str1, const void *str2, size_t n);

Сравнивает первые n байт str1 и str2.

void *memcpy(void *str1, const void *str2,size_t n);

Копирует n байт из str2 в str1

void *memmove(void *str1, const void *str2,size_t n);

Копирует n байт из str2 в str1, перекрыв. строк

void *memset(void *str1, int c,size_t n);

Копирует c в n байт в str1.

Функции преобразования строки в числовые данные

Определены в <stdlib.h>

```
int atoi (const char *str);
long int atol (const char *str);
double atof(const char *str);
```

На вход они принимают указатель на строку, завершенную нулем, а возвращают - число, которое этой строкой описывается.

atoi и atol воспринимают следующий формат числа:

[*пробелы*][*знак*]*цифры*

atof, соответственно:

[*пробелы*][*знак*][*цифры*][*.цифры*][{d | D | e | E }[*знак*]*цифры*]

Здесь ***пробелы*** - любой из знаков пробела, табуляции (\t), вертикальной табуляции (\v) - они игнорируются. ***Знак*** - символ '+' или '-'. Если не указан, то считается, что число положительное. ***Цифры*** - символы от '0' до '9'. Для числа с плавающей точкой, если не указаны цифры до знака '.', то должна быть указана хотя бы одна цифра после него. После дробной части может быть указана экспонента, следующая за одним из символов-префиксов экспоненты.

```
char str[]="12345";
int  x,y;
double z;
x = atoi ("12345");
y = atol (str);
z = atof ("12.34");
```

Основной недостаток этих функций заключается в том, что они никак не сигнализируют об ошибке, если таковая произошла в процессе разбора переданной строки - несоответствие его формату или по иным причинам.

Эту проблему решает следующий набор библиотечных функций, также включенных в стандартную библиотеку:

```
long strtol(const char* str, char** end_ptr, int radix)
unsigned long strtoul(const char* str, char** end_ptr,
int radix)
double strtod(const char* str, char** end_ptr)
```

Эти функции имеют следующие отличия от предыдущей группы:

- Через параметр end_ptr они возвращают указатель на первый символ, который не может быть интерпретирован как часть числа.
- Контролируют переполнение и, если таковое произошло, сигнализируют об этом выставлением значения переменной errno в ERANGE, а также возвращают, соответственно, LONG_MAX/LONG_MIN, ULONG_MAX/ULONG_MIN и +/-HUGE_VAL в зависимости от знака числа в переданной строке.
- **strtod** использует информацию о текущих установленных (через setlocale) региональных настройках, таким образом может корректно интерпретировать числа с символом ',' в качестве разделителя целой и дробной части.
- Для функций **strtol** и **strtoul** можно указать основание системы счисления. При этом, если в качестве основания передан 0, то основание определяется автоматически по первым символам числа. Если это символ '0', а сразу за ним идет цифра - то основание принимается равным 8. Если первая цифра '0', а за ней идет символ 'x' или 'X', то основание принимается равным 16. В остальных случаях основание принимается равным 10. В качестве цифр в этом случае можно использовать символы '0' - '9' и 'A' - 'Z' или 'a' - 'z', а основание может принимать значения от 2 до 36.
- Есть варианты этих функций для преобразования чисел, описанных unicode-строками. Они, соответственно, носят названия **wcstol wcstoul** и **wcstod**.

Типичное использование этих функций такое:

```
char* end_ptr;
long val = strtol(str, &end_ptr, 10);
if (*end_ptr)
{
    // Сигнализируем об ошибке в строке
}
```

```
if ((val==LONG_MAX||val==LONG_MIN)&&errno==ERANGE)
{
    // Сигнализируем о переполнении
}
```

Пример:

```
int n; char *p;
    n=strtol("12a",&p,0);
    printf("n=%ld, stop=%c\n",n,*p); //n=12, stop=a
    n=strtol("0x12",&p,0);
    printf("n=%ld, stop=%c\n",n,*p); //n=18, stop=
    n=strtol("01117",&p,2);
    printf("n=%ld, stop=%c\n",n,*p); //n=7, stop=7
    return 1;}
```

*Функции преобразования числовых данных в строку ****

(дополнительный материал)

Нет специализированной функции (описанной в стандарте языка) для обратного преобразования (числа в строку). Но такие функции есть в RTL компиляторах.

Преобразование целого числа в строку

```
char* _itoa(int value, char* string, int radix);
wchar_t* _itow(int value, wchar_t* string, int radix);
char* _ltoa(long value, char* string, int radix);
wchar_t* _ltow(long value, wchar_t* string, int radix);
char* _ultoa(unsigned long value, char* string, int radix);
wchar_t* _ultow(unsigned long value, wchar_t* string, int radix);
```

эти функции входят в библиотеку компиляторов от Microsoft.
Исходные типы представлены следующими аббревиатурами:
i - int;
l - long;
ul - unsigned long;
i64 - int64;
ui64 - unsigned int 64;
формат строки, соответственно:
a - однобайтный char (ANSI);
w - wide-char (Unicode).
Параметры, принимаемые на вход, имеют следующий смысл. Первый параметр (value) - это значение, которое необходимо преобразовать в строку. Второй (string) - буфер, в который будет помещен результат преобразования. А третий (radix) - основание системы счисления, в которой будет представлено число.

. К неспециализированным стандартным функциям, выполняющим необходимое преобразование, можно отнести функцию **sprintf**:

```
int sprintf(char* buffer, const char* format [, argument] ... );
```

выполняющую форматированный вывод в заданный буфер, а также другие функции из этого семейства. Для необходимого нам преобразования необходимо воспользоваться форматной строкой "%d". При этом будет выполнено стандартное преобразование в десятичное представление числа, аналогичное предыдущим функциям. Формат полученного числа определяется тэгом типа. Существуют следующие тэги типов:
d (или **i**) - знаковое целое в десятичном формате;
u - беззнаковое целое в десятичном формате;
o - беззнаковое целое в восьмеричном формате;
x (или **X**) - беззнаковое целое в шестнадцатеричном формате.
Можно использовать также специальные спецификаторы тэгов типа (указываются непосредственно перед тэгом) для того, чтобы указать специальный формат, в котором передано число:
l - передано длинное целое (signed/unsigned long);
h - передано короткое целое (signed/unsigned short);
l64 (для Microsoft) или **L** (для Borland) - передано 64-битное целое (signed/unsigned __int64).
Вообще говоря, при использовании спецификатора типа h необходимо понимать, что при передаче в функцию printf переменных типа short (по стандарту) продвигаются до типа int. Таким образом, этот спецификатор просто дает указание функции форматирование игнорировать старшие разряды полученного целого параметра.
Функции форматного вывода хороши тем, что им можно явно указать формат, в котором мы хотим получить результат. В частности, форматная строка "%08x" преобразует переданное число в строку, содержащую шестнадцатеричное представление числа шириной 8 символов, при этом, при необходимости, строка будет дополнена слева незначащими нулями до 8 символов. Также достоинством этих функций (перед предыдущей группой) является то, что строка преобразуется в соответствии с установленными *для программы* (с помощью вызова функции **setlocale**) региональными настройками (locales).
К основным недостаткам методов форматного вывода можно отнести их низкую скорость работы (на порядок медленней их специализированных аналогов), невозможность преобразования в системы счисления, отличные от восьмеричной, десятичной, и шестнадцатеричной, а также отсутствия контроля за типами и размером переданного буфера, что может приводить к трудновыявляемым ошибкам.

Преобразование числа с плавающей точкой в строку

```
char* _ecvt(double value, int count, int* dec, int* sign);
char* _fcvt(double value, int count, int* dec, int* sign);
char* _gcvt(double value, int digits, char* buffer);
```

- для Microsoft-компиляторов

Назначение функций следующее:
Функция **ecvt** конвертирует число (value) в заданное (count) количество символов, не выполняя при этом никакого форматирования. Через параметр dec возвращается позиция, с

которой начинается дробная часть (позиция десятичной точки), начиная с первого символа строки, а через параметр sign - признак того, что число имеет знак. Тут необходимо сделать ряд следующих замечаний:

1. Если строковое представление числа уже, чем необходимое количество символов, то число справа дополняется нулями.

2. сли строковое представление числа шире, чем необходимое количество символов, то возвращаемая через dec позиция находится правее завершающего строку нуля или левее первого символа строки. Выбор знака dec зависит от знака десятичной экспоненты числа. Например, для value= 3.5678e20 и count=6 функция вернет строку "356780", а dec будет равно 21. А для значения 3.5678e-20 и 6, будет возвращено, соответственно, "356780" и -19.

3. Преобразование производится во внутренний статический буфер, разделяемый с функцией **fcvt**, по этому необходимо быть осторожным при использовании функции в многопоточной среде.

Функция **fcvt** отличается от **ecvt** только тем, что параметр count задает не общее число символов, а число символов после десятичной точки (точность представление числа). При этом строковое представление (по необходимости) дополняется справа нулями до получения нужной ширины. Параметры dec и sign ведут себя аналогичным (как и для **ecvt**) образом. Для приведенных выше примеров возвращенные значения будут "35678000000000000000000000" и 21, а также "" и -19 соответственно. В последнем случае возвращенная пустая строка означает, что строковое представление содержит только нули.

Функция **gcvt** преобразует заданное число (value) в привычное строковое представление и помещает результат в переданный буфер (buffer). При этом в буфер будет помещено не более digits цифр. Если число может быть представлено таким количеством цифр, то оно будет преобразовано в десятичный (фиксированный) формат, иначе будет выполнено преобразование в экспоненциальный формат, и заданное количество цифр будет содержать мантисса числа. Полученное строковое представление может быть дополнено справа нулями для получения необходимого количества разрядов.

При использовании этой функции необходимо помнить следующее: буфер должен быть достаточно большим, чтобы принять digits цифр, завершающий ноль, а также знак, десятичную точку и экспоненту. Никакого контроля не производится, и граница буфера может быть нарушена. При формировании строки региональные настройки учитываются.

Для вывода чисел с плавающей точкой у функций семейства **printf** есть следующие тэги типов:

e (или **E**) (exponent) - преобразует число в экспоненциальное представление;

f (fixed) - преобразует число в десятичное представление основываясь на дополнительных форматных параметрах (ширина и точность);

g (или **G**) (general) - преобразует число либо в экспоненциальное, либо в десятичное представление в зависимости от того, какое представление будет короче.

Так же, как и для целых чисел, можно указать дополнительные спецификаторы:

L - передано число в формате long double.

Необходимо заметить, что при передачи переменных типа float производится их продвижение до типа double.

Дополнительно можно указать общее количество значащих цифр в строковом представлении (ширну), а также количество цифр в дробной части (точность) в следующем виде:

%<*ширина*>.<*точность*><*тэг_типа*>.
Подробнее об этих форматах можно прочитать в документации по форматной строке функций этого семейства, т. к. подробное описание всех возможностей функций семейства printf выходит за рамки этой статьи.
В C++ числа можно конвертировать в символьное представление с помощью операторов потокового вывода (<<). При этом для выводимых чисел можно установить желаемую точность (метод precision) и ширину (метод width). Желаемый формат устанавливается путем вывода в поток соответствующего форматного флага:
dec - целое в десятичном формате;
hex - целое в шестнадцатеричном формате;
oct - целое в восьмеричном формате;
scientific - число с плавающей точкой в экспоненциальном формате;
fixed - число с плавающей точкой в фиксированном формате.
При этом форматы для целых чисел и чисел с плавающей точкой устанавливаются независимо.
При выводе региональные настройки также учитываются, но устанавливать их нужно особым образом, и обсуждение этого вопроса выходит за рамки этой статьи.
Если говорить о возможностях библиотек MFC и VCL, то в MFC единственный способ преобразовать число в строку - это метод **Format** класса CString. Этот метод работает аналогично методу **sprintf**.
У VCL в этом плане возможностей больше. У класса AnsiString существуют следующие статические методы:
IntToHex - преобразует число в шестнадцатеричное представление.
FormatFloat - преобразует число с плавающей точкой в строку в соответствии с заданной (в виде строки) маской, что позволяет получать числа практически в любом желаемом формате.
FloatToStrF - преобразует число в строку в соответствии с несколькими самыми распространенными форматами (sffGeneral, sffExponent, sffFixed), которые соответствуют форматам функции sprintf (g, e, f). Также этот метод может добавить в строку разделители тысяч (формат sffNumber), или преобразовать в представление денежных единиц (формат sffCurrency). При этом все разделительные символы, формат числа и т. п. будут взяты из *текущих региональных настроек системы*.
Также существуют (но не отражены в документации) нестатические методы класса AnsiString **sprintf** и **printf**, аналогичные по возможностям соответствующим RTL-ным, но не принимающие на вход буфер, в качестве которого используется экземпляр класса AnsiString.
В WinAPI (симметрично методу **GetDlgItemInt**) существует метод **SetDlgItemInt**, выполняющий необходимые преобразования числа в строку:

```
UINT SetDlgItemInt(HWND hDlg, int itemId, UINT value, BOOL signed)
UNIT CWnd::GetDlgItemInt(int itemId, UINT value, BOOL signed)
```

Последним параметров в эти функции передается признак знаковости числа.

упражнение для подготовки к КР (выполнить без компьютера)

```
int *x;
x=(int*)malloc(12);
*x=1;
x++;
*x=2;
x++;
*x=3;
printf("%d %d  %d \n",x[-2],x[-1],x[0]);
printf("%d %d  %d \n",x[0],x[1],x[2]);
printf("%d %d  %d \n",*(x-2),*(x-1),*x);
printf("%d %d  %d \n",*x--,*x--,*x--);
printf("%d %d  %d \n",x[0],x[1],x[2]);
```