Строки

М. А. Ложников

13 февраля 2020 г.
Ревизия: 1

Это предварительная невыверенная версия. Читайте на свой страх и риск.

1 Символы в языке C

В языке C помимо типов данных для целых чисел и чисел с плавающей точкой есть тип данных для хранения символов. Этот тип данных называется `char`. Не смотря на то, что этот тип данных предназначен для хранения отдельных символов, он является обычным целочисленным типом данных, то есть ему можно присваивать целые числа, конвертировать его в целые числа, а также выполнять с переменными этого типа арифметические операции.

Тип данных `char` занимает в памяти 1 байт [2] (то есть `sizeof(char) = 1`) и обычно (но не всегда) принимает целые значения от -128 до 127. Дело в том, что на некоторых архитектурах размер одного байта может отличаться от 8 бит. В языке C количество бит в байте хранится в специальной константе `CHAR_BIT` [3], определённой в заголовочном файле `<limits.h>`.

Таким образом, для хранения символов используются обычные целые числа. Соответствие между числами и символами определяется кодировкой. Как правило числа, которые соответствуют латинским символам, знакам препинания, а также некоторым служебным символам (например, перевод строки) не зависят от кодировки. Это выполняется для символов ASCII. Однако, символы других алфавитов (не ASCII) в различных кодировках могут иметь разные значения или вообще отсутствовать.

2 Символьные константы

Для инициализации символьных типов данных можно использовать специальные символьные константы. Символьная константа записывается в одинарных кавычках. Это может быть либо символ алфавита, либо знак препинания, либо служебный символ. Например,

```
1 char sym = 'a'; // Инициализируем переменную sym строчной латинской буквой a.
2
3 sym = 'D'; // Заглавная латинская буква D.
4 sym = ' '; // Пробел.
```

Может показаться, что компилятор языка C конвертирует символьную константу в числовое значение, которое ей соответствует. На самом деле, всё немного не так. В действительности имеет место обратный процесс. Компилятор всего-навсего использует то значение, при помощи которого символьная константа была представлена в исходном коде программы. То, как эта константа выглядит при редактировании исходного кода в текстовом редакторе определяется кодировкой, выбранной

в текстовом редакторе. А вид этой символьной константы в выводе программы на экран определяется кодировкой, выбранной в терминале.

Таким образом, при использовании символов, отличных от ASCII кодировка файлов исходного кода программы должна совпадать с кодировкой терминала, в котором эта программа запускается.

Помимо символов алфавита `'a'`, `'b'`, `'c'`, ..., `'z'` и `'A'`, `'B'`, `'C'`, ..., `'Z'` доступны знаки препинания `','`, `'.'`, `':'`, `';'`, `'!'`, `'?'`, `'%'`, `'$'`, `'#'` и т. д., а также различные специальные символы, среди которых важно упомянуть следующие:

- `'\n'` — символ перевода строки;
- `'\t'` — символ табуляции;
- `'\r'` — символ возврата каретки;
- `'\"'` — символ двойной кавычки;
- `'\''` — символ одинарной кавычки;
- `'\\'` — символ \;
- `'\0'` — символ конца строки, значение которого в любой кодировке равно 0.

Таким образом, специальные символы записываются при помощи символа экранирования \. Отметим, что строковый литерал служит для представления ровно одного символа.

Замечание 2.1 *Не смотря на то, что для представления символов в языке C есть специальный тип* `char`*, типом данных символьной константы в языке C является тип данных* `int`*.*

3 Ввод/вывод символов

3.1 Первый способ

Для ввода/вывода отдельных символов можно использовать функции `scanf()`/`printf()`, указав в формате `"%c"`.

```
1  char sym;
2
3  /* Пример считывания символа с клавиатуры. */
4  if (scanf("%c", &sym) != 1) {
5    printf("Can't read symbol!\n");
6    return 0;
7  }
8
9  /* Пример вывода символа на экран. */
10 if (sym == 'A')
11   printf("sym = A\n");
12 else
13   printf("sym = %c\n", sym);
```

Отметим, что символы считываются последовательно, разделители считываются точно так же, как и все остальные символы и записываются в выходные переменные. Иными словами, никакие символы не игнорируются.

Аналогично, можно использовать функции `fscanf()`/`fprintf()` при работе с файлами.

3.2 Второй способ

Функции `scanf()`/`printf()` слишком "тяжёлые" для считывания/вывода одного символа. Есть функции полегче, предназначенные только для этого, а именно `getchar()`/`putchar()` и `fgetc()`/`fputc()`.

```
int sym;

/* Пример считывания символа с клавиатуры. В случае ошибки или конца файла
   возвращается константа EOF, определённая в заголовочном файле <stdio.h>. */
if ((sym = getchar()) == EOF) {
  /* Функция perror() выводит сообщение в поток вывода ошибок stderr. */
  perror("Can't read symbol!\n");
  return 0;
}

printf("Symbol: ");

/* Вывод символа на экран. Функция возвращает EOF в случае ошибки. */
if (putchar(sym) == EOF) {
  perror("Can't write the symbol!\n");
}

putchar('\n');
```

То же самое, только с файлами:

```
FILE* fin;
FILE* fout;
int sym;

if ((fin = fopen("input.txt", "rt")) == NULL) {
  perror("Can't open input file!\n");
  return 0;
}

/* Пример считывания символа из файла. В случае ошибки возвращается константа EOF. */
if ((sym = fgetc(fin)) == EOF) {
  perror("Can't read symbol!\n");
  fclose(fin);
  return 0;
}

if ((fout = fopen("output.txt", "wt")) == NULL) {
  perror("Can't open output file!\n");
  fclose(fin);
  return 0;
}

/* Пример записи символа в файл. */
if (fputc(sym, fout) == EOF) {
```

```
25   perror("Can't write the symbol!\n");
26 }
27
28 putchar('\n');
29
30 fclose(fin);
31 fclose(fout);
```

Кроме того, есть функция `int ungetc(int ch, FILE* stream)`, которая кладёт символ `ch` обратно в файл/поток `stream` и возвращает сам символ в случае успеха и `EOF` в случае ошибки. Эта функция может быть полезна для того, чтобы положить обратно в поток по ошибке считанный символ для того, чтобы считать его в дальнейшем из этого потока. Гарантируется, что функция отработает без ошибки хотя бы один раз, однако несколько последовательных вызовов `ungetc()`, не прерываемых операциями чтения из потока, могут не сработать.

Замечание 3.1 *При использовании функций `getchar()` и `fgetc()` важно сохранять значение в тип данных `int`, а не `char`, поскольку множество возвращаемых значений функции больше множества возможных значений типа `char`. Функция возвращает значение типа `unsigned char` (беззнаковый `char`), приведённое к типу `int` или константу `EOF`, значение которой равно -1. Значения типов `char` и `unsigned char` переводятся друг в друга без потерь, однако константа `EOF` может совпасть со значением считанного символа при конвертации возвращаемого значения в тип `char`/`unsigned char`.*

Замечание 3.2 *В заголовочном файле `<stdio.h>` определены стандартные потоки ввода/вывода и поток вывода ошибок, а именно `stdin`, `stdout` и `stderr` соответственно, которые имеют тип данных `FILE*`. Таким образом, вызов `fgetc(stdin)` эквивалентен вызову `getchar()`, а вызов `fputc(sym, stdout)` эквивалентен вызову `putchar(sym)`.*

4 Относительный порядок символов

Во всех кодировках символы алфавита расположены последовательно в алфавитном порядке, то есть им соответствуют последовательные целые числа. Символы, обозначающие цифры также расположены последовательно от `'0'` до `'9'`. Кроме того, символы пунктуации тоже расположены последовательно.

```
1 int i;
2
3 /* Выводим строчные латинские символы от 'a' до 'j'. */
4 for (i = 0; i < 10; i++)
5   putchar('a' + i);
6
7 /* Выводим заглавные латинские символы от 'D' до 'H'. */
8 for (i = 0; i < 5; i++)
9   putchar('D' + i);
```

Тот факт, что символы алфавита расположены по порядку, можно использовать для проверки того, что символ является символом алфавита, а также, например, для конвертации заглавных символов в строчные.

```c
/* Функция проверяет, является ли символ цифрой. */
int IsDigit(int sym) {
  if (sym >= '0' && sym <= '9')
    return 1;

  return 0;
}

/* Функция проверяет, является ли символ символом латинского алфавита. */
int IsAlpha(int sym) {
  if ((sym >= 'a' && sym <= 'z') || (sym >= 'A' && sym <= 'Z'))
    return 1;

  return 0;
}

/* Функция конвертирует заглавные латинские буквы в строчные, а остальные оставляет
   без изменения. */
int ToLower(int sym) {
  if (sym >= 'A' && sym <= 'Z')
    return sym + 'a' - 'A';

  return sym;
}
```

В заголовочном файле `<ctype.h>` определены функции для работы с отдельными символами. Например, функции `int isalnum(int c)`, `int isalpha(int c)`, `int isdigit(int c)`, `int ispunct(int c)` и `int isspace(int c)` возвращают число, отличное от нуля, если символ `c` является алфавитным или цифрой; алфавитным; цифрой; символом пунктуации или пробельным символом соответственно.

Замечание 4.1 *Функции стандартной библиотеки умеют работать не только с кодировкой ASCII. См. функцию `setlocale()` [4].*

5 Строки и строковые литералы в языке C

Строки в языке C представляются в виде массивов элементов типа `char`, причём каждая строка должна оканчиваться символом конца строки `'\0'`. Таким образом, строка, состоящая из N символов должна храниться в массиве длины не менее $N + 1$. Длину строки хранить необязательно.

Строковые литералы (константы) в языке C записываются внутри двойных кавычек и имеют тип `const char*` — массив указателей на `const char`, ключевое слово `const` означает, что элементы этого массива нельзя изменять. Каждый строковый литерал неявно заканчивается символом конца строки. Статический массив элементов типа `char` можно инициализировать при объявлении строковым литералом.

```c
/* Инициализируем при объявлении статический массив str строковым литералом. Каждый
   строковый литерал заканчивается символом конца строки '\0', и в массив str
   он тоже пропишется. */
char str[256] = "Hello world!";

```

```
/* А вот следующий код уже вызовет ошибку компиляции. Нельзя преобразовать указатель
   в статический массив, то есть тип const char* в тип char[]. */
/* str = "Something else."; */

/* Объявляем указатель на const char. */
const char* ptr;

/* Здесь всё в порядке, ошибки нет. */
ptr = "Some string.";
```

Замечание 5.1 *При работе со строками в языке C важно отличать символ конца строки `'\0'` от символа перевода строки `'\n'`. Символ конца строки предназначен для того, чтобы функции работы со строками могли определить количество символов, которое нужно обработать. Символ перевода строки влияет на способ вывода строк, например, функции вывода строки на экран выводят символы с новой строки как только встретят символ перевода строки. Таким образом, одна строка в языке C может содержать несколько символов перевода строки. Отметим, что в английском языке строка в языке C и строка в текстовом редакторе или в выводе на экране называются разными словами: string и line соответственно.*

Замечание 5.2 *Обычно текстовые редакторы в Windows вставляют два символа `"\r\n"` для обозначения перевода строки, в то время как текстовые редакторы в Linux обычно вставляют только символ `'\n'`.*

5.1 Пример перевода строки в число

```
/* Функция для перевода строки в целое число. */
int AToI(const char* str) {
  int number = 0;
  int sign = 1;

  /* Если строка пустая. */
  if (*str == 0)
    return 0;

  /* Если первый символ --- знак "минус". */
  if (*str == '-') {
    sign = -1;

    /* Перемещаем указатель на следующий символ. */
    str++;
  }

  /* Цикл до тех пор пока указатель указывает на символ, обозначающий цифру. */
  for (; IsDigit(*str); str++) {
    number *= 10;

    /* Символы, обозначающие цифры, расположены по порядку. Значение выражения
       *str - '0' равно текущей цифре. */
    number += *str - '0';
  }
```

```
26
27   return number * sign;
28 }
```

В заголовочном файле <stdlib.h> объявлена функция int atoi(const char* str), которая переводит строку в целое число, а также функция double atof(const char* str), которая переводит строку в число с плавающей точкой.

6 Ввод/вывод строк

Как и в случае с отдельными символами строки можно считывать и выводить различными способами.

6.1 Первый способ

Для ввода/вывода строк можно использовать функции scanf()/printf() с форматом "%s", а также их аналоги fscanf()/fprintf() для чтения/записи из файла/в файл. Стоит отметить, что функции scanf()/fscanf() считывают строку до первого разделителя.

```
1 char buffer[256]; /* Заведём массив достаточного размера. */
2
3 /* В функции scanf() перед buffer не нужно ставить &, поскольку массивы передаются
4    в функции по указателям. */
5 if (scanf("%s", buffer) != 1) {
6   printf("Can't read string!\n");
7   return 0;
8 }
9
10 printf("String: \"%s\"\n", buffer);
```

Замечание 6.1 *Отметим, что в printf() не следует передавать строку напрямую вместо формата, поскольку строка может содержать символы форматирования %.*

6.2 Второй способ

Функция char* fgets(char* str, int count, FILE* stream) считывает из потока/файла stream символы в массив str до символа перевода строки '\n', но не более count - 1 символов. Если функция fgets() дошла до символа перевода строки, то она запишет его в массив. Кроме того, она всегда записывает символ конца строки '\0'. Функция возвращает указатель str в случае успеха и NULL в случае ошибки. В массиве str должна быть выделена память под не менее чем count элементов типа char.

Функция int fputs(const char* str, FILE* stream) выводит строку str в файл/поток stream. В случае успеха она возвращает неотрицательное значение, а в случае ошибки константу EOF.

```
1 char buffer[256];
2
3 /* Считываем строки с клавиатуры до тех пор, пока считываются. */
4 while (fgets(buffer, 256, stdin) != NULL) {
5   printf("New line: %s", buffer);
6 }
```

6.3 Считывание чисел из строк и форматирование строк

Точно так же как и в случае с форматированным вводом/выводом в файл/из файла в стандартной библиотеке языка C существуют функции для форматирования строк, а также для считывания форматированных данных из строки. Функции называются `sscanf()`/`sprintf()`. Они работают точно так же, как и `fscanf()`/`fprintf()`, но первым параметром у них является строка.

```
char buffer[256] = "3.14 2.71";
double a, b;

if (sscanf(buffer, "%lf%lf", &a, &b) != 2) {
  /* Этого не должно произойти. */
  printf("Can't read numbers!\n");
  return 0;
}

/* Функция sprintf() затрёт данные, которые были в строке buffer до вызова функции. */
sprintf(buffer, "Sum is equal to %f", a + b);

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

6.4 Считывание чисел из строк

Следующие функции, определённые в `<stdlib.h>`, позволяют считать число из строки, а также узнать, на какой позиции считывание остановилось.

```
char firstString[256] = "12345abcde";
long firstValue;
double secondValue;
char* pos;

/* Первый параметр --- строка, откуда считываются данные, второй параметр ---
   указатель на указатель на первый символ, который не будет считан, то есть первый
   символ, не входящий в число. Третий параметр --- основание системы исчисления,
   в которой записано число. Если вторым аргументом передать NULL, то аргумент будет
   проигнорирован функцией. */
firstValue = strtol(firstString, &pos, 10);

/* Выведет Value: 12345, position: abcde */
printf("Value: %ld, position: %s\n", firstValue, pos);

secondValue = strtod("1.2345e-3abcd", &pos);

/* Выведет Value: 0.001234, position: abcd
   Числа с плавающей точкой по-умолчанию выводятся в фиксированном формате
   с точностью 6 знаков после запятой. */
printf("Value: %f, position: %s\n", secondValue, pos);
```

7 Другие функции работы со строками

Все функции данного раздела объявлены в заголовочном файле <string.h>.

7.1 Реализация некоторых функций

Рассмотрим реализации некоторых функций, аналогичные которым есть в стандартной библиотеке языка C.

```c
/* Функция возвращает длину строки. */
size_t StrLen(const char* str) {
  size_t length = 0;

  for (; *str != 0; str++)
    length++;

  return length;
}

/* Копирует строку src на место строки dest. Данные, которые были в строке dest
   затираются. */
char* StrCpy(char* dest, const char* src) {
  char* ptrToDest = dest;

  for (; *src != 0; src++) {
    *dest = *src;
    dest++;
  }

  /* Прописываем символ конца строки. */
  *dest = 0;

  return ptrToDest;
}

/* Добавляет строку src в конец строки dest. */
char* StrCat(char* dest, const char* src) {
  char* ptrToDest = dest;

  /* Пропускаем символы, которые записаны в dest. У этого цикла нет тела, поскольку
     после закрывающейся круглой скобки стоит точка с запятой. */
  for (; *dest != 0; dest++);

  /* Добавляем в конец dest символы из src. */
  for (; *src != 0; src++) {
    *dest = *src;
    dest++;
  }

  /* Прописываем символ конца строки. */
```

```
42   *dest = 0;
43
44   return ptrToDest;
45 }
```

В стандартной библиотеке языка C есть функции, аналогичные рассмотренным — `strlen()`, `strcpy()`, `strcat()`, а также функция `strcmp()`, предназначенная для сравнения двух строк в лексикографическом порядке.

Замечание 7.1 *Отметим, что функции стандартной библиотеки* ***`strlen()`****,* ***`strcpy()`****,* ***`strcat()`****,* ***`strcmp()`*** *могут привести к неопределённому поведению в случае, если в их аргументах не прописан символ конца строки. Для того, чтобы предостеречь себя от подобных ошибок рекомендуется использовать функции* ***`strnlen()`****,* ***`strncpy()`****,* ***`strncat()`****,* ***`strncmp()`****, которые принимают ещё один дополнительный параметр, ограничивающий сверху число обрабатываемых символов.*

7.2 Поиск подстроки в строке

Рассмотрим пример использования некоторых других функций.

```
1  const char* str = "Hello world!";
2  char* ptr;
3
4  /* Возвращает указатель на первое вхождение подстроки (второй параметр)
5     в строку (первый параметр) или NULL, если подстрока в строке не найдена. */
6  ptr = strstr(str, "wor");
7
8  if (ptr != NULL) {
9    printf("Found substring: %s\n", ptr);
10 }
```

7.3 Сортировка строк по возрастанию

Рассмотрим пример простейшей сортировки строк методом “пузырька”.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  #define MAX_LINES 10
6  #define MAX_LINE_LENGTH 256
7
8  /* Функция для освобождения памяти. Она принимает массив указателей и освобождает
9     память по каждому указателю. */
10 void FreeMemory(char* lines[], int numLines) {
11   int i;
12
13   for (i = 0; i < numLines; i++)
14     free(lines[i]);
15 }
```

```
16
17 int main(void) {
18   /* Статический массив указателей (MAX_LINES элементов, тип элемента --- char*).
19      Инициализируем все элементы нулём. */
20   char* lines[MAX_LINES] = { 0 };
21   char buffer[MAX_LINE_LENGTH];
22   int numLines, i, k;
23
24   for (numLines = 0; numLines < MAX_LINES; numLines++) {
25     /* Считываем новую строчку в массив buffer. */
26     if (fgets(buffer, MAX_LINE_LENGTH, stdin) == NULL)
27       break;
28
29     /* Выделяем память под новую строку. */
30     lines[numLines] = (char*) malloc(sizeof(char) * (strlen(buffer) + 1));
31
32     if (lines[numLines] == NULL) {
33       /* Не удалось выделить память. */
34       printf("Can't allocate memory!\n");
35       FreeMemory(lines, numLines);
36       return 0;
37     }
38
39     /* Копируем строку в массив lines. */
40     strcpy(lines[numLines], buffer);
41   }
42
43   /* Сортировка по возрастанию в лексикографическом порядке пузырьком. */
44   for (i = 0; i < numLines - 1; i++) {
45     for (k = 0; k < numLines - i - 1; k++) {
46       /* Функция strcmp() возвращает положительное число, если первая строка
47          больше второй в лексикографическом порядке; отрицательное число, если первая
48          строка меньше второй в лексикографическом порядке и 0, если строки равны. */
49       if (strcmp(lines[k], lines[k + 1]) > 0) {
50         char* tmp;
51
52         /* В случае неправильного прядка просто переставляем местами указатели.
53            Никакого копирования не происходит. */
54         tmp = lines[k];
55         lines[k] = lines[k + 1];
56         lines[k + 1] = tmp;
57       }
58     }
59   }
60
61   /* Выводим результат на экран. */
62   printf("Result:\n");
63   for (i = 0; i < numLines; i++)
64     printf("%d. %s", i + 1, lines[i]);
65
66   /* Освобождаем память. */
```

```
67   FreeMemory(lines, numLines);
68
69   return 0;
70 }
```

7.4 Разбиение строки, используя набор разделителей

Следующий пример демонстрирует разбиение строки при помощи набора разделителей.

```
1  /* Следующие друг за другом строковые литералы объединяются в одну строку. */
2  char buffer[256] = "There were four of us --- George, and William Samuel Harris, "
3                     "and myself, and Montmorency.";
4  const char* delimiters = "\r\n\t,.!?- ";
5  char* token;
6
7  token = strtok(buffer, delimiters);
8
9  while (token != NULL) {
10   printf("Token: %s\n", token);
11
12   /* В последующие разы строку передавать не нужно, функция strtok() её запомнит. */
13   token = strtok(NULL, delimiters);
14 }
```

Замечание 7.2 *Функция* `strtok()` *не выделяет памяти. В переменную* `token` *записывается указатель на часть массива* `buffer`*. Таким образом, функция меняет строку, которая в неё передаётся, вставляя на место разделителей символы конца строки.*

8 Несколько слов о Unicode

В этом разделе приведена лишь минимальная информация для ознакомления.

При работе с символами, отличными от ASCII, приходится уделять внимание выбору кодировки, поскольку тип `char` не может вместить в себя все возможные символы всех алфавитов. Это создаёт некоторые трудности, особенно в программах, которые должны работать с несколькими различными языками, поскольку каждый язык требует своей кодировки.

Для того, чтобы избежать лишних проблем с кодировками был введён стандарт Unicode. Однако и этот стандарт имеет несколько различных вариантов, например, UTF-16, в котором каждый символ кодируется 16 битами или UTF-32, в котором каждый символ кодируется 32 битами.

В языке C есть специальный тип данных `wchar_t`, предназначенный для хранения символов Unicode. Однако и здесь отсутствует какой-либо стандарт, на системах, поддерживающих UTF-32, размер `wchar_t` равен 4 байтам (например, в Linux), исключением является ОС Windows, в которой размер `wchar_t` равен 2 байтам. В стандартной библиотеке языка C есть специальные функции, работающие со строками из символов `wchar_t`, а также функции для того, чтобы преобразовывать обычные строки в “широкие” строки (массивы `wchar_t`) и обратно, используя различные кодировки.

В настоящее время часто используется кодировка UTF-8, в которой символы кодируются переменным числом байт. Например, ASCII символы в этой кодировке кодируются 1 байтом, а символы русского алфавита — двумя байтами. Во многих дистрибутивах Linux эта кодировка выбрана по-умолчанию, например, в файловой системе или в текстовых редакторах.

Список литературы

[1] *Брайан Керниган, Деннис Ритчи* Язык программирования Си.

[2] `https://en.cppreference.com/w/c/language/sizeof`

[3] `https://en.cppreference.com/w/c/types/limits`

[4] `https://en.cppreference.com/w/c/locale/setlocale`