Строки

Представление строк (1)

- Строка – это последовательность символов, заканчивающаяся и включающая первый нулевой символ (англ., null character ‘\0’ (символ с кодом 0)).

- Преимущества подхода
  - Простота.
- Недостатки подхода
  - Отсутствие быстрого способа определения длины строки.
  - Тщательность при работе с нулевым символом.

Представление строк (2)

Определение переменной-строки, которая может содержать до 80 символов обычно выглядит следующим образом:

```
#define STR_LEN 80
...
char str[STR_LEN+1];   // !
```

Поскольку строка – массив символов, для доступа к элементу строки может использоваться операция индексации:

```
int count_spaces(const char *s)
{
    int count = 0;
    for (int i = 0; s[i] != '\0'; i++)
        if (s[i] == ' ')
            count++;
    return count;
}
```

Строковый литерал (1)

- Строковый литерал – последовательность символов, заключенных в двойные кавычки.

```
char str[] = “String for test”;

printf(“Max is %d\n”, max);
```

- Строковый литерал рассматривается компилятором как массив элементов типа char. Когда компилятор встречает строковый литерал из n символов, он выделяет n+1 байт памяти, которые заполняет символами строкового литерала и завершает нулевым символом.

Строковый литерал (2)

- Массив, который содержит строковый литерал, существует в течение всего времени выполнения программы.
- В стандарте сказано, что поведение программы не определено при попытке изменить строковый литерал.
- Обычно строковые литералы хранятся в read only секции.

Строковый литерал (3)

```
char *p = "abc", ch;

printf("addr(p) %p, addr(\"abc\") %p\n", p, "abc");
// addr(p) 0040a064, addr("abc") 0040a064

ch = "abc"[1];
printf("ch = %c\n", ch);
// ch = b

// Ошибка времени выполнения
*p = 'j';
// т.е. правильно было бы описать p как "const char *p"
```

Строковый литерал (4)

Если строковый литерал слишком длинный, Си позволяет продолжить его на следующей строке.

```
   printf(“My name is \
Igor”);  // продолжение должно начинаться с начала следующей строки
```

Существует более удобный вариант, благодаря следующему правилу: «когда два или более строковых литерала расположены рядом, компилятор объединяет их в одну строку».

```
    printf(“My name is ”
                    “Igor”);
```

Инициализация строковых переменных

- char str_1[] = {‘J’,‘u’,‘n’, ‘e’, ‘\0’};
- char str_2[] = “June”;
- char str_3[5] = “June”;
- char str_4[3] = “June”;

```
// error: initializer-string for array of chars is too long
```

- char str_5[4] = “June”;

```
// str_5 не строка!
```

Массив символов и указатель на строковый литерал

```
// массив символов
char str_arr[] = "June";
// указатель на строковый литерал
char *str_ptr = "June";

void process(const char *str);

str_arr[0] = 'j';      // ок
str_ptr[0] = 'j';      // ошибка времени выполнения
```

Массив строк

Способ 1: двумерный массив строк

```
char arr_1[][9] = {"January", "February", "March"};
```

J	a	n	u	a	r	y	\0	\0
F	e	b	r	u	a	r	y	\0
M	a	r	c	h	\0	\0	\0	\0

Способ 2: массив указателей на строки (“ragged array”)

```
/*const*/ char *arr_2[] = {"January", "February", "March"};
```

[0]	==>	J	a	n	u	a	r	y	\0	
[1]	==>	F	e	b	r	u	a	r	y	\0
[2]	==>	M	a	r	c	h	\0			

Вывод строк

```
#include <stdio.h>
...
char str[] = "Hello, world!";

printf("%s\n", str);
puts(str);
```

Ввод строк

```
#include <stdio.h>
...
char str[10];

scanf("%s", str);
// Через scanf нельзя ввести строку с пробелами!
gets(str);
```

«Правильный» ввод строки (1): собственная реализация

Функции scanf и gets небезопасны и недостаточно гибки. Программисты часто реализуют свою собственную функцию для ввода строки, в основе которой лежит посимвольное чтение вводимой строки с помощью функции getchar.

```
#include <stdio.h>
int getchar(void);
```

О чем следует задуматься:

- Должна ли функция пропускать «разделители»?
- Какой символ должен приводить к окончанию ввода? Этот символ должен включаться в строку?
- Что делать если строка слишком длинная?

«Правильный» ввод строки (2): собственная реализация

```
int read_line(char *s, int n)
{
    int ch, i = 0;
    while ((ch = getchar()) != '\n' && ch != EOF)
        if (i < n - 1)
            s[i++] = ch;
    s[i] = '\0';
    return i;
}
```

- Параметры функции: s – массив, в котором сохраняются символы, n – размер этого массива.
- Функция возвращает количество символов, сохраненных в массиве.
- Символы, которые не помещаются в массив, игнорируются.

«Правильный» ввод строк (3): стандартная библиотека

char *fgets(char *s, int size, FILE *stream);

Прекращает ввод когда (любое из)

- прочитан символ ‘\n’;
- достигнут конец файл;
- прочитано size-1 символов.

Введенная строка всегда заканчивается нулем.

fgets(str, sizeof(str), stdin);

Обработка строк

- Нет специальных операций для работы со строками.

```
char str_1[] = “June”;
char str_2[] = “July”;
char str_3[10];

str_3 = str_1;    // ошибка компиляции

if (str_1 < str_2)
// формально ошибки нет, но что будет сравниваться?
```

- Есть функции стандартной библиотеки для работы со строками.

```
string.h
```

Обработка строк

char* strcpy(char *s1, const char *s2);

```
char src[] = "Hello!";
char dst[20];

strcpy(dst, src);
```

Вместо функции strcpy безопаснее использовать функцию strncpy.

char* strncpy(char *s1, const char *s2,
size_t count);

```
strncpy(dst, src, sizeof(dst) - 1);
dst[sizeof(dst) - 1] = '\0';
```

Обработка строк

size_t strlen(const char *s);

```
char dst[20];
size_t len;

strcpy(dst, "Hello!");
len = strlen(dst);       // len = 6, а не 20
```

Обработка строк

int strcmp(const char *s1, const char *s2);

значение < 0, если s1 меньше s2

0, если s1 равна s2

значение > 0, если s1 больше s2

Строки сравниваются в лексикографическом порядке (как в словаре).

int strncmp(const char *s1, const char *s2

size_t count);

Лексикографический порядок

Строка s1 меньше строки s2, если выполнено любое из двух условий:

- первые i символов строк s1 и s2 одинаковы, а символ s1[i+1] меньше символа s2[i+1] (пример, “abc” < “abd” или “abc” < “bcd”);
- все символы строк s1 и s2 одинаковы, но строка s1 короче строки s2 (пример, “abc” < “abcd”).

Функция strcmp сравнивает символы, сравнивая значения кодов, которые представляют эти символы.

Почему strcmp возвращает «нечеткие» значения

Исторические особенности реализации.

В одном из первых изданий Kernighan, Ritchie можно найти:

```
int strcmp(char *s, char *t)
{
    int i;

    for (i = 0; s[i] == t[i]; i++)
        if (s[i] == '\0')
            return 0;

    return s[i] - t[i];
}
```

Обработка строк

char* strdup(const char *s); // НЕ c99
char* strndup(const char *s, size_t count); // НЕ c99

```
char* str;

str = strdup("Hello!");
if (str)
{
    ...
    free(str);
}
```

int sprintf(char *s, const char *format, ...);
// c99
int snprintf(char *s, size_t num, const char *format, ...);

Обработка строк

char* strtok(char *string, const char *delim);

```
char str_test_1[] = "   This   is a,,, test string!!!";

char *pword = strtok(str_test_1, "\n ,.!?");

while (pword)
{
    printf("[%s]\n", pword);

    pword = strtok(NULL, "\n ,.!?");
}
```

Обработка строк

Перевод числа в строку

```
#include <stdlib.h>

// Семейство функций (atoi, atof, atoll)
long int atol(const char* str);

// Семейство функций (strtoul, strtoll, ...)
long int strtol(const char* string, char** endptr, int basis);
```

Примеры

```
size_t string_len(const char *str)
{
    size_t n;

    for (n = 0; str[n] != '\0'; n++)
        ;

    return n;
}

size_t string_len(const char *str)
{
    const char *beg = str;

    while (*str)
        str++;

    return str - beg;
}
```

Примеры

```
char*  string_cat(char *s1, const char *s2)
{
    char *cur = s1;

    while (*cur)
        cur++;

    // без скобок здесь warning, который превращается в ошибку
    while ((*cur++ = *s2++))
        ;

    return s1;
}
```

Примеры

Подсчёт слов в строке

- Начало слова: это когда разделитель сменяется «не-разделителем».
- Частный случай: «не-разделитель» идёт первым символом.

```
int n_words(char *s)
{
    int k = 0, n = strlen(s);
    for (int i = 0; i < n; i++)
        if (!strchr("\n ,.!?", s[i]))
        {// Текущий символ – не-разделитель!
            if (i == 0 || strchr("\n ,.!?", s[i - 1]))
            {// А предыдущий - разделитель.
                k++;
            }
        }
    return k;
}
```

Примеры

Разделяем строку на слова

- Все знаки пунктуации заменяются нулями (признаком конца строки).
- Адреса начала слов записываются в массив w.
- Возвращается число слов.

```
int split(char *s, char *w[])
{
    int k = 0, n = strlen(s);
    for (int i = 0; i < n; i++)
        if (strchr("\n ,.!?", s[i])) // Текущий символ — разделитель
            // Заменяем его на признак конца строки.
            s[i] = 0;
        else // Текущий символ — не разделитель.
            if (i == 0 || s[i - 1] == 0) // А предыдущий — разделитель
                // В массив слов записываем адрес текущего символа.
                w[k++] = s + i;
    return k;
}
```

Примеры

Печать всех слов в строке

- Для выделения памяти под массив слов (указателей на строки) предварительно вызываем n_word.
- Некультурное альтернативное решение: объявить w как char *w[100].

```
void print_words(char *s)
{
    int k = n_words(s);
    if (k == 0)
        return; // Нет слов — ничего не делаем
    // Объём памяти — число элементов (k) на размер одного элемента
    // (sizeof(w[0]))
    char **w = malloc(k * sizeof(w[0]));
    split(s, w);
    for (int i = 0; i < k; i++)
        printf("[%s]\n", w[i]);
    free(w);
}
```