1 Занятие №3

1.1 Литеральные значения

Рассмотрим правила записи констант (литеральных значений) в языке **Си**.

1.1.1 Целые

Целые значения могут записываться в программе в десятичной, восьмеричной и шестнадцатеричной системе счисления. *Восьмеричная константа* начинается с символа `0` («ноль»), за которым идут цифры `0—7`, например `0377`. *Шестнадцатеричная константа* начинается с символов `0x`, затем идут цифры `0—9`, `a—f`, `A—F`. Пример `0xFF`. *Десятичная константа* начинается с цифр `1—9`, далее идут цифры `0—9`.

Язык **Си** не позволяет записывать целые числа в двоичной системе. Обратите внимание, что ведущий ноль является признаком *восьмеричного* числа. Запись `09` неверна и вызовет ошибку компиляции.

Тип константы — это минимальный тип, который может содержать данное значение. Для констант, заданных в десятичной форме, последовательно выбираются **`int`**, **`long`**, **`long long`** (для **C99**). Для констант, заданных в восьмеричной или шестнадцатеричной форме, последовательно выбираются **`int`**, **`unsigned int`**, **`long`**, **`unsigned long`**, **`long long`**, **`unsigned long long`** (для **C99**).

Тип константы можно задать явно указанием суффикса `u` или `l`. Эти суффиксы могут употребляться совместно. Например, `10u` — константа 10 типа **`unsigned int`**, `6uL` — константа 6 типа **`unsigned long`**. `7ll` — константа 7 типа **`long long`**.

По правилам языка **Си** знак `-` перед числом не является частью записи числа, а является унарной операцией, применённой к положительному числу. Поэтому, если размер типа `long long` равен 16 битам, запись `-32768` *неверна* и даст результат `0`! Для другого размера целых типов пример соответственно меняется.

1.1.2 Вещественные

Вещественная константа может содержать целую и дробную часть мантиссы и порядок. Вещественные числа могут записываться в десятичной или шестнадцатеричной (**C99**) системах счисления.

Запись вещественного числа в десятичной системе счисления такая же, как в языке **Паскаль** и может состоять из целой части, дробной части и порядка числа. Признаком вещественного числа является либо присутствие десятичной точки `.`, либо присутствие `e` или `E` для обозначения порядка. Примеры вещественных чисел приведены ниже:

```
1.2
1.
.5
1e4
1e+6
1.e-7
.7E12
```

Вещественное число может записываться и в шестнадцатеричной системе счисления. В этом случае перед числом ставится префикс `0x`, а знак порядка `p` или `P` обязателен. Целая и дробная часть мантиссы записывается в шестнадцатеричной системе счисления, а порядок

— в десятичной системе счисления. Порядок показывает, на какую степень двойки должна быть умножена мантисса. Например, запись `0x1c.2fp5` задаёт вещественное число, равное $(\texttt{0x1c} + \frac{\texttt{0x2f}}{\texttt{0x100}}) \cdot 2^5 = 901.875$. Другие примеры шестнадцатеричных вещественных чисел:

```
0x1p10
0x1.ddP-5
```

По умолчанию вещественные константы имеют тип **`double`**. Для явного задания типа можно использовать суффикс `l` или `L` для указания типа **`long double`**. Суффикс `f` или `F` для указания типа **`float`**. Например, `1.5L`, `0x3.aap10f`.

Как и в случае целых чисел, знак «плюс» или «минус» перед вещественным числом является не частью числа, а унарной операцией, применённой к положительному константному значению.

1.1.3 Литерные

Литерные константы мы уже рассматривали. Напомним, что литерные константы имеют тип **`int`**. В апострофах может быть записано несколько символов, в этом случае соответствующее целое значение зависит от компилятора.

Изначально на большинстве архитектур для хранения литерных значений использовался тип **`char`**, чего было достаточно для хранения всех значений из диапазона кодов ASCII (0—127). Затем стали использоваться 8-битные кодировки, и для хранения кодов символов из диапазона (0—256) следует использовать тип **`unsigned char`**, чтобы избежать потенциальных ошибок, связанных с индексированием по коду символа.

8-битные кодировки и сейчас широко используются, но также используются и 16-битные кодировки (Unicode, или UCS-16), и кодировки с переменной длиной символа (UTF-8). Для представления литерных значений в расширенной кодировке используются «длинные» символьные литералы, например `L'a'`. Такие литералы имеют тип **`wchar_t`**, совпадающий с некоторым целым типом, диапазона значений которого достаточно для представления всех символов в расширенной кодировке, используемой на данной платформе или в данном языковом окружении.

1.1.4 Строковые

Строковые литералы имеют тип **`char const *`** (константный указатель на тип **`char`**). Все строковые константы имеют в конце строки неявный символ с кодом `'\0'` — терминатор строки. Строка не имеет явного поля длины, в отличие от языка Turbo Pascal. Сейчас мы рассмотрим работу со строками подробнее.

1.2 Простейшая работа со строками

Язык Си не поддерживает специального строкового типа. Строки хранятся в массивах типа **`char`**, **`signed char`** или **`unsigned char`**. Большой набор функций работы со строками предоставляется стандартной библиотекой. В языке Си одной строкой является последовательность символов, завершающаяся специальным символом-терминатором с кодом `0`. Таким образом,

- строки не содержат длины строки явно и для вычисления длины её необходимо просматривать до первого символа с кодом `0`,

- нет ограничений языка на длину строки,
- в строке не может присутствовать символ с кодом 0.

Переменная для хранения строки определяется как массив

```
char str[N];
```

где N задаёт объем памяти, отводимый для строки. Поскольку строка всегда хранит символ-терминатор '\0', максимальное число значащих символов в такой строке на единицу меньше (N-1 символ). *Неправильное выделение памяти под символьную строку — одна из распространённых и опасных ошибок программирования на* **Си**!

Строки можно инициализировать следующим образом:

```
char str[20] = "a␣string";
```

В этом примере будет задано значение 9 байт (8 значащих символов и один символ-терминатор), остальные 11 байт будут либо обнулены, либо содержать произвольное значение.

Символьные массивы можно определять без указания размера, если присутствует инициализация. Например,

```
unsigned char x[] = "xxx";
```

В этом случае под массив x будет выделено 4 байта.

Обратите внимание, что в Си нет «строк переменного размера», то есть строк, память под которые автоматически расширяется, если строка становиться длиннее текущего буфера. При работе со строками необходимо внимательно следить за тем, чтобы строка копировалась в буфер достаточного размера!

По аналогии с длинными символьными литералами есть и «длинные» символьные строки, которые записываются с префиксом L, например, L"string". Для хранения таких строк используются массивы типа **wchar_t**.

Для манипуляций со строками в языке **Си** используются стандартные библиотечные функции. Чтобы их можно было использовать, в начале программы нужно подключить заголовочный файл string.h.

```
#include <string.h>
```

Некоторые из этих функций мы рассмотрим ниже.

Функция strcmp, прототип которой упрощённо записывается

```
int strcmp(char s1[], char s2[]);
```

сравнивает две строки s1 и s2. Если строка s1 лексикографически меньше строки s2, функция возвращает отрицательное значение. Если строка s1 лексикографически больше строки s2, функция strcmp возвращает положительное значение. Если две строки равны, функция возвращает ноль. Проверка на равенство двух строк записывается несколько неожиданным способом

```
if (!strcmp(s1, s2)) { /* строки равны */ }
```

Обратите внимание, что *строки нельзя сравнивать обычными операциями сравнения* !=, == и т. д. Дело в том, что в этом случае будут сравниваться не значения строк, а адреса этих строк. Компилятор в этом случае не даст никаких предупреждений.

Функция strlen с прототипом

```
int strlen(char s[]);
```

вычисляет количество значащих символов в строке, то есть число символов до символа-терминатора. Например `strlen("xx")` даёт результат 2.

Функция `strcpy` с прототипом

```
char *strcpy(char dst[], char src[]);
```

копирует строку `str` в строку `dst`, включая символ-терминатор. Забегая вперёд, функция возвращает указатель на первый символ строки `dst`.

Как и везде в языке Си, контроль переполнения массива полностью лежит на программисте. Существует вариант этой функции (`strncmp`), который позволяет ограничить максимальное количество копируемых символов.

Функция `strncpy` имеет следующий прототип:

```
char *strncpy(char dst[], char src[], size_t n);
```

Здесь **`size_t`** — это тип, который используется в стандартной библиотеке языка **Си** для размеров объектов. Это один из беззнаковых целых типов (обычно **`unsigned long`**). Функция `strncpy` работает следующим образом:

- Если длина строки `src` меньше значения `n`, то в буфер `dst` копируется вся строка, а остаток буфера `dst` заполняется нулевыми байтами.
- Если длина строки `src` больше или равна `n`, то в буфер `dst` копируеются первые `n` символов строки. *При этом в конец скопированной строки нулевой байт-терминатор не добавляется!*

Как и `strcpy`, эта функция возвращает адрес первого символа строки `dst`. Для копирования строки в другую строку удобнее использовать функцию `snprintf`.

Функция `strcat` с прототипом

```
char *strcat(char dst[], char src[]);
```

добавляет строку `str` в конец строки `dst`. Контроля количества скопированных символов не производится. Существует вариант этой функции (`strncat`), который позволяет ограничить число добавляемых символов.

Строку очистить можно с помощью функции `strcpy`

```
strcpy(s, "");
```

а можно проще:

```
s[0] = 0;
```

1.3 Ввод/вывод строк

Для вывода строк предусмотрен специальный спецификатор формата `%s` который можно использовать в функциях семейства `printf`. Например,

```
printf("His␣name␣is␣%s\n", name);
```

Символ-терминатор на печать выводиться не будет.

Считывать строки можно аналогичным спецификатором формата `%s` для функций семейства `scanf`. Например,

```
char buf[20];
scanf("%s", buf);
```

В этом случае сначала будут пропущены все пробельные символы, затем все символы до первого пробельного символа будут занесены в `buf`. После этого в `buf` будет добавлен символ-терминатор `'\0'`. Обратите внимание на отсутствие знака `'&'` перед `buf` в аргументах вызова `scanf` — массивы и так передаются по ссылке. Буфер `buf` должен быть достаточного размера, чтобы вместить все символы вводимой строки. Поскольку вводимые данные чаще всего не находятся под контролем программиста, то есть пользователь, возможно злонамеренно, может вводить строки произвольной длины, *использование такого неконтролируемого чтения в реальных программах очень опасно*. Задать ограничение на размер считываемой строки можно следующим образом:

```
char buf[20];
scanf("%19s", buf);
```

В данном случае в строку `buf` будет считано не более 19 символов, и в конец строки будет добавлен байт-терминатор `'\0'`. Поэтому *никогда не используйте спецификатор %s без ограничения размера вводимой строки!*.

Другой полезный спецификатор для `scanf` — `%n`. Этому спецификатору должна соответствовать переменная целого типа со знаком `&` (адрес). В эту переменную заносится число символов, прочитанное к моменту, когда был встречен этот спецификатор. Спецификатор `%n` не учитывается в числе успешно считанных спецификаторов в возвращаемом значении функции `scanf`.

Очень полезной является функция `sprintf` с прототипом

```
int sprintf(char s[], char format, ...);
```

Которая работает точно также, как `printf`, поддерживает все форматы вывода, но вместо печати на стандартный поток вывода, заполняет строку `s`. В конец строки `s` добавляется символ-терминатор. Результатом работы функции является количество выведенных символов (не считая символ-терминатор). Например,

```
sprintf(s, "%d", n);
```

позволяет получить в строке `s` символьное представление числа `n`. Функция `sprintf` не контролирует количество символов, записанное в строку `s`, поэтому возможно её переполнение.

В **C99** введена функция `snprintf` — безопасный вариант `sprintf`.

```
int snprintf(char s[], size_t n, char format, ...);
```

В строку `s` будет записано не более чем `n - 1` символов согласно спецификации формата, и всегда будет добавлен символ-терминатор строки. Функция возвращает количество символов, которые были бы записаны в строку `s`, если её размер был бы неограничен.

Следующий фрагмент выполняет конкатенацию двух строк `str1` и `str2` и помещает результат в строку фиксированного размера `buf`.

```
int buf[64];
snprintf(buf, sizeof(buf), "%s%s", str1, str2);
```

В случае сомнений используйте `snprintf`!

«Обратная» функция `sscanf` с прототипом

```
int sscanf(char s[], char format, ...);
```

считывает из форматные данные из строки вместо стандартного потока ввода. Функция возвращает число успешно считанных полей (как и `scanf`). Например,

```
sscanf(s, "%d", &n);
```

считывает в переменную `n` число из строки `s`. Для `sscanf` особенно полезен упоминавшийся выше спецификатор ввода `%n`. Он позволяет узнать, на каком символе строки остановилась функция `sscanf`.

Слово «строка» в русском языке может обозначать два совершенно разных понятия. Во-первых, это просто цепочка символов (string); во вторых, это последовательность символов, занимающая один ряд на экране или в напечатанном тексте (line). В первом случае иногда говорят «цепочка».

Очень часто бывает так, что стандартный поток ввода ассоциирован либо с терминалом, либо с файлом, который имеет текстовую структуру, то есть разбит на строки (line). В языке Си входной поток является последовательностью символов. Строки текстового файла во входном потоке разделяются символом `'\n'`, не зависимо от того, какой разделитель строк текста в действительности используется в операционной системе (например, в MS-DOS используются два символа `'\r'`, `'\n'`). Входной поток не имеет специального символа-признака конца входного потока. Таким образом, входной файл вида

```
a
b b
cc
```

будет представлен как поток символов

```
'a', '\n', 'b', ' ', 'b', '\n', 'c', 'c', '\n', '\n'
```

Функция `gets` с прототипом

```
char * gets(char s[]);
```

считывает одну строку входного текстового файла, то есть последовательность символов до символа `'\n'`, включая его. Считанные символы помещаются в строку `s`, причём символ `'\n'` заменяется на символ-терминатор `'\0'`. В случае, если достигнут конец входного потока, функция возвращает специальную константу `NULL`. Обратите внимание, что когда входной поток ассоциирован с терминалом, для того, чтобы был отмечен конец потока, нужно нажать специальную комбинацию клавиш. Буфер `s` должен быть достаточного размера, чтобы вместить всю считываемую последовательность символов. *Никогда не используйте эту функцию. Используйте `fgets`!*

Функция `fgets` позволяет считать одну строку входного текстового файла из произвольного потока. Пока мы будем её использовать только для стандартного потока ввода `stdin`.

```
char * fgets(char s[], int n, FILE *f);
```

Функция считывает из потока не более чем `n - 1` символов. Чтение прерывается, если считано `n - 1` символов или достигнут конец строки текста `'\n'`. Символ `'\n'` помещается в строку `s` (*отличие от `gets`!*). В любом случае в конец строки `s` добавляется нулевой байт-терминатор строки. Пример использования функции:

```
1   unsigned char buf[128];
2   while (fgets(buf, sizeof(buf), stdin)) {
3           printf("%s", buf);
4   }
```

В этом примере цикл **while** завершится, как только функция `fgets` вернёт `NULL`.

Функция `puts` с прототипом

```
char *puts(char s[]);
```

добавляет в выходной поток символы из строки `s`. Символ-терминатор заменяется на символ завершения строки `'\n'`.

1.4 Побочный эффект в операциях инкремента и декремента

Основной эффект операции — это выработка некоторого значения, которое может быть далее использовано в выражении. Побочный эффект — это другое воздействие на среду выполнения (например, изменение значения переменной).

Нами уже были рассмотрены операции инкремента (увеличения на 1) переменной `++`, и декремента (уменьшения на 1) переменной `--`. В языке Си они существуют в двух формах: префиксной и постфиксной, которые отличаются основным эффектом операции. Преинкремент `++a` — значение переменной вначале увеличивается на 1, и это уже увеличенное значение является значением операции, и далее в выражении может быть использовано. Постинкремент `a++` — значением этой операции является значение переменной до увеличения на 1. Например,

```
int a = 5, b, c;
b = 5 + ++a;
c = 5 + a++;
```

значением переменной `b` будет 11, а переменной `c` — 10.

1.5 Пример работы со строками текста в файле

Рассмотрим такую задачу. На стандартном потоке ввода задаётся последовательность неотрицательных целых чисел. Последовательность разбита на строки текста так, что в каждой строке текста входного файла находится хотя бы одно число. Длина каждой строки текста не превышает 1022 символов. Для каждой строки текста во входном файле найти максимальное целое число, находящееся в этой строке, и напечатать его на стандартный поток вывода. Если строка не содержит чисел (только пробельные символы), на стандартный поток вывода ничего не печатать.

Для решения этой задачи мы поступим следующим образом: будем построчно читать входной файл с помощью функции `fgets`, затем анализировать каждую считанную строку с помощью функции `scanf`. Поскольку по условию задачи требуется анализировать файл строго по строкам текста, мы проверим, что в результате работы функции `fgets` строка была считана целиком.

```
1 #include <stdio.h>
2 #include <string.h>
3 enum { MAX_LEN = 1022 };
4 int main(void)
5 {
6   char buf[MAX_LEN + 2];
7   int len, cur, n, max, val;
8
9   while (fgets(buf, sizeof(buf), stdin)) {
```

```
10      len = strlen(buf);
11      if (len == sizeof(buf) - 1 && buf[sizeof(buf) - 2] != '\n') {
12        fprintf(stderr, "Line␣is␣too␣long\n");
13        return 1;
14      }
15      cur = 0;
16      max = -1;
17      while (sscanf(&buf[cur], "%d%n", &val, &n) == 1) {
18        if (val < 0) {
19          fprintf(stderr, "Invalid␣value\n");
20          return 1;
21        }
22        if (val > max) max = val;
23        cur += n;
24      }
25      n = 0;
26      sscanf(&buf[cur], "␣%n", &n);
27      cur += n;
28      if (buf[cur]) {
29        fprintf(stderr, "Invalid␣string\n");
30        return 1;
31      }
32      if (max >= 0) printf("%d\n", max);
33      else printf("\n");
34    }
35    return 0;
36  }
```

Обратите внимание, что запись `&buf[cur]` в строках 17 и 26 — это подстрока строки `buf`, начинающаяся с позиции `cur` и продолжающаяся до конца строки `buf`. Символ пробела в спецификации формата в строке 26 позволяет пропустить все пробельные символы в строке.

1.6 Упражнения

1. Написать функцию, которая реверсирует строку, переданную в качестве параметра.
2. Написать функцию, которая в массиве из целых чисел все числа, меньшие или равные заданному переставляет в начало массива, а все числа, большие или равные заданному — в конец массива.
3. Проверить на равенство строки (функция `strcmp`).
4. Скопировать из входного потока в выходной все строки, длина которых больше 20.
5. Скопировать из входного потока в выходной все строки, которые содержат целое число, большее 2004.