Семинар #3: Символы, строки и файлы.

Таблица ASCII

Символ	Код	C	K	C	K	C	K	C	K	C	K	C	K	C	K	C	K	C	K
\0	0	&	38	0	48	:	58	D	68	N	78	X	88	b	98	l	108	v	118
\t	9	’	39	1	49	;	59	E	69	O	79	Y	89	c	99	m	109	w	119
\n	10	(	40	2	50	<	60	F	70	P	80	Z	90	d	100	n	110	x	120
		)	41	3	51	=	61	G	71	Q	81	[	91	e	101	o	111	y	121
(пробел)	32	*	42	4	52	>	62	H	72	R	82	\	92	f	102	p	112	z	122
!	33	+	43	5	53	?	63	I	73	S	83	]	93	g	103	q	113	{	123
"	34	,	44	6	54	@	64	J	74	T	84	^	94	h	104	r	114	|	124
#	35	-	45	7	55	A	65	K	75	U	85	_	95	i	105	s	115	}	125
$	36	.	46	8	56	B	66	L	76	V	86	‘	96	j	106	t	116	~	126
%	37	/	47	9	57	C	67	M	77	W	87	a	97	k	107	u	117		

Символы

Целые числа `char`

На предыдущем семинаре были рассмотрены различные типы данных, используемые для представления целых чисел. Одним из таких типов является `char` — он предназначен для хранения знаковых целых чисел размером в 1 байт, то есть в диапазоне от –128 до 127. Такой тип данных удобно использовать для работы с небольшими числами. Для ввода и вывода значений типа `char` применяется спецификатор формата `%hhi`. Вот пример, в котором используются две переменные типа `char` как числа:

```
char a = 10;
char b = 30;
printf("%hhi\n", a);     // Напечатает 10
printf("%i\n", a + b);   // Напечатает 40
```

Как хранятся символы. Кодировка ASCII.

Все символы в памяти компьютера хранятся в виде целых чисел. Каждому символу соответствует определённый числовой код, с которым и работает процессор. Способ сопоставления символов числам называется *кодировкой*. Наиболее фундаментальной и распространённой кодировкой является ASCII. Она включает базовый набор символов: латинский алфавит, цифры, специальные знаки и управляющие коды. Стандарт ASCII определяет 128 символов, и именно он лёг в основу всех последующих систем кодирования. Важно помнить, что компьютер хранит и обрабатывает символы исключительно как числа. Кодировка используется для преобразования этих чисел в символы при выводе или обратно при вводе.

Так как ASCII кодирует 128 символов, поэтому для их хранения достаточно 7 бит. Однако из-за того, что работа с байтами (8 бит) эффективнее, каждый символ всё равно занимает ровно один байт, где старший бит не используется. В языке C для хранения таких числовых кодов используется тип `char`.

Спецификатор `%c` в функции `printf`

Спецификатор `%c` в функции `printf` предназначен для вывода символа. Он преобразует переданное целое число в соответствующий ему символ согласно таблице кодировки и печатает результат.

```
char a = 64;
printf("%hhi\n", a);     // Напечатает 64
printf("%c\n", a);       // Напечатает @
```

Спецификатор %c в функции scanf

Спецификатор %c в функции scanf предназначен для считывания символа. Когда scanf видит такой спецификатор, он читает с экрана 1 символ, потом по таблице ASCII получает код этого символа и этот код записывает в переменную типа char.

```
char a;
scanf("%c", &a);
printf("%c\n", a);
```

Символьные литералы

Для удобства работы с символами с языке были введены символьные литералы. В коде они выглядят как символы в одинарных кавычках, но являются просто числами, соответствующими коду символа.

```
char a = '@';  // Теперь a равно 64
char b = '5';  // Теперь b равно 53
printf("%hhi %hhi %i\n", a, b, a * b);  // Напечатает 64 53 3392
printf("%c %c\n", a, b);                // Напечатает @ 5
```

Примеры программ, работающих с символами

- Программа, которая будет печатать на экран все символы с кодами от 32 до 126:

```
#include <stdio.h>
int main()
{
    for (int i = 32; i <= 126; ++i)
        printf("Symbol = %c, Code = %hhi\n", i, i);
}
```

- Программа, которая считывает один символ и если этот символ является заглавной буквой, то программа печатает Yes, иначе, программа печатает No.

```
#include <stdio.h>
int main()
{
    char a;
    scanf("%c", &a);
    if (a >= 65 && a <= 90)
        printf("Yes\n");
    else
        printf("No\n");
}
```

Но лучше, вместо чисел, соответствующим кодам символов использовать символьные литералы:

```
#include <stdio.h>
int main()
{
    char a;
    scanf("%c", &a);
    if (a >= 'A' && a <= 'Z')
        printf("Yes\n");
    else
        printf("No\n");
}
```

Но ещё лучше, вместо этого использовать удобные функции из библиотеки `ctype.h`. Например, функция `isupper` проверяет, является ли символ заглавной буквой:

```
#include <stdio.h>
#include <ctype.h>
int main()
{
    char a;
    scanf("%c", &a);
    if (isupper(a))
        printf("Yes\n");
    else
        printf("No\n");
}
```

- Программа, которая считывает символ и, если этот символ является строчной буквой, то делает эту букву заглавной и печатает. Если символ -– не строчная буква, то просто печатает его.

```
#include <stdio.h>
int main()
{
    char x;
    scanf("%c", &x);
    if (x >= 'a' && x <= 'z')
        x -= ('a' - 'A');
    printf("%c\n", x);
}
```

Или то же самое с использованием функций из библиотеки `ctype.h`:

```
#include <stdio.h>
#include <ctype.h>
int main()
{
    char x;
    scanf("%c", &x);
    if (islower(x))
        x = toupper(x);
    printf("%c\n", x);
}
```

Функции из библиотеки `ctype.h`

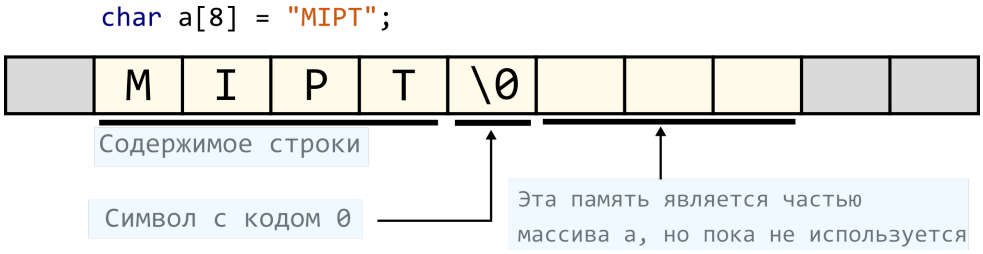
```
isalpha - проверить, что символ - буква (A-Z или a-z)
isupper - проверить, что символ - заглавная буква (A-Z)
islower - проверить, что символ - строчная буква (a-z)
isdigit - проверить, что символ - цифра
isspace - проверить, что символ - пробельный символ (пробел, '\n' или '\t')
toupper - переводит буквы нижнего регистра в верхний регистр
tolower - переводит буквы верхнего регистра в нижний регистр
```

Функции `putchar` и `getchar`

Функции `putchar` и `getchar` используются для печати одного символа и считывания одного символа соответственно. Можете использовать эти функции или можете использовать функции `printf`/`scanf` со спецификатором `%c` для печати/считывания одного символа.

Строки

В отличие от многих языков высокого уровня, в языке C не существует встроенного типа строки. Вместо этого строки реализованы как массивы элементов типа `char`, хранящими коды символов в определённой кодировке. Ключевая особенность, которая отличает обычный массив символов от строки, – это обязательное наличие *нуль-терминатора* (символа с кодом 0, который записывается как `'\0'`). Этот специальный байт, размещённый в конце последовательности символов, служит маркером конца строки.



Следует отличать длину строки от размера массива в котором она хранится. Например, на схеме изображённой выше, длина строки равна 4, а размер массива в котором хранится эта строка равен 8. Важно помнить, что для хранения строки длиной `n` необходим массив размером не менее `n + 1` символов, так как нужно учитывать нуль-терминатор.

Объявление и инициализация строк

Создавать строки можно как и массивы, но лучше через специальных синтаксис с помощью строкового литерала.

```
// Все три строки будут содержать "MIPT"
char a[10] = {77, 73, 80, 84, 0};
char b[10] = {'M', 'I', 'P', 'T', '\0'};
char c[10] = "MIPT"; // Символ 0 поставится автоматически
```

Однако, после того как строка создана, её нельзя изменить путём присваивания:

```
char a[10] = "Cat";  // OK, так можно инициализировать строку
a = "Dog";           // Ошибка, строки нельзя присваивать целиком
```

Присваивать строки нужно либо поэлементно в цикле либо с использованием стандартной функции `strcpy`.

Печать строк. Спецификатор %s

Обычные массивы нельзя печатать одной командой `printf`, но специально для строк ввели модификатор `%s`, благодаря которому можно печатать и считывать строки одной командой.

```
char a[10] = "MIPT";
printf("%s\n", a);  // Напечатает MIPT
```

Когда `printf` видит `%s` он начинает бежать от начала массива и печатать каждый символ пока не встретит нулевой символ. Если вдруг в массиве не будет нулевого символа, то `printf` выйдет за границы массива и это приведёт к неопределённому поведению.

Доступ к символам строки

Так как строка является массивом, то для доступа к её элементам можно использовать оператор индексирования, то есть квадратные скобочки:

```
char a[10] = "Cat";
printf("%c\n", a[2]);   // Напечатает t

a[1] = 'o';
printf("%s\n", a);      // Напечатает Cot
```

Считывание строк

Для считывания строк с экрана используется функция `scanf` со спецификатором `%s`.

```
char a[100];
scanf("%s", a); // Считываем строку, но строка должна быть меньше, чем 100 символов
                // Обратите внимание, что при считывании строк ставить & не нужно
                // Всё потому, что строки - это массивы, а они передаются в функции по
    адресу
```

Более точно, `scanf` делает следующее:

1. Перед началом чтения `scanf` считывает и игнорирует все пробельные символы.
2. После этого `scanf` начинает считывать непробельные символы и последовательно размещать их в массиве. Этот процесс прекращается, как только встречается любой пробельный символ или достигается конец ввода (комбинация клавиш `Ctrl-D`).
3. Последний считанный пробельный символ не помещается в массив и остаётся в буфере стандартного ввода. Вместо этого `scanf` добавляет нулевой символ в конец записанной строки.

Если вы запустите программу и введёте слишком большую строку, больше размера массива, то `scanf` выйдет за границы массива и это приведёт к неопределённому поведению.

Безопасное считывание строк

Функции `scanf` можно указать максимальное количество символов для считывания:

```
char a[100];
scanf("%99s", a); // Безопасно считываем строку
```

Теперь, если строка на входе будет больше чем 99 символов, то в массив `a` запишется только первые 99 символов входящей строки и ещё останется место на нулевой символ. Таким образом, массив не переполнится ни при какой входящей строке.

Считывание до определённого символа

По умолчанию строка считывается до первого пробельного символа. Если вы хотите считать до определённого символа, то можно использовать следующий синтаксис:

```
char a[100];
scanf("%[^X]", a);  // Считываем до первого символа X
scanf("%[^@]", a);  // Считываем до первого символа @
scanf("%[^\n]", a); // Считываем до переноса строки
```

Буферизация стандартного входа

Буферизация – это стратегия, при которой данные не передаются сразу от источника к приёмнику, а накапливаются в промежуточной области памяти (*буфере*), и только затем, при заполнении буфера или наступлении необходимых условий, передаются дальше. Буферизация входного потока – это широко распространённая техника, используемая в программировании для ускорения операций ввода-вывода. Она применяется как при работе с файлами, так и при взаимодействии с терминалом.

В контексте языка C буферизация стандартного ввода означает, что символы, введённые вами в терминале (включая пробелы и переносы строк), не передаются программе немедленно. Вместо этого они накапливаются в специальной области памяти – буфере. По умолчанию, при вводе в терминале буферизация работает построчно, то есть строка передаётся из терминала в буфер при нажатии на клавишу `Enter`. Функции ввода, такие как `scanf`, обращаются уже к буферу. Понимание как работает буферизация, хотя бы на базовом уровне, необходимо, чтобы не совершать ошибки при нетривиальном считывании.

Рассмотрим как работает буферизация на следующем примере. В данной программе считывается сначала одно число, затем печатается это число. После этого считывается ещё одно число и также печатается это число.

```c
#include <stdio.h>
int main()
{
    int a;
    scanf("%i", &a);
    printf("a = %i\n", a);

    int b;
    scanf("%i", &b);
    printf("b = %i\n", b);
}
```

- Если сначала ввести одно число и нажать `Enter`, то программа считает число и напечатает его на экран и попросит ввести другое число. Если ввести второе число и нажать `Enter`, то программа также считает его, напечатает и завершится. В данном случае всё работает как и ожидалось.
- Если же в терминале сразу же ввести два числа, разделённых пробелом, то программа не будет запрашивать больше чисел, а сразу же считает оба числа и напечатает их. Может показаться, что в этом случае второй вызов функции `scanf` вообще не отработал. На самом же деле, во второй раз функция `scanf` считывала число из буфера, в котором сохранилось неиспользованное число с прошлого вызова функции `scanf`. Поэтому во второй раз функция `scanf` отрабатывает мгновенно.

Остаточные пробельные символы в буфере

Одна из самых частых ошибок при считывании, возникает если в буфере остались не прочтённые пробельные символы. Ошибку можно проиллюстрировать на следующем примере:

```c
#include <stdio.h>
int main()
{
    int a;
    scanf("%i", &a);        // Пользователь вводит: "123\n"
                            // Считывается "123", а "\n" остаётся в буфере
    char b;
    scanf("%c", &b);        // Считывает '\n' из буфера
    printf("b = %i\n", b);
}
```

Для того, чтобы не допускать подобные ошибки, важно знать что конкретно происходит при вызове функции `scanf` с различными спецификаторами:

- `scanf` со спецификаторами для чтения чисел (`%i`, `%f` и т. д.):
  Сначала "съедает" все пробельные символы, затем считывает символы пока они соответствуют формату числа или пока не дойдёт до конца буфера. Символы после числа, в том числе пробельные, остаются в буфере.
- `scanf` со спецификаторами для чтения строк (`%s`):
  Сначала "съедает" все пробельные символы, затем считывает символы пока не дойдёт до пробельного символа или до конца буфера. Символы после строки, в том числе пробельные, остаются в буфере.
- `scanf` со спецификаторами для чтения символов (`%c`) и функция `getchar`:
  Просто считывает символ, ничего не "съедает". Все остальные символы, остаются в буфере.
- `scanf` со спецификаторами для чтения строки до заданного символа (`%[^\n]`):
  Считывает все символы до заданного символа (в том числе пробельные, ничего не "съедает"). Заданный символ и символы, следующие после него остаются в буфере.
- Если `scanf` в своей строке форматирования увидит хоть один пробельный символ, то он будет "съедать" **все** пробельные символы пока не встретит непробельный.

Стандартные функции библиотеки `string.h`

- `size_t strlen(const char str[])` - возвращает длину строки
- `char* strcpy (char a[], const char b[]))` - копирует строку `b` в строку `a`, т.е. аналог `a = b`.
- `char* strcat(char a[], const char b[])` - приклеивает копию строки `b` к строке `a`, т.е. аналог `a += b`.
- `int strcmp(const char a[], const char b[])` - лексикографическое сравнение строк (возвращает `0`, если строки одинаковые, положительное, если первая строка больше, и отрицательное, если меньше)
- `sprintf` - аналог `printf`, но вместо печати на экран, 'печатает' в строку.
- `sscanf` - аналог `scanf`, но вместо считывания на экран, 'считывает' из строки.

```c
#include <stdio.h>
#include <string.h>
int main()
{
    char a[100] = "Cat";
    char b[100] = "Dog";

    // Строки это массивы, поэтому их нельзя просто присваивать
    a = b; // Это не будет работать! Нужно использовать strcpy:
    strcpy(a, b);

    // Конкатенация (склейка) строк. Можно воспринимать как +=
    a += b; // Это не будет работать! Нужно использовать strcat:
    strcat(a, b);

    // Строки это массивы, поэтому их нельзя просто сравнивать
    if (a == b) {} // Это не будет работать! Нужно использовать strcmp:
    if (strcmp(a, b) == 0)) {}

    // Печатаем в строку. После этого в a будет лежать строка "(10:20)"
    sprintf(a, "(%i:%i)", 10, 20);
}
```

Аргументы командной строки

Программы могут принимать аргументы. Простейший пример – утилита `ls`. Если запустить `ls` без аргументов:

```
ls
```

то она просто напечатает содержимое текущей директории. Если же использовать эту программу с опцией `-l`:

```
ls -l
```

то на экран выведется подробное описание файлов и папок в текущей директории. Поведение программы `ls` изменилось так как изменились её аргументы командной строки. Или, например, когда мы компилируем программу, мы пишем что-то вроде этого:

```
gcc main.c -o result
```

В данном случае, строки `"gcc"`, `"main.c"`, `"-o"` и `"result"` являются аргументами командной строки. Обратите внимание, что название программы тоже считается аргументом командной строки.

В случае передачи информации программе через аргументы командной строки, информация передаётся при вызове программы. Чтобы передать что-либо программе через аргументы командной строки, нужно написать это в терминале при запуске программы сразу после её запуска.

Например, если мы хотим передать программе `a.out` строку `cat`, то программу нужно вызвать так:

```
./a.out cat
```

Если же мы хотим передать программе `a.out` число `123`, то программу нужно вызвать так:

```
./a.out 123
```

Только нужно помнить, что аргументы командной строки всегда воспринимаются как строки и в данном случае число `123` передастся как строка `"123"`.

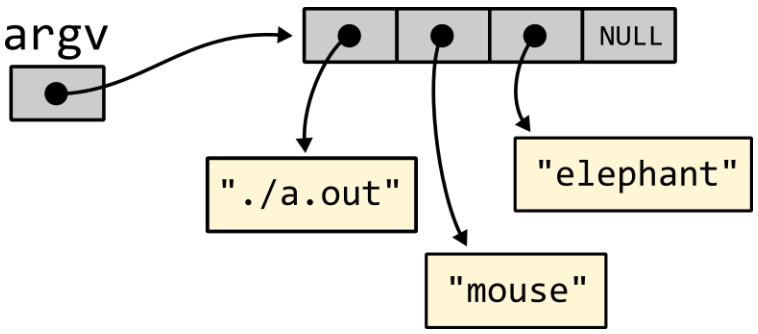
Если же мы хотим передать несколько аргументов, то просто перечисляем их через пробел:

```
./a.out mouse elephant
```

Аргументы командной строки можно получить в программе если использовать специальный вариант функции `main` с двумя аргументами, которые обычно называют `argc` и `argv`. Вот пример программы, которая печатает на экран все аргументы командной строки:

```c
#include <stdio.h>
int main(int argc, char** argv)
{
    for (int i = 0; i < argc; ++i)
    {
        printf("%s\n", argv[i]);
    }
}
```

- `argc` – это количество аргументов командной строки
- `argv` – это массив строк размера `argc`. Каждый элемент этого массива – это соответствующий аргумент командной строки.



Работа с текстовыми файлами

Текстовые файлы – это такие файлы, которые содержат только текст. В данном семинаре будем рассматривать только текстовые файлы в кодировке ASCII. Рассмотрим простейшую программу, которая создаёт файл `myfile.txt` и записывает туда строку `"Hello world!"`:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    FILE* fp = fopen("myfile.txt", "w");
    if (fp == NULL)
    {
        printf("Error!\n");
        exit(1);
    }
    fprintf(fp, "Hello world!");
    fclose(fp);
}
```

Разберём эту программу подробно:

- Функции для работы с файлами хранятся в библиотеке `stdio.h`.
- Подключаем библиотеку `stdlib.h`, так как будем использовать функцию `exit` для выхода из программы.
- Функция `fopen` используется для открытия файла. Ей нужно передать название название файла и режим открытия файла.
- Путь до файла считается от исполняемого файла. То есть, в данном примере, новый файл `myfile.txt` появится в той же папке, что и исполняемый файл.
- Основные режимы открытия файла:
  - `"r"` – открыть существующий файл для чтения (read). Если файл не существует, то функция вернёт значение `NULL`.
  - `"w"` – создать новый файл и открыть его для записи (write). Если файл с таким именем уже существует, то его содержимое удалится перед записью.
  - `"a"` – открыть для записи в конец файла (append). Если файл не существует, то он будет создан.
- Функция `fopen` возвращает специальный объект типа `FILE*` – указатель на структуру `FILE`. Используя этот объект, мы будем взаимодействовать с файлом. `fopen` возвращает `NULL` если при попытке открытия файла произошла ошибка.
- Функция `fprintf` используется для печати в файл. Она очень похожа на функцию `printf`, только первым аргументом нужно передать указатель, который мы получили из функции `fopen`.
- Аналогично, существует функция `fscanf`, которая считывает из файла и работает очень похоже на функцию `scanf`, только первым аргументом нужно передать указатель, который мы получили из функции `fopen`.
- Функция `fclose` закрывает файл. При закрытии файла освобождаются все ресурсы, которые были выделены предыдущими функциями.

Глобальные потоки `stdout` и `stdin`

В стандартной библиотеке определены глобальные переменные `stdout` и `stdin`, имеющие тип `FILE*`. Эти переменные соответствуют стандартному выводу (печать на экран) и стандартному вводу (считывание с экрана). Их можно использовать для передачи в функции для работы с файлами. Например:

```
printf("Hello\n");              // Печатает Hello на экран
fprintf(stdout, "Hello\n");     // Тоже печатает Hello на экран
```

Посимвольное чтение и запись

Ещё одна функция, которая может быть очень полезна для считывания из файла – это функция `fgetc`.

- `int fgetc(FILE* file)` – читает один символ из файла и возвращает его ASCII код. Если символов не осталось, то функция возвращает специальное значение, равное константе `EOF` (обычно она равна `-1`). Обратите внимание, что функция возращает значение типа `int`, а не `char`. Это необходимо, так как возвращаемое значение может принимать ещё одно дополнительное значение (`EOF`) в дополнении ко всем возможным кодам символов.
- `int fputc(int c, FILE* file)` – записывает символ, соответствующий коду `c`, в файл.

Пример программы, которая находит количество символов, являющимися цифрами, в файле:

```
#include <stdio.h>
int main()
{
    FILE* f = fopen("input.txt", "r");
    int c;
    int num_of_digits = 0;

    while ((c = fgetc(f)) != EOF)
    {
        if (c >= '0' && c <= '9')
            num_of_digits += 1;
    }
    printf("Number of digits = %d\n", num_of_digits);
    fclose(f);
}
```