Семинар #3: Строки. Классные задачи.

Таблица ASCII

Символ	Код	С	К	С	К	С	К	С	К	С	К	С	К	С	К	С	К	С	К
\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

Тип `char`. Спецификатор `%c` в функции `printf`

Тип `char` – это тип целочисленных чисел размером 1 байт. Часто используется для хранения кодов символов. Для считывания и печати чисел типа `char` используется спецификатор `%hhi`. Функция `printf` со спецификатором `%c` принимает на вход число и печатает соответствующий символ по таблице `ASCII`.

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

Спецификатор `%c` в функции `scanf`

Функция `scanf` со спецификатором `%c` считывает 1 символ и записывает код `ASCII` этого символа по соответствующему адресу.

```
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.

```
int a = '@'; // Теперь a равно 64
int b = '5'; // Теперь b равно 53
printf("%i %i %i\n", a, b, a * b);
```

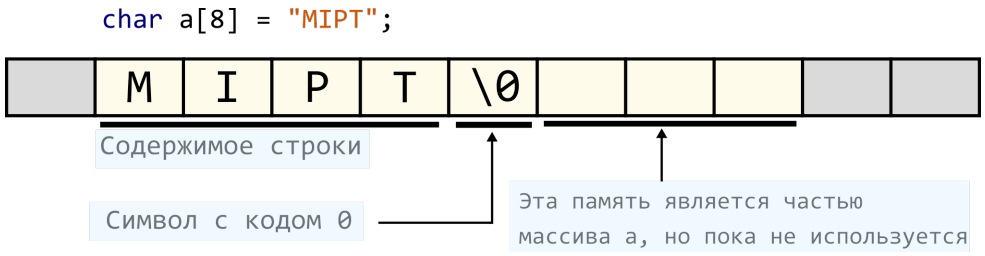
Библиотека `ctype.h`

В библиотеке `ctype.h` содержатся полезные функции для работы с символами:

```
isalpha - проверить, что символ - буква (A-Z или 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
```

Строки:

Строки - это массивы чисел типа `char`, которые хранят коды символов. Самое значительное отличие строк от массивов это то, что конец строки задаётся как элемент массива символом с кодом `0`.



Объявление, инициализация и изменение строк

Создавать строки можно также как и массивы, а можно и с помощью строки в двойных кавычках.

```
char a[10] = {77, 73, 80, 84, 0};
char b[10] = {'M', 'I', 'P', 'T', '\0'};
char c[10] = "MIPT"; // Символ 0 поставится автоматически
// Использовать = со строками можно только при создании, то есть это работать не будет:
// a = "CAT";
// Изменение элементов строк работает также как и у массивов:
a[1] = 'A';
```

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); // Печатаем строку
```

Считывание строк

`scanf` со спецификатором `%s` считывает строку до первого пробельного символа. Но если вы введёте слишком большую строку, которая не поместится в массив, то произойдёт ошибка – выход за границы массива.

```
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
```

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 считается только первые 99 символов этой строки.

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
```

Однако нужно быть осторожным, используя такой вид считывания. Дело в том, что функция `scanf` с обычными спецификаторами (такими как `%i` или `%s`) перед основным считыванием считывает все пробельные символы. Но, в режиме считывании до определённого символа, такого предварительного считывания пробельных символов не происходит. Поэтому, выполнение следующей программы приводит к ошибке:

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

    char str[100];
    scanf("%[^\n]", str);
    printf("str = %s\n", str);
}
```

Допустим мы запустили эту программу, ввели `123` и нажали `Enter`. Тогда в буфере будет лежать строка `"123\n"`. После того, как первый `scanf` отработает и считает число `123`, в буфере останется лежать лишь один символ переноса строки. Второй `scanf` должен считать до символа переноса строки, но этот символ лежит в буфере первым. Таким образом, второй `scanf` не считает ничего и строка `str` останется неинициализированной.

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.

```
#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"`.

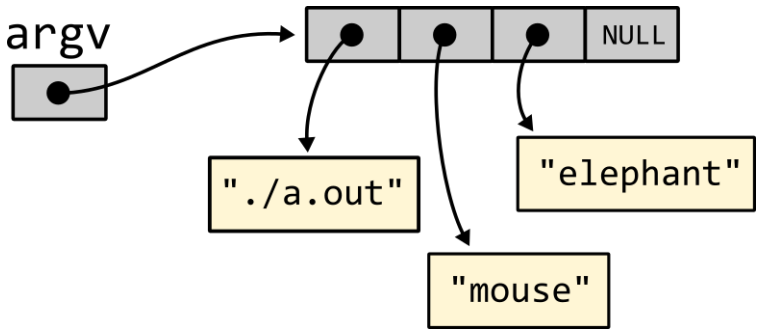
Если же мы хотим передать несколько аргументов, то просто перечисляем их через пробел:

```
./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 возвращает 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);
}
```