ЛЕКЦИЯ 11
СТРОКИ И АВТОМАТЫ

АЛГОРИТМИЗАЦИЯ И
ПРОГРАММИРОВАНИЕ

ЛЕКТОР ФУРМАВНИН С.А.

ЗНАЧЕНИЯ СИМВОЛОВ

То, что вы привыкли считать символами, на самом деле кодируется числами.

```
int c = 'A';
std::cout << c << std::endl; // что на экране?
```

Небольшие числа – это коды для символов

```
char x = 65;
std::cout << x << std::endl; // а сейчас?
```

В Unix формат символов – это UTF-8
Его первые 127 символов совпадают с ASCII

СПЕЦИАЛЬНЫЕ СИМВОЛЫ

0	NULL		NUL	CTRL+@
1	Start of head		SOH	CTRL+A
2	Start of text		STX	CTRL+B
3	End of text		ETX	CTRL+C
4	End of xmit		EOT	CTRL+D
5	Enquiry		ENQ	CTRL+E
6	Acknowledge		ACK	CTRL+F
7	Bell	\a	BEL	CTRL+G
8	Backspace	\b	BS	CTRL+H
9	Horz tab	\t	TAB	CTRL+I
10	Line feed	\n	LF	CTRL+J
11	Vert tab	\v	VT	CTRL+K
12	Form feed	\f	FF	CTRL+L
13	Carriage feed	\r	CR	CTRL+M
14	Shift out		SO	CTRL+N
15	Shift in		SI	CTRL+O

16	Data line escape	DLE	CTRL+P
17	Device ctrl 1	DC1	CTRL+Q
18	Device ctrl 2	DC2	CTRL+R
19	Device ctrl 3	DC3	CTRL+S
20	Device ctrl 4	DC4	CTRL+T
21	Neg acknowledge	NAK	CTRL+U
22	Sync idle	SYN	CTRL+V
23	End xmit block	ETB	CTRL+W
24	Cancel	CAN	CTRL+X
25	End of medium	EM	CTRL+Y
26	End of file	EOF	CTRL+Z
27	Escape	ESC	CTRL+[
28	File separator	FS	CTRL+\
29	Group separator	GS	CTRL+]
30	Record separator	RS	CTRL+^
31	Unit separator	US	CTRL+_

ТАБЛИЦА ASCII

0 -	16 - ▶	32 -	48 - 0	64 - @	80 - P	96 - '	112 - p
1 - ☺	17 - ◀	33 - !	49 - 1	65 - A	81 - Q	97 - a	113 - q
2 - ☻	18 - ↕	34 - "	50 - 2	66 - B	82 - R	98 - b	114 - r
3 - ♥	19 -‼	35 - #	51- 3	67 - C	83 - S	99 - c	115 - s
4 - ♦	20 - ¶	36 - $	52 - 4	68 - D	84 - T	100 - d	116 - t
5 - ♣	21 - §	37 - %	53 - 5	69 - E	85 - U	101 - e	117 - u
6 - ♠	22 - ▬	38 -&	54 - 6	70 - F	86 - V	102 - f	118 - v
7 -	23 - ↨	39 - '	55 - 7	71 - G	87 - W	103 - g	119 - w
8 -	24 - ↑	40 - (	56 - 8	72 - H	88 - X	104 - h	120 - x
9 -	25 - ↓	41 -)	57 - 9	73 - I	89 - Y	105 - i	121 - y
10 -	26 - →	42 - *	58 - :	74 - J	90 - Z	106 - j	122 - z
11 -	27 - ←	43 - +	59 - ;	75 - K	91 - [	107 - k	123 - {
12 -	28 - ∟	44 - ,	60 - <	76 - L	92 - \	108 - l	124 - |
13 -	29 - ↔	45 - -	61 - =	77 - M	93 - j	109 - m	125 - }
14 - ♫	30 - ▲	46 - .	62 ->	78 - N	94 - ^	110 - n	126 - ~
15 - ☼	31 - ▼	47 - /	63 - ?	79 - O	95 - ÷	111 - o	127 - ⌂
16 - ▶	32 -	48 -0	64 - @	80 - P	96 -	112 - p	

КАТЕГОРИИ СИМВОЛОВ

- Специальный заголовочный файл `<cctype>` содержит прототипы функций, категоризирующих символы
- Например, `isalpha(c)` проверяет, является ли символ одним из алфавитных символов (abcdef….), а `isdigit(c)` – является ли цифрой. Их объединяет `isalnum(c)`
- Иногда в программах, которые анализируют символьную информацию, важно знать, что на вход пришло что-то вроде буквы или что-то вроде пробела.

КАТЕГОРИИ СИМВОЛОВ

ASCII values			characters	iscntrl iswcntrl	isprint iswprint	isspace iswspace	isblank iswblank	isgraph iswgraph	ispunct iswpunct	isalnum iswalnum	isalpha iswalpha	isupper iswupper	islower iswlower	isdigit iswdigit	isxdigit iswxdigit
decimal	hexadecimal	octal													
0–8	\x0–\x8	\0–\10	control codes (NUL, etc.)	≠0	0	0	0	0	0	0	0	0	0	0	0
9	\x9	\11	tab (\t)	≠0	0	≠0	≠0	0	0	0	0	0	0	0	0
10–13	\xA–\xD	\12–\15	whitespaces (\n, \v, \f, \r)	≠0	0	≠0	0	0	0	0	0	0	0	0	0
14–31	\xE–\x1F	\16–\37	control codes	≠0	0	0	0	0	0	0	0	0	0	0	0
32	\x20	\40	space	0	≠0	≠0	≠0	0	0	0	0	0	0	0	0
33–47	\x21–\x2F	\41–\57	!"#$%&'()*+,-./	0	≠0	0	0	≠0	≠0	0	0	0	0	0	0
48–57	\x30–\x39	\60–\71	0123456789	0	≠0	0	0	≠0	0	≠0	0	0	0	≠0	≠0
58–64	\x3A–\x40	\72–\100	:;<=>?@	0	≠0	0	0	≠0	≠0	0	0	0	0	0	0
65–70	\x41–\x46	\101–\106	ABCDEF	0	≠0	0	0	≠0	0	≠0	≠0	≠0	0	0	≠0
71–90	\x47–\x5A	\107–\132	GHIJKLMNOP QRSTUVWXYZ	0	≠0	0	0	≠0	0	≠0	≠0	≠0	0	0	0
91–96	\x5B–\x60	\133–\140	[\]^_`	0	≠0	0	0	≠0	≠0	0	0	0	0	0	0
97–102	\x61–\x66	\141–\146	abcdef	0	≠0	0	0	≠0	0	≠0	≠0	0	≠0	0	≠0
103–122	\x67–\x7A	\147–\172	ghijklmnop qrstuvwxyz	0	≠0	0	0	≠0	0	≠0	≠0	0	≠0	0	0
123–126	\x7B–\x7E	\172–\176	{|}~	0	≠0	0	0	≠0	≠0	0	0	0	0	0	0
127	\x7F	\177	backspace character (DEL)	≠0	0	0	0	0	0	0	0	0	0	0	0

МЫ НЕ ПОЛЕЗЕМ ВГЛУБЬ

Далее мы будем работать только с ASCII строками, то есть со строками, состоящими из однобайтовых символов UTF8

Учет различных кодировок, русских букв, китайских иероглифов и всего такого просто не входит в нашу программу

СТРОКОВЫЕ ЛИТЕРАЛЫ

Следующая строка выглядит подозрительно знакомой

```
std::cout << “Hello, world!” << std::endl;
```

Символы “Hello, world!” в кавычках – это строковый литерал

H	e	l	l	o	,		w	o	r	l	d	!	\0

Строковый литерал – это строка, известная на этапе компиляции

Любая С-строка заканчивается нулевым символом (null-terminator)

Нулевой символ `'\0'` – это не символ `'0'`. Это спецсимвол, имеющий код 0.

ИЗМЕНЯЕМЫЕ СТРОКИ

Самый простой способ создать изменяемую строку – это объявить массив символов.

```
char hello[20] = “Hello, world!”;
```

Символы после завершающего нуля (он тут 14-й) содержат мусор
Далее содержимое этой строки можно поменять

```
hello[1] = ‘a’;
std::cout << hello << std::endl; // ->Hallo, world!
```

Изменяемая строка – это любой массив символов, завершающийся нулем.
Символы после завершающего нуля ничего не значат.

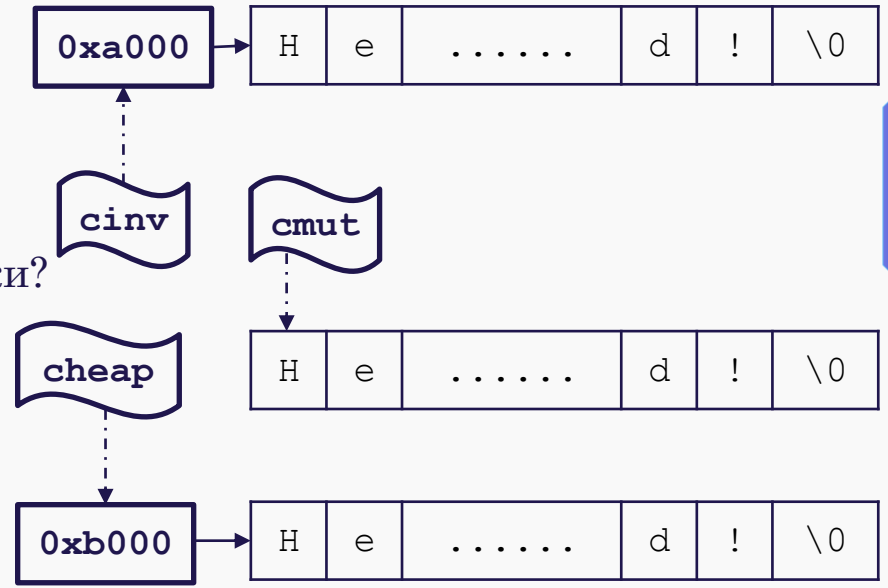
C-СТРОКИ И РАЗНЫЕ ВИДЫ ПАМЯТИ

Обычно используют указатель на первый элемент

```
char const * cinv = “Hello, world!”;
char cmut[] = “Hello, world!”;
char *cheap = new char[50];
strcpy(cheap, cinv);
```

Что вы думаете про следующие строки?

```
cheap = cinv; // ???
cinv = 0; // ???
cmut = cheap; // ???
```



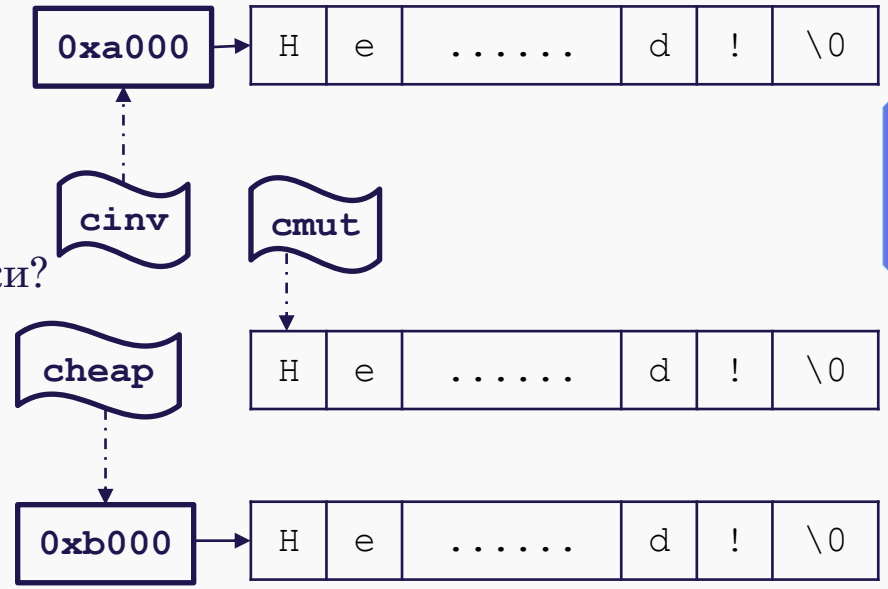
C-СТРОКИ И РАЗНЫЕ ВИДЫ ПАМЯТИ

Обычно используют указатель на первый элемент

```
char const * cinv = “Hello, world!”;
char cmut[] = “Hello, world!”;
char *cheap = new char[50];
strcpy(cheap, cinv);
```

Что вы думаете про следующие строки?

```
cheap = cinv; // CE + ML
cinv = 0; // OK
cmut = cheap; // CE
```



УПРАЖНЕНИЕ. СУММИРОВАНИЕ СТРОК

Ваша задача – написать программу, которая суммирует строчку, воспринимая символы в ней, как коды этих символов

H	e	l	l	o	,		w	o	r	l	d	!	\0
#72	#101	#108	#108	#111	#44	#32	#119	#111	#114	#108	#100	#33	#0

Для “Hello, world!” ответом будет 1161

КОПИРОВАНИЕ СТРОК

Одна строка может быть скопирована в другую

```
char dublicate[20];
const char *src = “Hello, world!”;
char *dst = dublicate;
while (*src != ‘\0’) {
    *dst = *src;
    dst += 1;
    src +=1;
}
*dst = \0;
```

КОПИРОВАНИЕ СТРОК

Одна строка может быть скопирована в другую

```
char dublicate[20];
const char *src = “Hello, world!”;
char *dst = dublicate;
while ((*dst++ = *src++) != ‘\0’) {}
// немного джигитовки
```

КОПИРОВАНИЕ СТРОК

Одна строка может быть скопирована в другую

```
#include <cstring>

char dublicate[20];
const char *src = “Hello, world!”;
strcpy(dublicate, src); // библиотечная функция
```

Вашим выбором по умолчанию должно быть именно использование библиотечных функций, так как они почти всегда корректны и куда лучше оптимизированы

В языке C и C++ много библиотечных функций для работы со строками

API ДЛЯ РАБОТЫ СО СТРОКАМИ

Библиотечная функция	Что делает
`strcpy(dest, src)`	Копирует `src` в `dest`
`strcat(dest, src)`	Дописывает `src` в конец `dest`
`strlen(src)`	Вычисляет длину `src`
`strcmp(src1, src2)`	Сравнивает `src1` и `src2`. Возвращает 0, 1, -1
`strchr(src, c)`	Ищет символ в строке
`strstr(haystack, needle)`	Ищет подстроку в строке
`strspn(src1, src2)`	Ищет максимальный префикс `src1` состоящий только из символов `src2`
`strcspn(src1, src2)`	Тоже, только не из символов `src2`
`strpbrk(src1, src2)`	Ищет первое вхождение любого символа из `src2`

ОБСУЖДЕНИЕ

Одна забавная странность в сигнатурах стандартной библиотеки заключается в том, что возвращаемый тип всегда `char*`

```
char* strstr(const char *s1, const char *s2);
```

Казалось бы, если мы принимаем оба аргумента `const char*`, почему бы и не возвращать `const char*`

У кого есть идеи почему так сделано?

УПРАЖНЕНИЕ. ПЕРЕВОРОТ ПОДСТРОК

Напишите программу, которая берет одно слово, а потом некоторый текст и переворачивает в этом тексте все вхождения этого слова.

Например, для слова «world» и текста «Hello, world!» результатом должно быть «Hello, dlrow!»

Вы можете предполагать, что текст состоит из слов, разделенных пробелами и пунктуацией, а слово состоит из алфавитных символов.

Вы также можете использовать любые функции работы с C-строками стандартной библиотеки, включая `strstr`

ОБСУЖДЕНИЕ

Еще одной проблемой `strcat` (кроме обсужденных ранее проблем с асимптотикой) является возможное переполнение буфера

Допустим, у нас строки живут только в динамической памяти

Можем ли мы тогда написать `strcat`, которая увеличивает буфер, если его не хватает? Как она могла бы работать?

РЕАЛЛОКАЦИИ

Чтобы немного расширить буфер в динамической памяти, делать `new` на новый размер и `delete` на старом месте может быть накладно

Здесь на помощь приходит `realloc`

```
void* realloc( void* ptr, std::size_t new_size );
```

Например,

```
int *arr = new int[100];
arr = static_cast<int *>(realloc(arr, 1000));
```

При нехватке памяти `realloc` вернет `nullptr`

Внимание: `realloc` не работает на стеке и в глобальной памяти

УПРАЖНЕНИЕ. CONCAT + REALLOC

Ваша задача написать функцию

```
char *strcat_r(char *dest, int bufsz, const char *src);
```

При нехватке места в старом буфере эта функция должна реаллоцировать память и возвращать новый указатель

Будьте очень внимательны с нулевым символом

УПРАЖНЕНИЕ. ЗАМЕНА В СТРОКЕ

Реализуйте функцию, осуществляющую замену всех подстрок в строке
Функция должна вернуть новую строку, аллоцированную в куче

```
char *replace(char *str, char const *from, char const *to);
```

Например,

```
char *s = new char[41];
strcpy(s, “Hello, %username!\n How are you, %username?”);
char *r = replace(s, “%username”, “Eric, the Blood Axe”);
```

Обратите внимание, что строка `from` может быть длиннее и короче, чем `to`

УПРАЖНЕНИЕ. ПОИСК CI ПОДСТРОКИ

Напишите простую функцию `strstrci`, которая ищет подстроку в строке, независимо от регистра символов (ci означает case insensitive)

```
char *strstrci(char const *haystack, char const *needle);
```

Ниже приведен пример применения

```
char const *needle = “Ab”; *src = “abracadaBra”;
char *pos1, *pos2, *pos3;
pos1 = strstrci(src, needle);       assert(pos1 != nullptr);
pos2 = strstrci(pos1 + 2, needle); assert(pos2 != nullptr);
pos3 = strstrci(pos2 + 2, needle); assert(pos3 == nullptr);
```

Первая найденная позиция “abracadaBra”, вторая “abracadaBra”

Оцените асимптотику наивного алгоритма

ЗАДАЧА АССЕМБЛИРОВАНИЯ

Простейший ассемблерный код состоит из мнемоник, которые работают с регистрами, константами и т.д.

```
add r0, r1, r2
or r0, r0, 14
sub r1, r0, 45
```

Представим, что нам надо считать массив мнемоник и их операндов

Эта задача является задачей **лексического анализа**.

ПРИМЕР: ЛЕКСИЧЕСКИЙ АНАЛИЗ

Нужно вывести выражение, содержащее мнемонику и операнды:

```
add r0, r1, r2
```

Казалось бы это несложно, но есть проблемы
Пробелы могут быть введены лишние или не введены вообще.

```
add r0,r1,r2
```

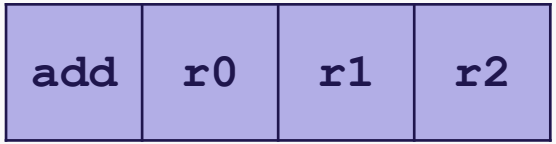
Нужна диагностика для ошибок

```
add r0,r1,
```

Как представить считанное выражение?

ЗАДАЧА ЛЕКСИЧЕСКОГО АНАЛИЗА

“ add r0, r1, r2”



- Как вы хотели бы хранить сведения о лексеме в памяти компьютера?
- Тип лексемы: оператор, скобка, число
- Вместе с оператором надо хранить тип операции, со скобкой – тип скобки, с числом – его значение.

ПРИМЕР: ЛЕКСИЧЕСКИЙ АНАЛИЗ

Нужно вывести выражение, содержащее мнемонику и операнды:

```
add r0, r1, r2
```

Казалось бы это несложно, но есть проблемы
Пробелы могут быть введены лишние или не введены вообще.

```
add r0,r1,r2
```

Нужна диагностика для ошибок

```
add r0,r1,
```

Как представить считанное выражение?

ОБЪЕДИНЕНИЯ

Решением проблемы являются объединения

Объединение очень похоже на структуру, только все поля лежат по одному адресу

```
struct S {
    int x;
    char c;
};
```


```
union U {
    int x;
    char x;
};
```

ОБЪЕДИНЕНИЯ

Брать из объединения лучше именно то, что туда положил

```
union IntChar {
    int x;
    char c;
};

union IntChar ic;

ic.c = 'a';
int y = ic.x; // type punning
```

И здесь удобно использовать перечисление

ОБЪЕДИНЕНИЯ

Здесь показана структура с объединением и перечислением

```
enum DTS {DT_DAY = 0, DT_TIME };

struct DT {
  enum DTS what;
  union DayOrTime {
    int day;
    time_t time;
  } u;
};
struct DT d1 = { .what = DT_DAY, .u.day = 42 }; // C style
DT d2 = { DT_DAY, 42}; // C++ compatible
```

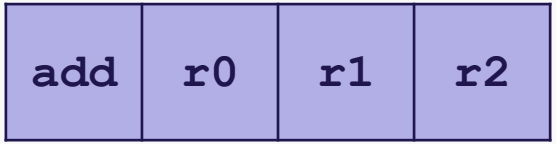
ИДЕЯ: МАССИВ ЛЕКСЕМ

Лексема – это операция, регистр или число

```
enum lexem_kind_t { OP, REG, NUM };
enum operation_t { ADD, SUB, MUL, DIV };

struct lexem_t {
    enum lexem_kind_t kind;
    union {
        enum operation_t op;
        int reg;
        int num;
    } lex;
};
```

“ add r0, r1, r2”



РЕКОМЕНДУЕМАЯ ЛИТЕРАТУРА

1. Бьерн Страуструп, Язык программирования C++/ ред. А. Боборыкин. – 4-е изд. - Москва: Издательство БИНОМ, 2023. – 1213 с.
2. Дональд Кнут Искусство программирования, том 1. Основные алгоритмы = The Art of Computer Programming, vol.1. Fundamental Algorithms. — 3-е изд. — М .: «Вильямс» , 2006. – 720 с.
3. Дональд Кнут Искусство программирования, том 2. Получисленные алгоритмы - The Art of Computer Programming, vol.2. Seminumerical Algorithms. — 3-е изд. — М .: «Вильямс» , 2007. — 832 с.
4. Кормен Т., Лейзерсон Ч., Ривест Р. Алгоритмы: построение и анализ. М.: МЦМНО, 1999. – 960 с.
5. Скотт Мейерс, Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2017. – 300 с.
6. Скотт Мейерс, Наиболее эффективное использование C++. 35 новых рекомендаций по улучшению ваших программ и проектов / ред. Д.А. Мовчан – 3-е изд. – Москва: ДМК Пресс, 2016. – 298 с.