Строки

Строка — это последовательность символов, записанная в памяти компьютера.

Строка представляется как массив кодов символов.

При отображении строки на экран используется **шрифт** — набор начертаний для каждого кода символа.

Кодировка — способ присвоения номеров символам. Также встречается название «кодовая страница».

7-битная кодировка

Носит название **ASCII** (American Standard Code for Information Interchange). Ориентируется на кодирование символов английского алфавита.

Является общей базой для большинства других кодировок.

Позволяет закодировать до 128 символов.

Первые 32 символа (с номерами 0..31) используются для служебных целей. Например, символы 10 и 13 кодируют переносы строк.

Символы с номерами 32..64 — это общие для всех языков символы пунктуации и цифры.

Номера 65..122 кодируют символы латинского алфавита.

Номера 123..127 — вспомогательные символы.

8-битные кодировки

Получаются расширением ASCII до 8 бит. Старший бит добавляет еще 128 значений.

Младшие 128 кодовых позиций (0..127) совпадают с ASCII.

Старшие 128 значений используются в разных целей в зависимости от задач кодировки.

- **CP-866**. Кодовая страница DOS для кодирования кириллицы.
- **Windows-1251**. Кодовая страница Windows для кодирования кириллицы.
- **KOI8-R**. Широко использовалась в Unix для кодирования кириллицы. При «срезании» старшего бита, русские символы превращаются в их английские фонетические аналоги, и текст остается читаемым.

Таблица Windows-1251

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0				©	Ё	§	Є	·		°						
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1																
	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
2		!	"	#	$	%	&		(	)	*	+				/
	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
6	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}]	~	□
	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
8	Ђ	Ѓ	,	ѓ	„	…	†	‡		‰	Љ	‹	Њ	Ќ	Ћ	Џ
	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
9	ђ	‘	’	"	"		–	—		™	љ	›	њ	ќ	ћ	џ
	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
A		Ў	ў	J	¤	Ґ	¦	§	Ё	©	Є	«		-	®	Ї
	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
B	°	±	І	і	ґ	µ		·	ё	№	є	»	j	S	s	ї
	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
C	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
D	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	▪ 223
E	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
F	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

8-битные кодировки

Очевидный недостаток 8-битных кодировок — ограниченное кодовое пространство.

Необходимо заранее знать кодировку текста, чтобы корректно отобразить его на экране. Обычно название кодировки записывается в том же файле латиницей.

Необходимо наличие нескольких версий одного и того же шрифта для разных кодировок.

Невозможно применение 8-битных кодировок для иероглифического письма.

Как правило, применение 8-битных кодировок оправдано только обратной совместимостью.

Unicode

Стандарт кодирования символов, используемый в подавляющем большинстве современных систем.

Каждому известному символу поставлен в соответствие уникальный номер. Обозначение номера символа U+NNNN, где NNNN — шестнадцатеричный номер.

Состоит из двух частей: Универсальный набор символов (Universal Character Set, UCS) и способ их двоичного представления (Unicode Transformation Format, UTF).

Обратно совместим с ASCII.

На сегодняшний день содержит порядка 150 тысяч символов, сгруппированных по назначению и национальной принадлежности.

Поддерживаются комбинированные символы, диакритические знаки, специальные символы форматирования.

UTF

Unicode Transformation Format, определяет способ записи символа с номером Unicode в виде байт.

- UTF-16. Произошел от первой версии Unicode, отводит 16 бит на один символ. При необходимости кодирования символа с номером старше U+FFFF, используется суррогатная пара из двух символов UTF-16. Диапазон значений: 1 112 064 кодовые позиции.
- UTF-32. Отводит 32 бита на 1 символ, гарантированно кодирует любую кодовую позицию Unicode без применения дополнительных средств. Является избыточным для хранения данных, так как старшие биты числа всегда равны 0 и не используются. Применяется для внутренних нужд текстовых процессоров.

UTF-8

UTF-8 является наиболее универсальным и экономичным форматом записи числа Unicode.

Применяется на 95% веб сайтов по состоянию на 2020 год.

Обратно совместим с ASCII.

В зависимости от номера символа, кодирует его последовательностью от 1 до 4 байт.

UTF-8

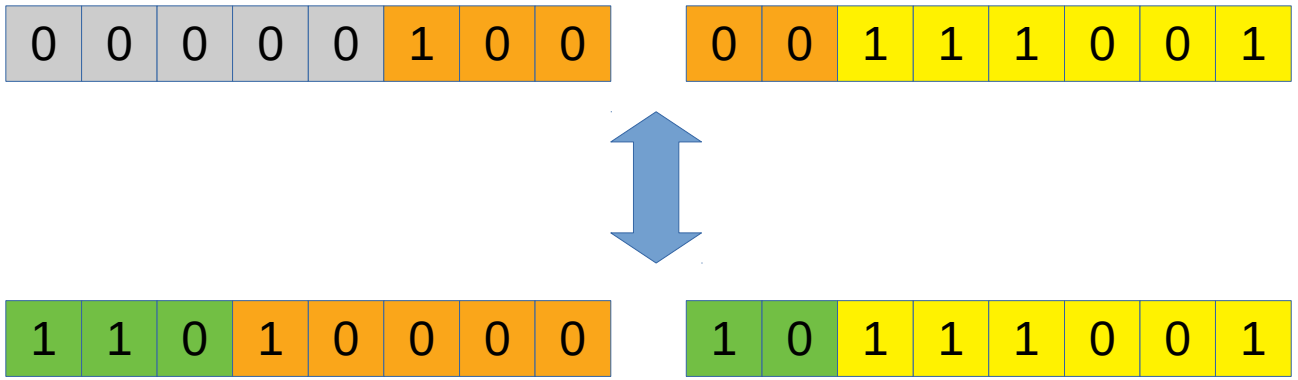
При записи символа в промежутке 0..127 (U+0000..U+007F), код символа записывается «как есть» в единственный байт. Этот режим полностью совместим с ASCII.

Символы за пределами 127-го кодируются в многобайтном виде. Старшие биты первого байта содержат количество единиц по количеству байт в символе, затем идет ноль и значащие биты. Старшие биты последующих байт содержат значение 10, затем идут 6 значащих бит.

Диапазон номеров	Байт	Значащих бит	Шаблон
U+0000..U+007F	1	7	0xxx xxxx
U+0080..U+07FF	2	11	110x xxxx 10xx xxxx
U+0800..U+FFFF	3	16	1110 xxxx 10xx xxxx 10xx xxxx
U+10000..U+10FFFF	4	21	1111 0xxx 10xx xxxx 10xx xxxx 10xx xxxx

UTF-8

Символ	Номер	Двоичный вид	Представление в UTF-8
4	U+0034	00110100	00110100
g	U+0067	01100111	01100111
Ф	U+0424	00000100 00100100	11010000 10100100
й	U+0439	00000100 00111001	11010000 10111001
€	U+20AC	00100000 10101100	11100010 10000010 10101100
∞	U+221e	00100010 00011110	11100010 10001000 10011110
☺	U+263A	00100110 00111010	11100010 10011000 10111010



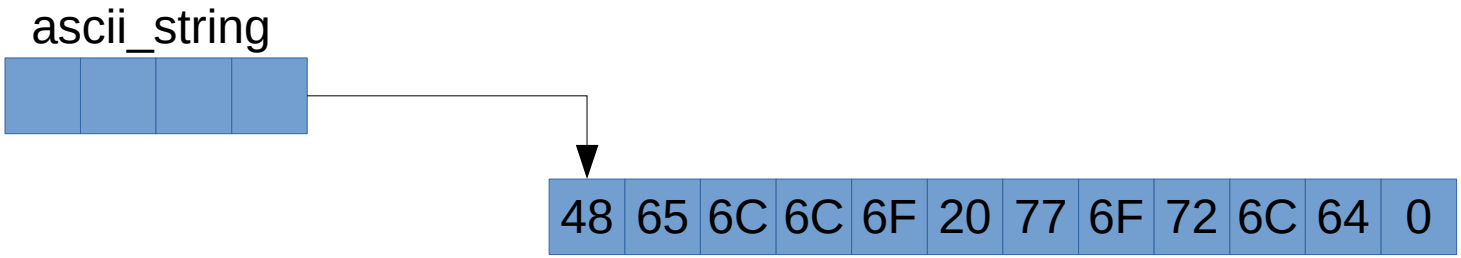
Строки С

Язык С хранит строку в виде массива из элементов **char**, который является знаковым целым 8-битным числом. Функции для работы со строками принимают в качестве строки указатель на первый элемент: тип **char***.

Указатель не несет информации о длине строки, поэтому окончанием строки считается символ с кодом 0.

Строковый литерал записывается в двойных кавычках. Он преобразуется при компиляции в константный массив и автоматически дополняется нулевым символом.

```
char* ascii_string = "Hello world";
```



Строки С

Со строкой можно работать как с обычным массивом. Отдельные символы строки соответствуют типу **char**.

```
char* ascii_string = "Hello world";
char spc = ascii_string[5];
```

Для работы с отдельными символами существует **символьный литерал**, записываемый в одинарных кавычках.

```
ascii_string[0] = 'h';  // "hello world";
```

Запись нуля в строку позволяет сделать ее короче. Символы после нуля игнорируются.

```
ascii_string[6] = 'a';  // "hello aorld";
ascii_string[7] = 'l';  // "hello alrld";
ascii_string[8] = 'l';  // "hello allld";
ascii_string[9] = 0;    // "hello all";
```

Строки C

Вспомогательные функции для работы со строками расположены в файле string.h:

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

- **strlen(char*)** — возвращает длину строки, не считая нулевого символа.
- **strcpy(char* dest, char* src)** — копирует символы из строки src в строку dest. Необходимо, чтобы в буфере dest было выделено достаточно места.
- **strcat(char* dest, char* src)** — добавляет символы из строки src в конец строки dest. Буфер dest должен быть достаточной длины.

Строки C

- **strcmp(char* s1, char* s2)** — производит посимвольное сравнение строк. Функция останавливается на первом различии, либо когда наткнется на 0. Возвращает <0, если первая строка меньше, >0, если первая строка больше, и 0, если строки равны.
- **strchr(char* s, int chr)** — ищет символ с номером chr в строке s. Возвращает указатель на найденный символ, либо NULL, если такого символа в строке нет.
- **strstr(char* s1, char* s2)** — ищет строку s2 внутри строки s1. Возвращает указатель на начало найденной строки, либо NULL, если такой строки нет.

Служебные символы

При обработке строки компилятором особым образом обрабатывается символ обратной косой черты (****). Этот символ, вместе со следующим за ним, преобразуется в один символ в конечном массиве по следующим правилам:

Последовательность	Результат	Пояснение
\0	00	Нулевой символ, окончание строки
\a	07	Вместо вывода данного символа воспроизводится звуковой сигнал
\b	08	Возврат назад на 1 символ
\t	09	Табуляция (широкий пробел)
\n	0A	Перевод строки
\r	0D	Возврат каретки

Любой другой символ вставляется без изменений. Например, \" заменяется на символ двойной кавычки. Для вставки в строку обратной косой черты следует записать \\.

Семейство printf

Стандартная библиотека C содержит в заголовочном файле stdio.h функции **printf** и **scanf** для обеспечения форматированного ввода-вывода с возможностью преобразования аргументов.

Существуют версии функций для ввода-вывода в строку (sprintf, sscanf) и в файл (fprintf, fscanf).

При обработке строки **printf** ищет символ %, затем анализирует записанное после него мнемоническое обозначение и выводит вместо данной последовательности значение аргумента.

Количество последовательностей вида %<...> должно совпадать с количеством аргументов, не считая строки-формата.

```
#include <stdio.h>
printf("Message: %s (%d bytes)\n",
       "Hello", strlen("Hello"));
// Message: Hello (5 bytes)
```

Семейство printf

Функция **scanf** производит обратную операцию: находит в введенной с клавиатуры строке последовательности символов, соответствующие формату, и переводит их в значения переменных. Переменные в scanf передаются в виде их адресов.

Количество последовательностей вида %<...> должно совпадать с количеством аргументов, не считая строки-формата. Возвращаемое значение — количество успешно прочитанных переменных.

```
#include <stdio.h>
char* msg = "Base has 4496 records";
int i;
int res = sscanf(msg, "Base has %d records", &i);
printf("%d (%d)\n", i, res);          // 4496 (1)
res = sscanf(msg, "Base %d", &i);
printf("%d (%d)\n", i, res);          // 4496 (0)
```

Формат printf

%[флаги][ширина][.точность][размер]тип

Тип записывается как один символ и определяет способ преобразования данных. Является обязательным.

Обозначение	Тип C	Пояснение
d, i, u	int	Целое число в десятичной системе
o	int	Целое число в восьмеричной системе
x, X	int	Целое число в шестнадцатеричной системе
f, F	double	Вещественное число в обычной форме записи
e, E	double	Вещественное число в экспоненциальной форме записи (например, 6.02E+23)
g, G	double	Вещественное число. Форма записи определяется автоматически.
s	char*	Строка
%		Последовательность %% используется для вывода символа %

Формат printf

%[флаги][ширина][.точность][размер]тип

Размер указывается как набор символов и используется при отклонении размера переменной от стандартного:

- %ld — использование long int вместо int
- %lld — использование long long int вместо int
- %lf — использование long double вместо double
- %hd — использование short int вместо int
- %hhd — использование signed char вместо int

```
char c;
sscanf("%hhd", "44", &c);
long long int li = 1LL << 32;
printf("%d", li);       // 0
printf("%lld", li);     // 4294967296
```

Формат printf

%[флаги][ширина][.точность][размер]тип

Ширина записывается как число и определяет минимальное количество символов, которое должно занимать выводимое значение. Может отсутствовать.

Точность записывается как число после точки. Для целых типов определяет минимальное количество выводимых символов, дополняя недостающие нулями. Для вещественных — количество символов после запятой. Для строк — максимальное количество выводимых символов. Может отсутствовать.

При отсечении символов используются правила округления.

```
printf("%4d:%4d\n", 171, -218);  //  171:-218
printf("%4d:%4d\n", 1, 0);       //    1:   0
printf("%.4d:%.4d\n", -2, 44);   // -0002:0044
printf("%8.3f\n", 3.1415926);    //    3.142
```

Формат printf

%[флаги][ширина][.точность][размер]тип

Флаги определяют дополнительные параметры вывода и могут отсутствовать

Символ флага	Назначение
-	Выравнивание значения по левому краю поля. По умолчанию выравнивание идет по правому краю.
+	Принудительный вывод знака числа.
<пробел>	Вывод пробела перед положительными числами.
0	Дополнение чисел нулями до ширины поля.

```
printf("Total: %-8d points\n", 22);
// Total: 22       points
printf("Coords: % 4d:% 4d\n", 171, -218);
// Coords:  171:-218
```

sprintf и sscanf

Функции **sprintf** и **sscanf** работают со строкой вместо терминала и могут использоваться для перевода числа в строку и обратно. Первым аргументом этих функций всегда идет массив char'ов, с которым осуществляется работа.

```
#include <stdio.h>

// Преобразование из числа в строку
double pi = 3.141592654;
char s_pi[128];
sprintf(s_pi, "%8.6f", pi);

// Преобразование из строки в число
const char* s_e = "2.71828";
float e;
sscanf(s_e, "%g", &e);
```

Строки Unicode в C

Для работы со строками Unicode предусмотрен тип **wchar_t**, соответствующий широкому символу.

Стандарт не определяет размер и реализацию wchar_t, поэтому он может быть разным в разных компиляторах и не должен использоваться для передачи строк между разными платформами.

Все функции для работы со строками продублированы для широких символов и доступны в заголовочном файле **wchar.h** с другим префиксом.

Для записи широкого строкового литерала необходимо использовать префикс **L**.

```
#include <wchar.h>
wchar_t* ws = L"Привет мир";
int i = wcslen(ws);      // wcslen вместо strlen
printf("%d\n", i);       // 10
```

Строки в C++

В языке C строки построены на работе с указателями, требуют ручного управления памятью. Это требует аккуратной работы и может приводить к трудноуловимым ошибкам.

Строки в C++ представлены классом **string**, который аналогичен контейнеру **vector<char>**, но дополнен функциями, упрощающими работу. Класс расположен в заголовочном файле **string** в пространстве имен **std**.

Строки **string** берут на себя функции по управлению памятью. При этом со string можно работать как с обычным массивом.

Объект string знает объем выделенной и фактически используемой под строку памяти. Запись нулевого символа не приводит к изменению фактической длины строки.

Для широких символов тот же заголовочный файл содержит класс **wstring**, оперирующий с символами **wchar_t**.

Строки в C++

```
#include <string>
#include <iostream>
#include <stdio.h>
using namespace std;

string s = "Hello world";
cout << s.size() << endl;                    // 11
s = "Test" + " " + "subject";
cout << s << endl;                           // Test subject
s.clear();
cout << s << endl;                           //
for (int a=0; a<10; ++a)
    s += to_string(a);
cous << s << endl;                           // 0123456789
cout << strlen(s.c_str()) « endl;            // 10
```

Строки в C++

```
#include <string>
#include <iostream>
#include <stdio.h>
using namespace std;

string s = "Test subject";
cout << s.find("e") << endl                    // 1
     << s.find("e", 2) « endl;                 // 9
s[4] = '-';
cout << s << endl;                             // Test-subject

string s2 = s.substr(0, 4);
cout << s2 << endl;                            // Test
if (s2 == "Test")
    cout << "test passed" << endl;
```

Потоки ввода-вывода в C++

Ввод и вывод данных в C++ происходит при помощи потоков ввода-вывода, а также операторов ввода в поток (<<) и вывода из потока (>>).

Операторы ввода-вывода автоматически преобразуют данные в строку и обратно.

Потоки ввода-вывода, соответствующие текстовому терминалу, объявлены в файле <iostream>.

```
#include <iostream>
#include <cmath>
using namespace std;

double x, y;
cout << "Input coords: ";
cin >> x >> y;
cout << "Module: " << sqrt(x*x + y*y) << endl;
```

Потоки ввода-вывода в C++

Заголовочный файл **<sstream>** содержит класс **stringstream**, осуществляющий ввод и вывод через строку. Этот класс можно использовать для преобразования между числами и строками.

Метод **str()** без аргументов позволяет получить строку, хранимую внутри потока, а с аргументом — заменить ее на новую.

```
#include <sstream>
#include <iostream>
using namespace std;

stringstream ss;
ss << 33.22;
string txt = ss.str();
ss.str("3.1416");
double d;
ss >> d;
cout << txt << ", " << d << endl; // 33.22, 3.1416
```

Форматирование потоков C++

Для форматирования потоков ввода-вывода используется заголовочный файл **<iomanip>**.

Для задания минимального количества символов при выводе следующего значения используется функция **setw**. Настройка setw сбрасывается после вывода значения.

```
#include <iostream>
#include <iomanip>
using namespace std;

cout << setw(10);
cout << 2.718281828 << endl;          //    2.71828
cout << 2.718281828 << endl;          // 2.71828
cout << setw(15);
cout << "Hello" << endl;              //           Hello
cout << "world!" << endl;             // world!
```

Форматирование потоков C++

Для задания точности при выводе следующего значения используется функция **setprecision**. Не влияет на вывод строк.

```
#include <iostream>
#include <iomanip>
using namespace std;

cout << setprecision(5);
cout << 2.718281828 << endl;          // 2.7183
cout << setprecision(10);
cout << 2.718281828 << endl;          // 2.718281828
```

Форматирование потоков C++

Для определения символа заполнения, совместно с setw, используется функция **setfill**. По умолчанию вывод дополняется до нужного размера пробелами.

```
#include <iostream>
#include <iomanip>
using namespace std;

cout << setw(10);
cout << 2.718281828 << endl;          //     2.71828
cout << setw(10) << setfill('0');
cout << 2.718281828 << endl;          // 0002.71828
cout << setw(10) << setfill('_');
cout << "Hello!" << endl;             // ____Hello!
```

Форматирование потоков C++

Константа **fixed** переводит вывод вещественных чисел в вариант с фиксированной точкой, а **scientific** — в экспоненциальный вид.

```
#include <iostream>
#include <iomanip>
using namespace std;

cout << fixed << 0.00271828 << endl;
// 0.002718
cout << scientific << 0.00271828 << endl;
// 2.718280e-03
```

Функция **setbase** позволяет задать систему счисления для вывода целых чисел: 8, 16 и 10 (по умолчанию).

```
cout << setbase(8);
cout << 271828 << endl;                  // 1022724
cout << setbase(16);
cout << 271828 << endl;                  // 425d4
```

Форматирование потоков C++

Константы **left** и **right** позволяют выравнивать значение в поле по левому или правому краю.

```
#include <iostream>
#include <iomanip>
using namespace std;

cout << left;
cout << "|" << setw(15) << "Hello!"  << "|" << endl;

// |Hello!          |

cout << right;
cout << "|" << setw(15) << "Hello!"  << "|" << endl;

// |          Hello!|
```

Строки Qt

Очевидная проблема строк C++ — проблемы с кодировкой (в случае string) или стандартизацией (в случае wstring).

Также необходимость в переводе данных между числами и строками порождает громоздкий код.

При разработке Qt был разработан собственный класс для хранения строк — **QString**.

QString хранит строку в виде массива символов **QChar**. Для хранения данных применяется кодировка UTF-16. Строки Qt поддерживают суррогатные пары для представления всего диапазона Unicode.

Класс QString содержит все необходимые методы для анализа строк (парсинг), для преобразования между строками и числами, а также для перевода символов в байтовый массив в нужной кодировке. Все классы Qt работают со строками QString.

Строки Qt

При работе со строками Qt рекомендуется перевести все файлы в кодировку UTF-8. В таком случае, можно пользоваться строковыми литералами с русскими буквами без дополнительных методов, а также напрямую передавать такие литералы в методы Qt, ожидающие на вход QString.

```
QString s("Строка с русскими буквами");
s = "Привет!"
ui->pushButton->setText("Я кнопка!");
```

Строку QString можно преобразовать к байтовому массиву QByteArray в необходимой кодировке и преобразовать его к строке C. Также можно преобразовать QString к std::string и std::wstring.

```
char* c_str      = s.toUtf8().data();
std::string ss   = s.toStdString();
std::wstring ws = s.toStdWString();
```

Строки Qt

Для преобразования числового значения в строку, можно использовать метод **QString::number**, конструирующий новую строку.

```
QString s = QString::number(4.22);   // "4.22"
```

Преобразование обратно в число происходит методами **toInt()**, **toLongLong()**, **toFloat()**, **toDouble()**.

Необязательный аргумент — указатель на bool, в который будет помещен флаг успешности операции.

```
QString s = "23.5611";
double d = s.toDouble();
s = "Hello";
bool b;
d = s.toDouble(&b);
if (!b)
    cout << "String to double conversion failed";
```

Строки Qt

Строки Qt поддерживают макроподстановку при помощи метода **arg**. Маркером для подстановки является строка вида %N, где N — натуральное число. Метод arg находит макрос с наименьшим числом и заменяет его на переданное значение. Существует множество полиморфных методов arg для разных типов.

```
QString s = QString("Coords: %1 x %2").arg(14).arg(-12);
int num = 0xBEEF;
QString s = QString("%1")
            .arg(0xBEEF, 8, 16, QChar('0'));
// 0000beef

s = QString("login: %2, password: %1")
            .arg("qwe").arg("111");
// login: 111, password: qwe
```

Строки Python

Строки в Python представлены типом **str**, доступным по умолчанию.

Строковый литерал Python может быть записан как в одинарных, так и в двойных кавычках. При этом двойные кавычки могут быть расположены внутри одинарных и наоборот.

```
s = "Hello world!"
print(type(s))          # <class 'str'>
```

Строки являются одним из базовых типов данных и работают как обычные переменные. Для строк доступны операторы сложения и умножения.

```
s = "Hello" + " " + "world!";   # "Hello world!"
s = "_" * 10                    # "__________"
```

Строки Python

Для строки доступен оператор квадратных скобок. При этом строка работает как тьюпл: можно выбирать из неё отдельные значения, но нельзя модифицировать. При необходимости изменить строку, необходимо создать новую строку на основе старой.

```
s = "Hello world!"
print(s[-1])                        # !
print(s[:4])                        # Hell
s = s[:5] + " cruel " + s[-6:]      # Hello cruel world!
```

Строки сравниваются посимвольно через операторы **<**, **>**, **==** и **!=** с учетом регистра символов.

```
"Иванов" < "Петров"                 # True
"сидоров" == "Сидоров"              # False
```

Поиск в строках Python

Для быстрой проверки, встречается ли указанная последовательность в строке, существует оператор **in**.

```
"ll" in "Hello world"    # True
"v"  in "Hello world"    # False
```

Для поиска в строке используется метод **find**. Он возвращает индекс символа, с которого начинается найденная последовательность, либо -1, если ничего не найдено. Второй необязательный аргумент — индекс символа, с которого следует начать поиск.

```
s = "Hello world"
s.find('o')              # 4
s.find('o', 5)           # 7
```

Кодировки в Python

Существует различие между строками в Python 2 и Python 3:

- В Python 2 тип **str** отводит 8 бит на символ строки и эквивалентен байтовому массиву **bytearray**. Для работы со строками в кодировке Unicode предусмотрен отдельный тип **unicode**. Его можно сконструировать, добавив **u** перед строковым литералом. Обычные строковые литералы восьмибитные.

```
s = u"Привет!"
```

- В Python 3 тип **str** работает в кодировке Unicode и эквивалентен типу **unicode**, оставленному для сместимости. Строковый литерал с префиксом **u** и без него эквивалентны. Для создания байтового массива **bytearray** необходимо добавить префикс **b** перед литералом.

```
s = b"This is threated as bytes"
```

Кодировки в Python

Строки Python поддерживают множество различных кодировок. Для преобразования из строки в байтовый массив в необходимой кодировке используется метод **encode**. Обратная операция делается методом **decode**. Аргумент — имя кодировки.

```
s = u"Привет!"
print(s.encode('utf-8'))
# b'\xd0\x9f\xd1\x80\xd0\xb8\xd0\xb2\xd0\xb5\xd1\x82'
s = b'\x0D\x0A'.decode('utf-8')
# "\r\n"
```

При невозможности представления байта в печатном виде при выводе строки на экран, Python заменяет его на последовательность вида \xNN, где NN — шестнадцатеричный код символа. Для записи символа через номер unicode используется форма \uNNNN, где N = шестнадцатеричный номер символа. Такие последовательности также работают при вводе с клавиатуры и в исходном коде.

Коды символов

Для получения номера символа используется встроенная функция **ord**. Аргумент функции — строка единичной длины.

```
ord('4')      # 52
ord('Щ')      # 1065
```

Обратное преобразование делается при помощи функции chr. Она принимает номер символа и возвращает строку единичной длины, содержащую этот символ.

```
chr(52)       # '4'
chr(1200)     # 'Ұ'
```

Служебные символы

Python поддерживает те же служебные последовательности, что и C, для записи непечатных символов.

Последовательность	Результат	Пояснение
\0	00	Нулевой символ, окончание строки
\a	07	Вместо вывода данного символа воспроизводится звуковой сигнал
\b	08	Возврат назад на 1 символ
\t	09	Табуляция (широкий пробел)
\n	0A	Перевод строки
\r	0D	Возврат каретки

В строке фактически хранится номер символа, но при выводе он заменяется на последовательность, если символ является непечатным. При обработке ввода последовательности обрабатываются особым образом, а всё остальное записывается как есть. Обратная косая черта записывается через \\.

Преобразования чисел и строк

Любой тип данных Python может быть преобразован к строке. Для этого надо вызвать конструктор **str** и передать ему нужное значение в качестве аргумента.

```
s = str(1287)              # '1287'
```

Число можно преобразовать в строку в разных системах счисления, используя функции **hex**, **oct** и **bin** для 16, 8 и 2 разрядов соответственно.

```
str(3217)       # '3217'
bin(3217)       # '0b110010010001'
hex(3217)       # '0xc91'
oct(3217)       #'0o6221'
```

Преобразования чисел и строк

Обратное преобразование из строки в число происходит путем вызова конструкторов **int** или **float**.

```
i = int('27')
f = float('3.22e11')
```

Дополнительный аргумент int — система счисления. Поддерживается любое число от 2 до 32.

```
int("1101", 2)    # 13
int("1101", 16)   # 4353
```

При невозможности преобразовать строку в число, возникает ошибка **ValueError**. Ее можно обработать через механизм исключений.

```
try:
    i = int('38 попугаев')
except ValueError:
    i = 0
```

Парсинг строк

Метод **strip** удаляет пробельные символы в начале и в конце строки.

```
s = "        a = 23 * math.sin(math.pi / 4)"
s.strip()    "a = 23 * math.sin(math.pi / 4)"
```

Метод **split** разбивает строку на части, используя переданную строку как разделитель.

```
s = "1, 2, 4, 6, 2"
ar = s.split(", ")    ["1", "2", "4", "6", "2"]
```

Метод **replace** создает новую строку, полученную из текущей путем замены всех встречающихся последовательностей.

```
text = "The big brown fox";
text = text.replace('big', 'small')
# "The small brown fox"
```

Форматированный вывод

Строки Python поддерживают макроподстановку аналогично функции **printf** в C. В процессе подстановки маркер последовательности, начинающийся с символа %, будет заменен на значение переменной. Значения переменных передаются как тьюпл, записанный через оператор % после строкового литерала.

```
output = "Coords: %d x %d" % (-4, 22)
print(output)                          # Coords: -4 x 22
```

Формат маркера незначительно отличается от формата, используемого в printf. Полный список настроек маркера доступен в документации Python.

Форматированный вывод

Более современный вариант — использование метода **format**. Маркер метода format — фигурные скобки.

```
output = "Coords: {} x {}".format(-4, 22)
print(output)                              # Coords: -4 x 22
```

Внутри фигурных скобок можно дополнительно настраивать параметры подстановки. Один из простых вариантов — изменение порядка подстановки аргументов.

```
output = "Bytes values: {2} {1} {0}"\
         .format(170, 187, 204)
print(output)
# Bytes values: 204 187 170
```

Полное руководство по format доступно в документации Python.