Особенности национального языка в программном коде

Олег Цилюрик

Редакция 31, от 01.02.2023

Оглавление

— Сейчас модно держать странных животных. Вон наш участковый Кирюхин поимел себе коалу и теперь эвкалипт высаживает на приусадебном участке. Коалы страсть как эвкалиптовые листья любят. Кирюхин пытался конечно эвкалиптовой настойкой пропитывать наши отечественные березовые листья, но тот ни в какую.

Кинофильм «Особенности национальной рыбалки»

От автора

Структура текста

Весь последующий текст состоит из 2-х частей, и разбит на подразделы. В первой части рассматриваются общие вопросы представления и техники работы с текстовой информацией, отличной от англоязычной. Сначала мы рассмотрим здесь вопросы локализации в языке C. Затем то же повторно будет рассмотрено на языке C++. Это классика … и POSIX API. И только затем вернёмся коротко к обзору современных (более поздних) языков программирования.

Основной упор далее будет сделан не на словесные описания, а на иллюстрации на примерах фрагментов кода, которые не нуждаются в особых пояснениях. Соответственно, этот материал не рассчитан на тех, кто первоначально изучает язык C (или C++, или любой другой), а предполагает уже достаточно обстоятельное знание языков.

Во второй части рассматриваются вопросы работы с регулярными выражениями в языке C. Совершенно естественно, что они в полной мере могут быть применимы и в C++. Затем повторно будет рассмотрена специфика исключительно C++. И далее, как и раньше — более современные языки.

В примерах здесь использованы только очень простые, вплоть до тривиальных, образцы регулярных выражений. Это сделано сознательно для упрощения, и объясняется это тем, что предметом нашего рассмотрения является не **составление** регулярных выражений и их **синтаксис**, а поведение самих регулярных выражений с Unicode строками, когда понятия символ и байт перестают быть тождественными.

Всё последующее изложение построено на стандартах POSIX и использовании исключительно операционной системы Linux. К Windows, из-за совершенно отличного там представления локализованных строк (Unicode, UTF-16), всё сказанное не относится **вообще** … кроме, разве что, самых общих фактов о структуре строчных данных.

Разметка и код

Цитируемые из сторонних источников фрагменты в тексте выделяются *курсивным шрифтом*. Протоколы выполнения команд и листинги программных кодов выделяются моноширинным шрифтом. В примерах выполнения команд, как это часто делается, вывод программы (системы) на терминал показывается обычным шрифтом, а пользовательский ввод с терминала пользователем — **жирным шрифтом**.

Архив всех представленных в тексте примеров кода (с прилагаемыми файлами протоколов сборки, изменений, выполнения, тестирования), чтобы не восстанавливать их из текста, может быть свободно скачан по ссылкам в блоге автора: http://mylinuxprog.blogspot.com/2016/09/cc.html.

В примерах, как это часто делается в публикациях, вывод программы (системы) на терминал показывается обычным шрифтом, а ввод с терминала пользователем — **жирным шрифтом**, иначе в показанных потоках вывода крайне сложно уследить, что является исходными сроками, а что результатами сопоставления с образцом.

Пара слов про авторские права

Проблемы локализации (вместо предисловия)

Вселенная – некоторые называют её Библиотекой – состоит из огромного, возможно, бесконечного числа шестигранных галерей, с широкими вентиляционными колодцами, ограждёнными невысокими перилами. Из каждого шестигранника видно два верхних и два нижних этажа – до бесконечности.

Хорхе Луис Борхес «Вавилонская Библиотека»

Предметом рассмотрения этих настоящих заметок являются вопросы использования **национальных** языков (русского, арабского, китайского … любых отличных от английского) в программном коде, главным образом для операционной системы Linux или подобных. Основное внимание будет уделено традиционным языкам C/C++ (классика жанра), но и беглому обзору состояния дел в современных, более новых языках программирования (Python, Go, Rust, Kotlin…) будет также бегло уделено некоторое внимание.

Как очень скоро будет показано и станет понятно, что после внедрения в IT-практику представлений (символов) Unicode, особенности использования самых разнообразных языков нивелировались и становятся единообразными (алфавитных или иероглифических систем, или даже систем специальных символов, таких как: математических или нотных символов). Важно чтобы такой набор символов (алфавит) отображался в кодировке Unicode! Поэтому во всём дальнейшем мы можем и будем везде употреблять термин **«национальный язык»** применительно не только к русскому языку, но расширенно — к **любому** языку: арабскому, китайскому, японскому …, более того, даже к языкам специальной нотации, например, к языку математических символов.

Вопрос использования национального языка в программном коде имеет несколько аспектов, «многослойный», как минимум это (а может и больше):

1. Использование таких языков программирования, в **синтаксисе** которого ключевые и зарезервированные слова допускаются (или предусмотрены **только** в таком виде) на национальном языке.

2. Допускает ли язык программирования **присваивание** переменным (и любым другим объектам программного кода) **имён**, записываемых на национальных языках.

3. Как производится **представление**, **создание, хранение** и **ввод/вывод** текстовых **литералов** (строчных **констант**) на национальных языках. Представление текстовых литералов может реализовываться либо «доморощенными» способами (поток байт: Pascal, ранний C), или использованием Unicode (в той или иной кодировке: UTF-8, UTF-16, UTF-32) — это не имеет значения.

4. Как производятся **манипуляции с содержимым** текстовых строк на национальных языках: поиск, замена, перестановки и другие **операции**. Может показаться, что это то же самое что и предыдущий пункт, но, как мы увидим вскоре, это принципиально разные вещи.

Все эти, а возможно, и другие стороны вопроса — **это всё совсем не одно и то же!** В любом языке программирования, или его среде, инструментарии (библиотеки, пакеты, модули и др.), могут реализовываться одни пункты, и вовсе не обеспечиваться другие.

Чтобы больше не обращаться к этому, остановимся очень коротко на п.1 этого перечисления. Это вовсе не курьёз и совсем не такое пустое начинание: на раннем этапе развития того, что позже стали обозначать как IT, в СССР разрабатывался **целый ряд** языков программирования с русскоязычным синтаксисом ключевых слов-операторов. Причём таких языков программирования, фундаментальные идеи которых потом повлияли на всё развития языков программирования в последующие 40-50 лет. Поэтому стоило бы оглянуться, и коротко их хотя бы назвать:

- Язык Рефал. Первая версия Рефала была создана в 1966 году Валентином Турчиным. Это единственный язык из этого перечисления получивший мировую известность.

- 1968г. и далее: Институтом кибернетики Академии наук Украинской ССР, под рук. акад.

Глушкова В.М. разработаны и производились вычислительные машины МИР и МИР-2 (МИР - это не претензии на Universe, а просто **М**ашина **И**нженерных **Р**асчётов), работающие по программам на языке Алмир/Аналитик. Там оператор цикла мог выглядеть как-то так:

```
ДЛЯ Ж=1 ШАГ 0.3 ДО π ВЫПОЛНИТЬ ...
```

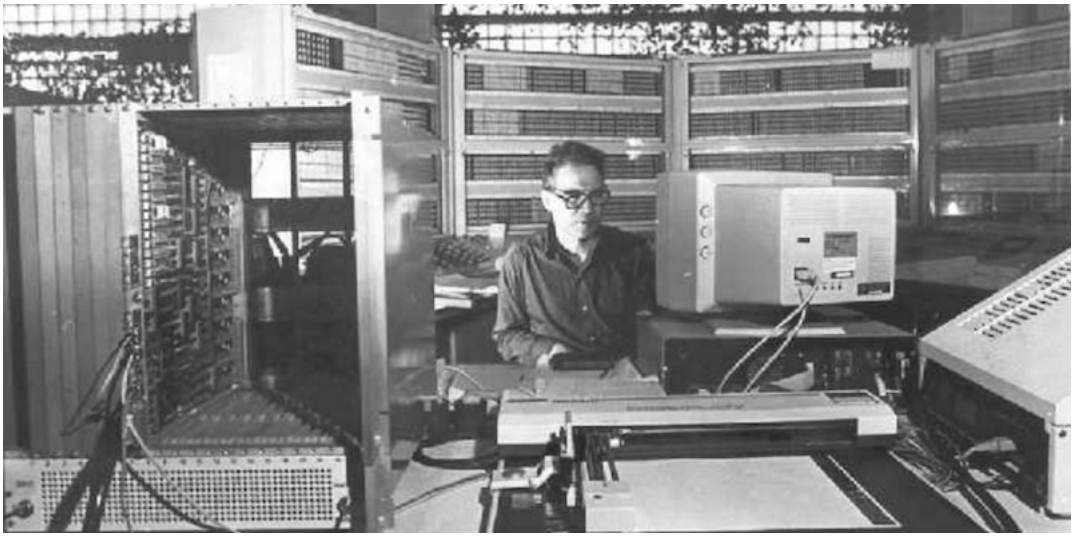
- Новосибирск, 1970 - 1981г.г., под руководством акад. Ершов А.П. создаётся обучающая система «Школьница» и языка Рапира, для там же созданного компьютера «Агат». Пример программы «Здравствуй, мир!»:

```
ПРОЦ СТАРТ();
    ВЫВОД: "ЗДРАВСТВУЙ, МИР!";
КНЦ;
```

Акад. Ершов А.П ещё в начале 80-х предполагал использовать лёгкость изучения языка Рапира для достижения **всеобщей компьютерной грамотности** (в 1980-м году!).

- Система программирования Бета: Изначально этот язык был назван — автокод Эльбрус, затем был переименован в Эль-76.

- Язык Сигма — название неожиданно очень удачно стало соответствовать сути разработанного языка, которую можно описать как «Символьный Генератор и Макроассемблер». Всего в истории языка Сигма было три его реализации: на М-20, на БЭСМ-6 и на самом языке Сигма.



И даже это ещё далеко не все...

Но русскоязычный **синтаксис** языка ничего принципиально не добавляет к **семантике** языка… На ранних этапах становления понимания семантики языков программирования это было вполне естественно. Язык программирования должен объединять разноязыких разработчиков, а не разъединять по какому-то ни было признаку. И уже к середине 80-х годов это было **понято**, и движение именно в этом направлении прекратилось… Так на этом мы и закончим краткий экскурс в 1-й из перечисленных аспектом поддержки национальных языков в программном коде. И к нему далее не станем обращаться…

Краткое замечание о соотношении пунктов 2 и 3 перечислений: константное представление и манипулирование содержимым строчных данных в коде. До тех пор, пока понятия символ и байт в текстовом представлении были **эквивалентными**, понятия хранения и оперирования с такими строками были тождественными. Но как только представление каждого символа в Unicode стало возможным представлять 1, 2, 3, 4 байтами (а потенциально до 6) — эта тождественность разрушается: если мы попытаемся **заменить** 3-байтовый символ в строке на 1-байтовый, то мы тут же разрушим всю структуру текстовой строки (образуются зависшие «остатки» в виде 2-х байт). И для корректного выполнения разнообразных манипуляций с контекстом строки нужно находить адекватные методы. Об этом мы подробно станем говорить ниже...

Литература и сетевые ресурсы

1. Разработка языков программирования и компиляторов в СССР

   https://habr.com/ru/company/ua-hosting/blog/273665/

2. Виктор Михайлович Глушков. Опережая время. 16 марта 2017.

   https://habr.com/ru/company/ua-hosting/blog/370259/

3. Ершов, Андрей Петрович

   https://ru.wikipedia.org/wiki/Ершов,_Андрей_Петрович

Часть 1. Локализация в Linux

Воспользуемся формулировками пусть и из самых поверхностных источников, Википедии (https://ru.wikipedia.org/wiki/Локализация_программного_обеспечения):

Локализация программного обеспечения — процесс адаптации программного обеспечения к культуре какой-либо страны. Как частность — перевод пользовательского интерфейса, документации и сопутствующих файлов программного обеспечения с одного языка на другой.

Для локализации в английском языке иногда применяют сокращение «L10n», где буквы «L» и «n» — начало и окончание слова Localization, а число 10 — количество букв между ними.

Во всех современных дистрибутивах Linux на сегодня по умолчанию используются локали, построенные на UTF-8 кодировании символьных представлений Unicode, например:

```
$ locale
LANG=ru_UA.UTF-8
LANGUAGE=ru_UA:ru
LC_CTYPE="ru_UA.UTF-8"
LC_NUMERIC=ru_UA.UTF-8
LC_TIME=ru_UA.UTF-8
LC_COLLATE="ru_UA.UTF-8"
LC_MONETARY=ru_UA.UTF-8
LC_MESSAGES="ru_UA.UTF-8"
LC_PAPER=ru_UA.UTF-8
LC_NAME=ru_UA.UTF-8
LC_ADDRESS=ru_UA.UTF-8
LC_TELEPHONE=ru_UA.UTF-8
LC_MEASUREMENT=ru_UA.UTF-8
LC_IDENTIFICATION=ru_UA.UTF-8
LC_ALL=
```

Использование Unicode уже гарантировано не требует каких-то отдельных кодовых страниц для различных национальных языков … как это требовалось в MS-DOS или Windows. Как видно из показанного листинга, локализация в общем виде предусматривает много аспектов: национальные представления денежных единиц, числовых величин, форматов даты и времени и др. … Нас в дальнейшем рассмотрении будут интересовать **только** некоторые аспекты, главным образом связанные с внутренним представлением **текстовой информации** в программе, и вопросы её ввода и вывода на периферийные устройства[1]. Потому и текст посвящён «национальным языкам в программном коде», а не «локализации в операционной системе» … кроме отдельных самых кратких аспектов системной локализации

Например относительно устранения избыточных локалей :

```
$ lsb_release -a
No LSB modules are available.
Distributor ID:       Linuxmint
Description:   Linux Mint 21
Release:       21
Codename:      vanessa
$ locale -a | wc -l
```

1 То, что в эпоху MS-DOS называлось «руссификация», и над чем тогда много копий было сломано...

```
28
$ locale -a | head
C
C.UTF-8
en_AG
en_AG.utf8
en_AU.utf8
en_BW.utf8
en_CA.utf8
en_DK.utf8
en_GB.utf8
en_HK.utf8
```

Или то же в Fedora:

```
$ lsb_release -a
LSB Version:   :core-4.1-amd64:core-4.1-noarch
Distributor ID:       Fedora
Description:   Fedora release 35 (Thirty Five)
Release:       35
Codename:      ThirtyFive
$ locale -a | wc -l
869
$ locale -a | head
aa_DJ
aa_DJ.iso88591
aa_DJ.utf8
aa_ER
aa_ER@saaho
aa_ER.utf8
aa_ER.utf8@saaho
aa_ET
aa_ET.utf8
af_ZA
```

Для инсталляционных дистрибутивов Linux присутствие всех возможных локалей — абсолютно **обязательно**, это понятно. Но весь этот избыток локалей:

- абсолютно избыточен в малых или встраиваемых конечных реализациях;
- даже в десктопных рабочих станциях избыток локалей требует заметных временных издержек при регулярных обновлениях инсталляции проводимых по сети;

В своей конкретной инсталляции вполне возможно оставить только минимум локалей необходимых в вашем языковом окружении, например подобными командами:

```
$ sudo localedef --delete-from-archive en_ZA.utf8 en_ZM en_ZM.utf8 en_ZW.utf8
```

И, в конечном итоге, последовательностью таких удалений можно свести весь набор к:

```
$ locale -a
C
C.UTF-8
en_GB.utf8
en_US.utf8
POSIX
ru_RU.utf8
ru_UA.utf8
```

Интернационализация

Существует более обобщенное понятие: интернационализация, подразумевающее проектирование и реализацию программного продукта и документации таким образом, который максимально упростит локализацию приложения. Не вникая в детали, возьмём на заметку, что одним из

основных элементов (среди других) техники интернационализации является возможность последующей загрузки локализированных элементов **в будущем** при желании пользователя[2] (даже если они отсутствуют на момент разработки).

Мы не будем дальше обращаться к этой технике, потому что это предмет совершенно других интересов. Но интернационализация программных проектов в целом не отменяет проблемы локализации, рассматриваемые далее. Программный код может получать потоки текстовой информации из внешних, относительно самого приложения, источников, с не прогнозируемым содержанием, и должен корректно работать с любым получаемым контекстом.

И если локализации посвящены репозиторные пакеты l10n, то интернационализации — пакеты i18:

```
$ aptitude search l10n | wc -l
239
$ aptitude search i18 | wc -l
70
```

Но ни то, ни другое, не есть предметом нашего интереса, и мы заниматься здесь этим не будем.

Символьные строки

> *В данный момент я лишь провожу инвентаризацию ... механически выстраиваю эти детали в ряд. Но это вполне стоящее занятие – постепенно, мало-помалу соединять реальность в единое целое. Так от трения камней или кусочков дерева друг о друга в конце концов выделяется тепло и появляется огонь. Это похоже на то, как из набора на первый взгляд бессмысленных, однообразно повторяющихся раз за разом звуков, складываются слоги…*
>
> *Харуки Мураками «Хроники Заводной Птицы».*

Здесь мы переходим к конкретным вопросам локализации текстовых строк в базовых и традиционных языках программирования C (как основа API Linux) и C++ (как наследник C). К состоянию дел в других современных языках программирования (Python, Go, Rust, Kotlin …) мы ещё вернёмся в конце текста.

Представление текстовой информации

> *Мир, в который вы собираетесь вступить, не имеет хорошей репутации.*
>
> *Иосиф Бродский, речь перед выпускниками Мичиганского университета.*

До некоторого времени (до начала 80-х) представление символьной информации базировалось, главным образом, на 7-битовом кодировании каждого символа (ASCII). Такая кодировка предполагала представление только основных латинских символов, цифровых символов и символов пунктуации (точка, запятая, дефис и т.д.). Если нужно было перейти на другую кодировку (тоже 7-бит), то **в потоке** байт-символов вставлялся символ перехода на другую кодировку ('\17' для русских символов), а при возврате в исходную кодировку — символ возврата ('\18'). Такая техника представления символов использовалась, например, в майнфреймах IBM (IBM-360, EC-1020), или мини-компьютерах DEC (LSI-11, PDP-11, «Электроника 60», «Электроника 79»).

Позже (в IBM PC и MS-DOS) были введены 8-битовое (расширенное) кодирование символов и кодовые страницы (исторически термин code page был введён корпорацией IBM). В таком варианте первая половина каждой кодовой таблицы (коды 0-127) как и раньше представляла ASCII набор символов (латинский алфавит), а вторая половина (код 128-255) — заполнялась алфавитом того или иного национального алфавита (имеющих алфавитные системы письма). Так, для конкретики, основная таблица, используемая для русского языка в MS DOS — CP866. В Windows для русского языка используется таблица CP1251 (но могут быть и другие, например ISO 8859-5 и т.д.). В ранних Linux (и других UNIX) в качестве русскоязычной кодовой страницы использовалась KOI-8R (но могут

2 Термин интернационализации не обязывает переводить текст программ или документацию на другой язык, он подразумевает разработку приложений таким образом, который сделает локализацию максимально простой и удобной, а также позволит избежать проблем при интеграции продукта для стран с отличающимися культурными традициями.

быть и другие). Всё, связанное с использованием кодовых страниц, мы не будем затрагивать в дальнейшем рассмотрении, как устаревший и отживший подход.

К началу 90-х годов всё расширяющееся число кодовых таблиц и порождаемая ними путаница всех безумно достали — число таблиц становится неподъёмным, а многие из них вообще практически не находят использования (так, например, кириллическая кодовая таблица ISO 8859-5 **никогда** не использовалась в русскоговорящих странах, но её упорно использовали зарубежные производители для «русской локализации»)… Кроме того, кодовые таблицы не позволяли покрыть языки с не алфавитной системой письма. В итоге, в 1991 году был предложен стандарт представления Unicode. Первый стандарт выпущен в 1991 году, последний — в 2016 (8.0.0), следующий ожидался летом 2017 года. Коды в стандарте Юникод разделены на несколько областей (страниц). Область с кодами от U+0000 до U+007F содержит символы набора ASCII с их соответствующими кодами. Под символы кириллицы выделены области знаков с кодами от U+0400 до U+052F, от U+2DE0 до U+2DFF, и от U+A640 до U+A69F.

В таблицы Unicode любой символ (будь то английского или китайского языка) выражается 32-битным значением. И это и есть тип `wchar_t`, который имеет в UNIX/Linux размер 4 байта (не путать с Windows, где `wchar_t` — это 2 байта, 16 бит). Но таблицы Unicode — это абстракция. А для представления этих значений нужно их как-то **кодировать**. И для этого предложены несколько систем кодирования: UTF-32[3] (это в чистом виде значения Unicode и `wchar_t` POSIX), UTF-16 (и `wchar_t` Windows ... но это нам не интересно) и UTF-8 (байтное кодирование **переменной длины**, которое было придумано всё теми же хорошо известными Кеном Томпсоном и Робом Пайком в 1992 году для ОС Plan 9). Любой существующий символ кодируется в UTF-8 последовательностью **от 1-го до 6-ти** последовательных байт (типа `char`). Символы **русского** языка (кирилица) отображаются в UTF-8 как **2 байта** на символ, но это не следует рассматривать как твёрдое правило или константу для всей строки: в едином потоке могут содержаться и латинские символы (1 байт на символ) и специальные, диакритические или другие символы (3-4 байт на символ)[4].

Кодирование UTF-8

Прежде чем рассматривать мультибайтные строки в коде, хорошо бы рассмотреть детально байтовую структуру кодируемых в UTF-8 символов, а для этого подготовить некоторые тестовые наборы строк на разных национальных языках. Несмотря на то, что нам желательно записать строки на самых замысловатых языках (алфавитных и иероглифических, с записью слева-направо и справа-налево) именно в Linux это сделать относительно несложно (из-за того, что **любая** текстовая информация в Linux представляется в UTF-8). Для этого можно, например, в переводчика Google заказывать фразу на перевод на нужный язык, а затем, не понимая в том что записано как результат, скопировать, в любом текстовом редакторе, этот результат в файл. В итоге у меня получился тестовый файл:

```
$ cat mult.dat
1 Здравствуйте    |
2 السلام عليكم     |
3 Dobrý den                  |
4 Hello                       |
5 שָׁלוֹם              |
6 नमस ्ते ं          |
7 こんにちは             |
8 今日は
9 안녕하세요             |
A 你好                        |
B Olá                         |
C Hola                        |
```

Здесь каждая строка начинается с 1-символьного номера, чтобы на неё можно было ссылаться на строку … другие небольшие странности формата тестовых строк будет объяснена очень скоро. Здесь каждая строка представляет примерно одно и то же на разных языках:

3 Стандарт Unicode состоит из двух основных разделов: универсальный набор символов (UCS, **U**niversal **C**haracter **S**et) и семейство кодировок (UTF, **U**nicode **T**ransformation **F**ormat). Универсальный набор символов задаёт однозначное соответствие символов кодам — элементам кодового пространства, представляющим неотрицательные целые числа (32-бит разрядности). Семейство кодировок определяет машинное представление последовательности кодов UCS.

4 Значения (uint32_t), закодированные в UTF-8, в принципе, могут быть длиной до 6 байт, однако стандарт Unicode не определяет символов выше 0x10ffff, поэтому символы Unicode могут иметь максимальный размер в 4 байта в UTF-8.

1. Русский
2. السلام عليك «Салам алейкум» — с арабского переводится как «мир вам» или «мир с вами».
3. С чешского Dobrý den : добрый день, здравствуйте, доброе утро
4. Английский
5. Шалом **שָׁלוֹם** : слово на иврите, означающее «мир», традиционное еврейское приветствие
6. Перевод с санкрита नमस ◌ृत ◌े : доброе утро, добрый день, здорово
7. Японское приветствие (konnichiwa): "приветствую вас", иероглифами на Хирагане: **こんにちは**
8. Японское приветствие (konnichiwa), иероглифами Катакана: **今日は**
9. Перевод корейского **안녕하세요** на русский: добрый день, доброе утро, добрый вечер
A. Китайский, **你好** перевод на русский: привет, добрый день, здравствуйте
B. Испанский - Olá : самое распространенное и общепринятое приветствие.
C. Испанский - Hola : привет

Первый и простейший способ взглянуть на внутренне представление этих строк — это стандартная утилита `hexdump` в байтовом отображении (не самый комфортный способ, но иногда вполне достаточный):

```
$ hexdump -C mult.dat
00000000  31 20 d0 97 d0 b4 d1 80  d0 b0 d0 b2 d1 81 d1 82  |1 ..............|
00000010  d0 b2 d1 83 d0 b9 d1 82  d0 b5 20 20 20 20 7c 0a  |..........    |.|
00000020  32 20 d8 a7 d9 84 d8 b3  d9 84 d8 a7 d9 85 20 d8  |2 ............ .|
00000030  b9 d9 84 d9 8a d9 83 d9  85 20 20 20 20 20 7c 0a  |.........     |.|
00000040  33 20 44 6f 62 72 c3 bd  20 64 65 6e 20 20 20 20  |3 Dobr.. den    |
00000050  20 20 20 20 20 20 20 20  20 20 20 20 20 20 7c 0a  |              |.|
00000060  34 20 48 65 6c 6c 6f 20  20 20 20 20 20 20 20 20  |4 Hello         |
00000070  20 20 20 20 20 20 20 20  20 20 20 20 20 20 7c 0a  |              |.|
00000080  35 20 d7 a9 d6 b8 d7 81  d7 9c d7 95 d6 b9 d7 9d  |5 ..............|
00000090  20 20 20 20 20 20 20 20  20 20 20 20 20 20 7c 0a  |              |.|
000000a0  36 20 e0 a4 a8 e0 a4 ae  e0 a4 b8 e0 a5 8d e0 a4  |6 ..............|
000000b0  a4 e0 a5 87 20 20 20 20  20 20 20 20 20 20 7c 0a  |....          |.|
000000c0  37 20 e3 81 93 e3 82 93  e3 81 ab e3 81 a1 e3 81  |7 ..............|
000000d0  af 20 20 20 20 20 20 20  20 20 20 20 20 20 7c 0a  |.             |.|
000000e0  38 20 e4 bb 8a e6 97 a5  e3 81 af 20 20 20 20 20  |8 .........     |
000000f0  20 20 20 20 20 20 20 20  20 20 20 20 20 20 7c 0a  |              |.|
00000100  39 20 ec 95 88 eb 85 95  ed 95 98 ec 84 b8 ec 9a  |9 ..............|
00000110  94 20 20 20 20 20 20 20  20 20 20 20 20 20 7c 0a  |.             |.|
00000120  41 20 e4 bd a0 e5 a5 bd  20 20 20 20 20 20 20 20  |A ......        |
00000130  20 20 20 20 20 20 20 20  20 20 20 20 20 20 7c 0a  |              |.|
00000140  42 20 4f 6c c3 a1 20 20  20 20 20 20 20 20 20 20  |B Ol..          |
00000150  20 20 20 20 20 20 20 20  20 20 20 20 20 20 7c 0a  |              |.|
00000160  43 20 48 6f 6c 61 20 20  20 20 20 20 20 20 20 20  |C Hola          |
00000170  20 20 20 20 20 20 20 20  20 20 20 20 20 20 7c 0a  |              |.|
00000180
```

(Здесь становится понятны и некоторые странности моего тестового файла: длина дополняется пробелами так, чтобы каждая строка в символьном дампе, справа, начиналась с новой строки, а символ ограничитель '|' — любой отчётливо различимый в дампе ограничитель строки).

Таким инструментом анализировать можно, но неудобно, поэтому была сделана небольшая программ несколько более специализированного свойства:

```
$ cat ss1.c
#include <stdio.h>
#include <string.h>
#include <stdint.h>
#include <limits.h>
#include <stdlib.h>
#include <wchar.h>
#include <locale.h>

inline void c2w(char *c, wchar_t *w) {
   int n = -1;
   setlocale(LC_ALL, "");              // только после этого работают преобразования!
   while (n != 0)
```

```
      c += (n = mbtowc(w++, c, MB_CUR_MAX));
}

void show_str(char* pstr) {
   wchar_t wbuf [strlen(pstr) * 2];
   char *pb = pstr + strlen(pstr) - 1;
   while(*pb == ' ') *pb-- = '\0';     // удаление хвостовых пробелов
   c2w(pstr, wbuf);
   printf("%s [%ld bytes:%ld symbols]\n",
          pstr, strlen(pstr), wcslen(wbuf));
   do
      printf("<%02X>", (uint8_t)*pstr++);
   while (*pstr != '\0');
   printf("\n");
}

int main( int argc, char **argv ) {
   char buf[4096], *pb;
   if (argc > 1) {
      strcpy(buf, argv[1]);
      show_str(buf);
   }
   else
      while (1) {
         if (NULL == fgets(buf, sizeof(buf) - 1, stdin)) break;
         if (1 == strlen(buf)) break;  // пустая строка
         if (index(buf, '\n') != NULL) *index(buf, '\n') = '\0';
         if (index(buf, '|') != NULL) *index(buf, '|') = '\0';
         pb = buf;
         if (' ' == *(pb + 1)) pb +=2; // убрать нумерацию строк
         show_str(pb);
      }
   return 0;
}
```

```
$ gcc -Wall -O2 -o ss1  ss1.c
```

Программа позволяет рассмотреть как строку заданную в командной строке:

```
$ ./ss1 今日は
今日は [9 bytes:3 symbols]
<E4><BB><8A><E6><97><A5><E3><81><AF>
```

Так и в диалоге с терминала:

```
$ ./ss1
你好
你好 [6 bytes:2 symbols]
<E4><BD><A0><E5><A5><BD>
नमस्ते
नमस्ते [18 bytes:6 symbols]
<E0><A4><A8><E0><A4><AE><E0><A4><B8><E0><A5><8D><E0><A4><A4><E0><A5><87>
```

Или символы из алфавитов специального назначения — символы валют:

```
$ ./ss1 €£₴₽
€£₴₽ [12 bytes:4 symbols]
<E2><82><AC><E2><82><A4><E2><82><B4><E2><82><BD>
```

А если программа может получать строки из `sysin`, то таким образом может получать строки и из любого текстового файла:

```
$ cat mult.dat | ./ss1
Здравствуйте [24 bytes:12 symbols]
<D0><97><D0><B4><D1><80><D0><B0><D0><B2><D1><81><D1><82><D0><B2><D1><83><D0><B9><D1><82><D0><B5
>
السلام عليكم [23 bytes:12 symbols]
```

```
<D8><A7><D9><84><D8><B3><D9><84><D8><A7><D9><85><20><D8><B9><D9><84><D9><8A><D9><83><D9><85>
Dobrý den [10 bytes:9 symbols]
<44><6F><62><72><C3><BD><20><64><65><6E>
Hello [5 bytes:5 symbols]
<48><65><6C><6C><6F>
שָׁלוֹם [14 bytes:7 symbols]
<D7><A9><D6><B8><D7><81><D7><9C><D7><95><D6><B9><D7><9D>
नमस ्ते ँ [18 bytes:6 symbols]
<E0><A4><A8><E0><A4><AE><E0><A4><B8><E0><A5><8D><E0><A4><A4><E0><A5><87>
こんにちは [15 bytes:5 symbols]
<E3><81><93><E3><82><93><E3><81><AB><E3><81><A1><E3><81><AF>
今日は [9 bytes:3 symbols]
<E4><BB><8A><E6><97><A5><E3><81><AF>
안녕하세요 [15 bytes:5 symbols]
<EC><95><88><EB><85><95><ED><95><98><EC><84><B8><EC><9A><94>
你好 [6 bytes:2 symbols]
<E4><BD><A0><E5><A5><BD>
Olá [4 bytes:3 symbols]
<4F><6C><C3><A1>
Hola [4 bytes:4 symbols]
<48><6F><6C><61>
```

Программа отчётливо разделяет такие понятия как **символ** в строке и отдельные **байты** в представлении этих символов. Число байт и символов в строке отчётливо не совпадают! В моём тестовом примере мы наблюдаем символы, которые представляются 1-м, 2-мя и 3-мя байтами.

Ещё один простой и быстрый способ декодировать любую строку на национальном языке в последовательность байт — это использование Python (версии 3), за счёт того, что Python по спецификации языка использует кодировку UTF-8. В диалоговом режиме это может выглядеть так:

```
$ python
Python 3.10.6 (main, Nov 14 2022, 16:10:14) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 'Здравствуйте'.encode()
b'\xd0\x97\xd0\xb4\xd1\x80\xd0\xb0\xd0\xb2\xd1\x81\xd1\x82\xd0\xb2\xd1\x83\xd0\xb9\xd1\x82\xd0\
xb5'
...
```

И символы алфавитов специального предназначения — например, а). музыкальные символы и б). специальные математические символы и знаки:

```
...
>>> '𝄞𝅗𝅥𝅘𝅥𝅯𝅘𝅥𝅰𝅘𝅥𝅱'.encode()
b'\xf0\x9d\x84\x9e\xf0\x9d\x84\x9a\xf0\x9d\x85\x9e\xf0\x9d\x84\x9a\xf0\x9d\x85\xa1\xf0\x9d\x84\
x9a\xf0\x9d\x84\x9a\xf0\x9d\x84\x9a'
>>> '𝔐ΞΔ∀∃∰'.encode()
b'\xf0\x9d\x94\x90\xf0\x9d\x9b\xaf\xf0\x9d\x9c\x9f\xe2\x88\x80\xe2\x88\x83\xe2\x88\xaf'
>>>
...
```

Но про использование Python мы поговорим отдельно позже.

Локализация строк в коде C/C++

Язык C и локализация

С тех пор, как в 1969—1973 годах язык C был разработан Деннисом Ритчи с коллегами, он остаётся неизменно и успешно используемым. Главной причиной такого долголетия[5] является, несомненно, то, что C является базовым языком написания операционных систем (для чего он, собственно, и был придуман) семейства UNIX (POSIX совместимых), и в частности Linux. И до тех пор, пока будет

5 Все другие языки программирования тех периодов (Algol, FORTRAN, COBOL, PL/1 …) практически полностью ушли из практического использования, или используются только в узко ограниченных областях применения, определяемыми традициями их использования и наработанными отраслевыми библиотеками.

жив Linux (и Android как его младший клон) — до тех пор будет жив и язык C[6][7].

По языку C существует множество книг, учебников, учебных курсов (ещё бы, при такой биографии!). Но, как ни странно, до сегодня **лучшим** руководством является оригинальная книга «Язык программирования Си», написанная в 1978 году, которую написали Брайан Керниган и Деннис Ритчи (легендарное руководство «K&R»), число изданий которой ведёт счёт уже на десятки. При всём богатстве выбора, все сегодняшние студенты начинают изучение языка C именно с K&R.

Но научиться просто языку C для практического программирования — мало! Ещё 50% успеха обеспечивает знание среды, окружения, **основных** библиотечных функций … которые по привычке и терминологически неправильно называют стандартной библиотекой C. Набор таких библиотечных функций, эволюционирующий в среде C, позже выкристаллизовался и формализовался в наборе стандартов POSIX[8].

Одной из слабо описанных частей языка C и стандартов API POSIX является проблема локализации текстовых строк в коде C и C++. Она состоит в том, как прозрачно (независимо от системы, настроек, конкретного декодирования в коде и т.д.) обрабатывать взаимодействие с внешней (относительно программного кода) средой (терминал, файловая система, сеть, …) на **любых** национальных языках … отличных от английского: русском, китайском, арабском, ...

Но … обратим внимание, что эта тема (работа с локализованными строками) почти не отражена при всей обширности публикаций по языкам C и C++. На то есть целый ряд причин:

- сам тип локализованных символов (`wchar_t`) появился в стандарте C89, но, в полной мере с API поддержки и т.п., только в стандарте C99 ... относительно недавно (по крайней мере, недавно, в сравнении с 45-летней историей C);
- и, конечно, этот тип и всё, что связано с локализацией, не может даже **упоминаться** в классической литературе по C периода его становления: K&R и т.п.;
- все более поздние книги и учебники по C, те которые **переводные**, тоже практически полностью обходят эту тему стороной ... их англоязычным авторам она совершенно не интересна, не близка — оно им не актуально ... да они и сами этой части языка просто слабо знают с ней не пересекаясь;
- отечественные же, русскоязычные **учебники** (а здесь встречаются только учебные книги по C, для студентов университетов, например ... кто же станет писать "не-учебник" по столь древнему языку?) — здесь авторы-педагоги, не являющиеся **практиками** программной разработки, сами также, главным образом, переписывают и пересказывают материал из англоязычных изданий ... ну, ещё придумают десяток собственных примеров кода; но раз в первоисточниках этого нет, то его и вообще нет в природе;
- далеко не последнюю роль (а может даже ведущую) сыграло то обстоятельство, что последние 25-30 лет основная масса русскоязычных практикующих программистов работали успешно подавляющим образом в аутосорсинге, на зарубежных заказчиков, где вопросы локализации не стояли;

Но картина и потребности времени радикально меняются … в связи с требованиями и объёмами импортозамещения и создания оригинального локализованного программного обеспечения. А также из-за свежих законодательных требований по использованию отечественных операционных систем, все они из которых являются теми или иными клонами Linux. Ещё жёстче становятся эти требования в условиях всё возрастающих санкций и эмбарго со стороны англо-саксонских (да и не только) стран. И это давление определённо не будет ослабляться в ближайший десяток лет...

Всё последующее изложение построено на стандартах POSIX и использовании операционной системы Linux. К Windows это относится **косвенно**, только в общих принципах и в той части, которая совместима с POSIX … или там где это коротко оговорено явно особо.

6 В новых языках программирования (Go, Python и мн. др.) сами стандарты языка оговаривают представление символьной информации в UTF-8 — там проблемы локализации гораздо проще. Но C (а также и C++) — это достаточно старые языки, и всё, что касается локализации, пришлось в них вводить «на ходу», при эксплуатации, более поздними стандартами.

7 В самые последние (2021-2022) годы был проделан (впервые за 50 лет UNIX и 30 лет Linux) успешный опыт написание фрагментов кода **ядра** операционной системы (Linux) на новом языке надёжного системного программирования Rust.

8 Linux использует набор системных API расширенный относительно классического POSIX, но эти расширения не затрагивают область локализации.

Примечание о примерах кода C/C++

Основной иллюстрацией рассматриваемых положений — это примеры кода. Предполагается показать много, но очень не крупных примеров. Поэтому, только отдельные, наиболее обстоятельные примеры будут показаны как отдельные законченые приложения, а вот набор простейших мини-тестов разумно оказалось свести в единое приложение (`unicode.c`), в котором определяется целый **набор последовательных** функций-тестов примерно вот такого вида:

```
void test00(void) { /* тест № 0 */
   ...
}
void test01(void) { /* тест № 1 */
   ...
}
...
void (*tests[])(void) = {       // последовательность тестов
        test00, test01, test02,
        test03, test04, test05,
        test06, test07, test08,
        test09, test10, test11,
        test12, test13, test14,
        /* ... */
};

static void do_test(int i) {
   printf("%02d ---------------------------------------\n", i);
   stdout = freopen(NULL, "w", stdout);
   tests[i]();
   stdout = freopen(NULL, "w", stdout);
}

int main(int argc, char **argv, char **envp) {
   int i, j;
   for (i = 0; i < sizeof(tests) / sizeof(tests[0]); i++)
      if (1 == argc)
         do_test(i);
      else
         for (j = 0; j < argc - 1; j++)
            if (atoi(argv[j + 1]) == i)
               do_test(i);
   printf("-----------------------------------------\n");
   return 0;
}
```

Назначение строк вида `stdout = freopen(…)` будет объяснено вскорости.

Такая структура последовательности мини-тестов позволяет: а). избежать загруженности отдельными однотипными приложениями, б). позволяет крайне легко добавлять код новых тестов в общий набор тестов, и в). при запуске выполнять либо всю последовательность помещённых тестов, либо указать только выборочные из них.

Вот так выполняются **вся последовательность** размещённых тестов:

```
$ ./unicode
...
```

А вот так может выполняться (отлаживаться) только **выборочный набор** из этих тестов:

```
$ ./unicode 1 3 5 6
...
```

Строки в C/C++

Представление и обработка строчной информации — это отчётливо слабая сторона языка C, и не самая сильная сторона C++. Для проектов, предполагающий активные контекстные операции с

текстовыми строками, лучше применить языки, гораздо лучше для того предназначенные: Perl, Python, Ruby, Go … в конце концов, bash.

Для внутреннего представления символьных строк, со времён ранних разработок в языках программирования (да и в других IT инструментах, например системах управления базами данных) было придумано **только** 2 принципиально различающихся способа: 1). последовательность символов, которой предшествует поле длины строки, числа последующих дальше символов и 2). последовательность символов, заканчивающаяся (ограниченная) символом, значения которого заведомо не может быть в составе строке (терминальным символом, обычно со значением 0 — поэтому этому значению ни в одной кодовой таблице не соответствовал никакой символ). 1-й способ получил известность как Pascal-строки, второй — как C-строки.

(Попутно отметим, и это только подчёркивает общность этих методов, что точно такие же 2 способа доступны для формата и разбиения «сообщений» в протоколе TCP/IP — в TCP нет никаких пакетов или дейтаграмм, TCP — это поток байт, и строки или сообщения в нём нужно формировать некоторым искусственным способом.)

В классическом C строки представляются просто как массив последовательных **байт**, отображающих символы (1 символ — 1 байт). По соглашению, завершением строки является байт с нулевым численным значением (этот символ-ограничитель не включается в состав строки, в её длину). Этим соглашением строки, вообще то говоря, разграничиваются с массивами вообще, например, с такими же массивами байт (массив байт, например, может иметь длину, ёмкость 100, а размещённая в нём текущая строка — длину 10 байт-символов, со значением 11-го элемента '\0'):

```
char array[100] = "123456789A";
```

Классически, во всех книгах по C, символьные строки представляются как массив `char`, символьные константы заключаются в двойные кавычки (`"this is a string"`), а отдельные символы — в одиночные кавычки ('R'). Такие строки ещё обозначают аббревиатурой ASCIIZ, подчёркивая, что это строка **исключительно** ASCII символов, завершающаяся нулевым байтом (Zero).

Размер `char` представляется байтом, хотя в разных реализациях (операционных систем) `char` может варьироваться как знаковое или беззнаковое байтовое значение (или указываться явно: `unsigned char` или `signed char`)... но для наших дальнейших целей это не существенно. Для работы со строками C стандарт POSIX определяет очень большой набор API (файл `<string.h>`) — набор функций разнообразной строчной обработки, многие из которых (но не все) имеют вид `str*()`.

Такие же строки (массивы) **байт** могут содержать (хранить) и **мультибайтные** последовательности локализованных символов, представленных в кодировке UTF-8 (принятой во всех современных реализациях Linux). При этом содержимое **одного** какого-то отдельно взятого, вычленненого байта в этой строке может быть полной бессмыслицей в терминологии «символов», например, байт со значением `0xD0`. Контекстная обработка (по содержимому) таких строк, классическими строчными функциями, для некоторых операций будет не корректной (и мы остановимся на этом подробно позже).

Посчитайте символы, байты и байт на символ в последнем, иероглифическом примере:

```
void test00( void ) {
   printf( "размер символа wchar_t вашей реализации = %ld байт\n", sizeof( wchar_t ) );
}
void test01( void ) {
   char str[] = "Привет по-русски!";
   printf( "%s [%ld байт]\n", str, strlen( str ) );
}
void test02( void ) {
   char str[] = "Hello, 世界";
   printf( "%s [%ld байт]\n", str, strlen( str ) );
}

$ ./unicode 0
00 --------------------------------------
размер символа wchar_t вашей реализации = 4 байт
----------------------------------------
$ ./unicode 1
01 --------------------------------------
```

```
Привет по-русски! [31 байт]
----------------------------------------
$ ./unicode 2
02 ------------------------------------
Hello, 世界 [13 байт]
----------------------------------------
```

Во всех **современных** дистрибутивах[9] Linux **всё** представление текстовой информации делается в кодировке UTF-8: текстовые файлы, файлы конфигурации, текстовые строки, настройки по умолчанию в текстовых редакторах и т.д. и т.п. (так же, как это имеет место в ОС Plan 9 или в более новых языках, например Python или Go, но C/C++ — это весьма старые языки). Когда вы **набираете** свой программный код C/C++ в своём любимом текстовом редакторе (или IDE), то вы уже тем самым вводите **все** символьные константы (то, что заключено в кавычки) в кодировке UTF-8 ... даже если вы набираете англоязычную строку `"xyz"` из примера в K&R 1979 года издания (когда ещё никто ничего не слышал про локализацию и Unicode).

Вы, конечно, можете **перенастроить** свой любимый текстовый редактор (или IDE), указав ему в качестве кодировки какую-то глупость... типа CP-866 или CP-1251 (и большинство редакторов такое позволяют сделать). И компилятор благополучно съест это, и на этапе выполнения функция `printf()` будет благополучно выводить это на терминал (или в файл), потому что функции ввода и вывода C никак не анализируют поток байт на принадлежность множеству допустимых символов, а тупо выводят байт за байтом в поток. Но на этапе выполнения вы будете при этом иметь большие хлопоты с визуализацией результатов (это будут не читаемые «кракозябры»)..., а если кто когда-то позже вздумает работать с этим вашим кодом, то поминать он вас будет такими словами, что в гробу вас будет крутить как пропеллер. Такое извращение может быть допустимо, но только только для очень специальных целей, например, подготовки кода для переноса в другую операционную систему. Таким образом, в итоге, **символьные константы** в текстовом файле, содержащем программный код C/C++, могут быть записаны в любой кодировке, которую вы использовали при подготовке этого кода, но везде в дальнейшем рассмотрении мы будем полагать, что эта кодировка — UTF-8[10].

Для представления же и обработки локализованных строк (национальных языков) позже (стандартом C89, а окончательно C99) был введен тип локализованных, широких (wide) символов `wchar_t`. Это абстрактный тип данных, не привязанный стандартом к какому-то фиксированному размеру. Но в POSIX/UNIX/Linux этот тип представляется 4-х байтным значением. В ОС Windows, использующей такое устаревшее представление Unicode как UTF-16, тип `wchar_t` имеет размер 2 байта[11]. Как бы там ни было, никогда не следует в своём коде делать неявные предположение о размере символа `wchar_t`.

Для записей широких символьных констант используется префикс-квалификатор: `L"this is a string"`. Точно так же обозначается и отдельный широкий символ: `L'R'`. (Эти примеры показывают, что как широкие символы могут записываться и англоязычные строки, но они при этом будут радикально отличаться содержанием от ASCIIZ строк **того же** написания.)

Так же, как и ASCIIZ, широкие строки завершаются нулевым значением **типа** `wchar_t` (но имеющим в этом случае размер 4 байт — `L'\0'`) — это **не байт** `'\0'`.

Язык C++ наследует все символьные представления своего предшественника C. Но, кроме того, библиотеки C++ вводят (заголовочный файл `<string>`) новое объектное представление строк: класс (тип) `string` — шаблонное (template) представление массива `char` динамического размера (который можно понимать как тип `vector<char>`). Для объектов этого класса определено множество **методов** обработки, которые будут многократно продемонстрированы далее. Сейчас для нас важно то, что `string` — это динамические массивы элементов типа `char`, и они точно так

9 Ещё не так много лет назад это было не так, и дистрибутивы Linux использовали по умолчанию 8-битовое представление символов в выбранной кодовой странице, например в KOI-8R для русскоязычного окружения.

10 Это очень напоминает ответ Генри Форда на замечания относительно цвета его автомобилей «Форд-Т»: «Цвет автомобиля может быть любым, при условии, что он черный».

11 Использование старого Unicode представления UTF-16 уже породило ряд проблем. Во-первых, из-за давно известной разницы в порядке байт (little endian и big endian), представляющих 16-бит целое, понадобилось вводить метку порядка байт (U+FEFF), а позже и кодировки размножились до различающихся UTF-16LE и UTF-16BE. Но хуже того, во-вторых, что со временем набор Unicode расширился, и 16 бит стало недостаточно для его полного представления. Тогда потребовалось введение суррогатных пар — кодирование символа двумя словами. И в одних версиях (Windows 7 или 8) такие символы распознаются, а в других (Windows 95 или XP) — нет, и это причина непереносимости. К счастью, при кодировании UTF-8 в POSIX системах таких проблем вообще не возникает, и больше на них мы не будем обращать внимания.

же **не пригодны** для контекстной **обработки** локализованных строк, как и `char[]`. (Но они вполне могут использоваться для **хранения** локализованных строк представленных кодированием UTF-8).

Аналогично, как для C тип `wchar_t`, для работы с локализованными текстами C++ вводит (файл `<wstring>`) класс (тип) `wstring` — динамические массивы элементов типа `wchar_t` (`vector<wchar_t>`).

Наконец, для взаимных преобразований мультибайтных (переменной длины) представлений символов и строк UTF-8 и широких локализованных символов UTF-32 (`wchar_t`) стандартная библиотека C (стандарт C99) вводит целую группу функций с именами вида `*mb*()` (multi bytes): `mblen()`, `mbtowc()`, `mbstowcs()`, `wcstombs()` и т. д. Их использование позволяет (и не предполагает) не ковыряться с внутренним побайтным UTF-8 представлением символов разных языковых страниц.

Всё это, пунктиром обозначенное в этой части, будет неоднократно и детально проиллюстрировано далее многочисленными примерами кода.

Локали и локализация

Для ввода/вывода ваш программный код должен знать **локаль** того устройства, с которым осуществляются операции ввода-вывода (локаль, локализация — это более обширное понятие, но нас будут интересовать только языковые настройки). Но кроме того, программа (используемые языковые библиотеки) может иметь свою собственную локализацию, установленную по умолчанию (при старте программы `main()`):

```
$ ./unicode 3
03 --------------------------------------
локаль программы по умолчанию: C
----------------------------------------
```

Программы на C (и C++) устанавливают по умолчанию такую локализацию. Но локализация C или POSIX устанавливают **7-битное** кодирование, способное представлять **только** основную таблицу ASCII: 0-127 (принятое на 70-е годы XX века, тот набор символов, которые могут встречаться в самой записи кода программы на C). Ни о каком интернациональном вводе/выводе в таком случае не может быть и речи, независимо от того, какая локализация (по умолчанию) установлена в операционной системе.

Для того, чтобы иметь возможность локализованного ввода/вывода кодом C/C++, необходимо установить в коде подходящую локаль — либо установленную по умолчанию в операционной системе, либо принудительно одну из тех, которые инсталлированы в этой операционной системе.

Локаль операционной системы **по умолчанию**:

```
$ locale
LANG=ru_RU.utf8
LC_CTYPE="ru_RU.utf8"
LC_NUMERIC="ru_RU.utf8"
LC_TIME="ru_RU.utf8"
LC_COLLATE="ru_RU.utf8"
LC_MONETARY="ru_RU.utf8"
LC_MESSAGES="ru_RU.utf8"
LC_PAPER="ru_RU.utf8"
LC_NAME="ru_RU.utf8"
LC_ADDRESS="ru_RU.utf8"
LC_TELEPHONE="ru_RU.utf8"
LC_MEASUREMENT="ru_RU.utf8"
LC_IDENTIFICATION="ru_RU.utf8"
LC_ALL=
```

Набор локалей, которые вообще инсталлированы в конкретной операционной системе:

```
$ locale -a | grep ru
ru_RU
ru_RU.iso88595
ru_RU.koi8r
ru_RU.utf8
```

```
russian
ru_UA
ru_UA.koi8u
ru_UA.utf8
```

Полное число установленных локалей может быть очень значительным (поэтому выше показана только небольшая часть их):

```
$ locale -a | wc -l
817
```

Эти команды крайне полезны для **правильной** записи локали в коде C/C++. При ошибке в записи (строки) локали возникнет ошибка установка локали в C (и возбуждение исключения в C++), текущая локаль программы при этом не изменится.

Детали локализации в C

В принципе, если вы собираетесь **только** хранить и выводить локализованные символьные строки (константы), не анализируя или трансформируя их содержимое (контекст), то вы можете вообще не заморачиваться с локализацией: многобойтные последовательности русских литер (в UTF-8) будут корректно копироваться, переноситься или отображаться. Повторим показанный уже ранее пример:

```
void test01( void ) {
   char str[] = "Привет по-русски!";
   printf("%s [%ld байт]\n", str, strlen(str));
}

$ ./unicode 1
01 ---------------------------------------
Привет по-русски! [31 байт]
-----------------------------------------
```

При этом нужно быть готовым к тому, что число **символов** в строке выше 17, но число **байт** в строке будет 31 (результат возвращаемый `strlen()` **как длина** строки). Поэтому работать не принимая во внимание локализацию можно, но при этом нужно соблюдать осторожность. Что происходит, если самонадеянно не задумываясь использовать строки `char[]` для представления русскоязычных (и любых других иноязычных) строк, легко увидеть проанализировав работу функции **побайтового** реверса строк, сравнив результаты для русскоязычной и англоязычной строк:

```
static char* revb(char *s) {
   int i, j;
   for(i = 0, j = strlen( s ) - 1; i <= j; i++, j--) {
      char c = s[i];
      s[i] = s[j];
      s[j] = c;
   }
   return s;
}

void test06( void ) {
   char se[] = "abcdefghijklmnopqrstu",
        sr[] = "абвгдеёжзийклмнопрсту";
   printf("%s => %s\n", se, revb(strdup(se)));
   printf("%s => %s\n", sr, revb(strdup(sr)));
}

$ ./unicode 6
06 ---------------------------------------
abcdefghijklmnopqrstu => utsrqponmlkjihgfedcba
абвгдеёжзийклмнопрсту => ⟌тсрѡонмлкйизжБvдгвба⟌
-----------------------------------------
```

Как только наш код начинает анализировать или изменять **содержимое** строки, нам необходимо работать с только с локализованными строками (строками широких символов `wchar_t[]`).

Первейшим действием программы мы должны установить (<locale.h>) подходящую локаль (при ненадлежащим образом указанной локали все преобразования между char[] и wchar_t[] в обоих направлениях будут **ошибочными**).

Это может быть принудительно указанная локаль:

```
void test04(void) {
   char *loc = setlocale(LC_CTYPE, "ru_RU.utf8");
   if(NULL == loc) perror("locale error");
   else fprintf(stdout, "локализация: %s\n", loc);
}

$ ./unicode 4
04 ---------------------------------------
локализация: ru_RU.utf8
----------------------------------------
```

Или это может быть локаль но умолчанию, устанавливаемая переменной окружения LANG:

```
void test05(void) {
   char *loc = setlocale(LC_ALL, ""); // по умолчанию ("") - из $LANG
   fprintf(stdout, "локализация: %s\n", loc);
}

$ echo $LANG
ru_RU.utf8
$ ./unicode 5
05 ---------------------------------------
локализация: ru_RU.utf8
----------------------------------------
$ LANG=japanese.euc; ./unicode 5
05 ---------------------------------------
локализация: japanese.euc
----------------------------------------
```

Примечание: Почему необходимо в программе устанавливать setlocale() в коде C и C++, например при преобразовании мультибайтного представления (UTF-8) в широкие символы wchar_t (UTF-32)? Это достаточно интересный вопрос если вспомнить, что: а). 4-х байтовое на символ представление Unicode (в понимании Linux) **однозначно** определяет как кодовую страницу (язык) так и код символа в этой таблице, а б). UTF-8 кодирование (от 1 до 6 байт на символ) однозначно соответствует символу Unicode.

Дело в том, что традиционно программа C/C++ сама и по умолчанию устанавливает локаль "C" или "POSIX" (так говорит стандарт POSIX). Это установилось много-много лет назад, и в такой локали не может быть никаких многобайтных символов UTF-8, в ней отображаются только **7-бит** ASCII символы (так было ещё на компьютерах семейства PDP, на которых первоначально отрабатывались и язык C и операционная система UNIX). В **любой** UTF-8 локали, независимо от языковой локализации, все преобразования с любыми языками будут выполняться корректно. Что и показывает нам пример:

```
void test14(void) {
   printf("locale: %s\n", setlocale(LC_ALL, NULL));
   setlocale(LC_CTYPE,"en_US.utf8");
   printf("%ls : %s\n", L"русская строка в локали", setlocale(LC_CTYPE, NULL));
}

$ ./unicode 14
14 ---------------------------------------
locale: C
русская строка в локали : en_US.utf8
----------------------------------------
```

Таким образом, выполняя setlocale() в коде C/C++, мы не сколько устанавливаем нужную нам

локаль, сколько **восстанавливаем** символьное представление UTF-8, по умолчанию используемое во всех современных дистрибутивах Linux.

API для работы со строками

Для традиционных строк C предоставляется (<string.h>) очень большое число функций для работы со строками вида str*() и подобные им (memmove() и др.) — на все случаи жизни. Относительно операций со строками C важно напомнить вот такие правила, которые очень часто забывают и нарушают начинающие программисты, и которые поэтому стоит напомнить:

1. Строки char[] (char*) **нельзя присваивать**. В C операция присваивания (=) — это копирование значения (даже для агрегатных переменных с типом struct {...}). Для копирования значений строк предназначен целый ряд функций группы: strcpy(), strncpy(), memcpy(), memmove(), …
2. Строки нельзя сравнивать (операциями ==, <, > и т. п.). Для сравнения строк вводится операция strcmp() (и strncmp()), которая возвращает результат лексографического сравнения — целое число, которое меньше, больше нуля или равно нулю, если одна строка соответственно предшествует (меньше), следует(больше) или равна другой строке, с которой сравнивается.

Для строк широких (локализованных) символов определён (<wchar.h>) практически полностью эквивалентный[12] набор таких же функций, имеющих вид wcs*(). Например, вызову strlen() сопоставлен вызов wcslen(), strsncpy() сопоставлен wcsncpy(), strcat() сопоставлен wcscat(), memmove() сопоставлен wmemmove() и т.д.

Кроме того, определён (стандартом C99) целый ряд функций для работы с мультибайтными последовательностями (UTF-8), взаимными преобразования между ними и широкими символами (mbtowc(), mblen(), mbstowcs(), wcstombs() и др.). Это механизм взаимных преобразований между char[] и wchar_t[].

Функции ввода/вывода дополнены (<wchar.h>) эквивалентами относительно работы с байтовыми строками: fputws() (эквивалент fputs()), fputwc() (эквивалент fputc()), и так далее: getwchar(), fgetwc(), fgetws() и т.д.

Наконец, API форматного ввода вывода (printf(), sprintf(), scanf() и т.д.) получили (C99) новый формат для широких локализованных строк: если для байтовый строк используется формат %s, то для широких строк — формат %ls.

Это краткого обзора API строк широких символов вполне достаточно для работы с локализованными строками. Тонкие детали использования функций этого API можно получить из man-страниц, которые предоставлены по всем таким функциям.

Разрушение потоков ввода/вывода

Начнём выводить в выводной поток (это наиболее наглядно) традиционные и широкие символы:

```
void test10(void) {
   setlocale(LC_ALL, "");
   char cs[] = "c-строка";
   wchar_t ws[] = L"w-строка";
   printf("%s\n", cs);
   printf("%ls\n", ws);
   printf("%s\n", cs);
   printf("%ls\n", ws);
}
```

Казалось бы, что у нас попеременно в выходной поток пишутся и традиционные и широкие строки:

```
$ ./unicode 10
10 ---------------------------------------
```

12 Чтоб это не было неожиданностью — не совсем для всех функций srt*() вы найдёте прямой аналог wcs*(). Для 2-х подобных вызовов strtok() и strtok_r() (потоково-безопасный вариант, не вовлекающий в работу статических переменных), представлен только 1 эквивалент с 3-мя параметрами (эквивалентный именно strtok_r()): wcstok(wchar_t*, const wchar_t*, wchar_t**).

```
c-строка
w-строка
c-строка
w-строка
----------------------------------------
```

Но это дорогостоящее заблуждение! (в смысле поиска такой ошибки в более-менее объёмном проекте). Сделаем два симметричных тестовых приложений:

```
void test11( void ) {
   int res;
   char cs[] = "c-строка\n";
   wchar_t ws[] = L"w-строка\n";
   setlocale( LC_ALL, "" );
   res = fputws( ws, stdout );
   printf( "%d: %m\n", res );
   res = fputs( cs, stdout );
   printf( "%d: %m\n", res );
   res = fputws( ws, stdout );
   printf( "%d: %m\n", res );
   res = fputs( cs, stdout );
   printf( "%d: %m\n", res );
}

void test12(void) {
   int res;
   char cs[] = "c-строка\n";
   wchar_t ws[] = L"w-строка\n";
   setlocale(LC_ALL, "");
   res = fputs(cs, stdout);
   printf("%d: %m\n", res);
   res = fputws(ws, stdout);
   printf("%d: %m\n", res);
   res = fputs(cs, stdout);
   printf("%d: %m\n", res);
   res = fputws(ws, stdout);
   printf("%d: %m\n", res);
}
```

Результаты их выполнения оказываются обескураживающе различающимися:

```
$ ./unicode 11 12
11 ---------------------------------------
w-строка
w-строка
12 ---------------------------------------
c-строка
 1: Выполнено
 -1: Выполнено
c-строка
 1: Выполнено
 -1: Выполнено
----------------------------------------
```

Выходной поток `sysout` (и любой другой поток: `sysin`, `FILE*` ...) **разрушается** если в него чередуется (хотя бы один раз) вывод традиционных и широких строк (да так, что в первом из показанных тестов `printf()` не может вывести даже сообщение о произошедшей ошибке!). Об этом явно сказано в C++ документации API `<iostream>` и чуть позже мы зацитируем эту фразу, когда дойдём непосредственно к C++.

Как же тогда выполнить, если необходимо, чередующийся вывод (или ввод) традиционных и широких символов в один поток? Нужно **переоткрыть** поток! Так:

```
stdout = freopen( "/dev/stdout", "w", stdout );
```

Или даже так, ещё проще:

```
stdout = freopen( NULL, "w", stdout );
```

И тогда показанный тест примет такой, например, вид:

```
void test13(void) {
   int res;
   char cs[] = "c-строка\n";
   wchar_t ws[] = L"w-строка\n";
   setlocale(LC_ALL, "");
   res = fputs(cs, stdout);
   printf(" %d: %m\n", res);
   stdout = freopen(NULL, "w", stdout);
   res = fputws(ws, stdout);
   stdout = freopen(NULL, "w", stdout);
   printf(" %d: %m\n", res);
   res = fputs(cs, stdout);
   printf(" %d: %m\n", res);
   stdout = freopen(NULL, "w", stdout);
   res = fputws(ws, stdout);
   stdout = freopen(NULL, "w", stdout);
   printf(" %d: %m\n", res);
}
$ ./unicode 13
13 --------------------------------------
c-строка
 1: Выполнено
w-строка
 1: Выполнено
c-строка
 1: Выполнено
w-строка
 1: Выполнено
----------------------------------------
```

А как же `printf()` в одном из тестов выше (`test10()`) спросите вы? ... когда вперемешку выводились и традиционные и широкие символы (даже в едином вызове `printf()`). Но `printf()` — это **библиотечный** вызов, не **системный** (он описан в секции 3 man, а не 2). Он последовательно вызывает библиотечный `sprintf()` и, затем, системный `write( 1, ... )` для вывода в `stdout`. После `sprintf()` (форматирования строки вывода) и формата `%ls` — в строке подлежащей выводу нет уже никаких `wchar_t`, там только мултибайтные UTF-8 цепочки char, поэтому проблем и не возникает. Но если поток вывода **разрушен** (для `char`) ранее выполненным вызовом `fputws()`, то уже и строка `char[]`, подготовленная `sprintf()` **не может** быть выведена.

Некоторые примеры

Посимвольное преобразование русскоязычной строки, представленной ASCIIZ в мультибайтной кодировке UTF-8 в строку широких локализованных символов:

```
#define LENGTH 160
char    buf  [LENGTH] = "тестовая русскоязычная  строка  в UTF-8 с прямым порядком слов ";
wchar_t wbuf [LENGTH];
//-----------------------------------------------------
inline void c2w(char *c, wchar_t *w) {
   int n = -1;
   setlocale(LC_ALL, "");  // только после этого работают преобразования!
   While (n != 0)
      c += (n = mbtowc(w++, c, MB_CUR_MAX));
}

void test07(void) {
```

```
   printf("преобразование UTF-8 символов в широкие (wchar_t):\n");
   printf("строка UTF-8 до преобразования: '%s'\n"
          "длина UTF-8 строки = %d байт\n",
          buf, (int)strlen(buf) );
   c2w(buf, wbuf);
   printf("локаль программы установлена: %s\n",
          setlocale(LC_ALL, NULL));
   printf("преобразованная строка: '%ls'\n"
          "длина преобразованной строки = %d символов (%ld байт)\n",
          wbuf, (int)wcslen(wbuf),
          wcslen(wbuf) * sizeof(wchar_t));
}
```

Здесь MB_CUR_MAX — это константа, максимальная длина в байтах на символ в выбранной локали, и может иметь значения до 6-ти.

```
$ ./unicode 7
07 ---------------------------------------
преобразование UTF-8 символов в широкие (wchar_t):
строка UTF-8 до преобразования: 'тестовая русскоязычная  строка  в UTF-8 с прямым порядком слов
'
длина UTF-8 строки = 110 байт
локаль программы установлена: ru_RU.utf8
преобразованная строка: 'тестовая русскоязычная  строка  в UTF-8 с прямым порядком слов '
длина преобразованной строки = 63 символов (252 байт)
----------------------------------------
```

Обратное преобразование полученной строки (`wchar_t[]`) в форму UTF-8. Если в предыдущем примере мы преобразовывали строки посимвольно (`mbtowc()`) в цикле, то теперь используем функции, преобразующие целиком всю строку (`wcstombs()`):

```
void test08(void) {
   int n;
   c2w(buf, wbuf);
   printf("обратное преобразование в UTF-8: %d байт\n", n = wcstombs(NULL, wbuf, 0));
   wcstombs(buf, wbuf, n + 1);  // с завершающим нулём
   printf("преобразованная UTF-8 строка: '%s'\n", buf);
}
```

```
$ ./unicode 8
08 ---------------------------------------
обратное преобразование в UTF-8: 110 байт
преобразованная UTF-8 строка: 'тестовая русскоязычная  строка  в UTF-8 с прямым порядком слов '
----------------------------------------
```

Реверс русскоязычных **слов**, составляющих фразу. Здесь уже работает анализ и трансформация **содержимого** локализованного текста:

```
void revers(wchar_t *w) {
   wchar_t *sec, wb[40];
   if(NULL == (sec = wcschr(w, L' '))) return;
   wcsncpy(wb, w, sec - w)[sec - w] = L'\0';
   while(L' ' == *sec) sec++;
   revers(sec);
   wcscat(wcscat(wmemmove(w, sec, wcslen(sec) + 1), L" "), wb);
}

void test09(void) {
   c2w(buf, wbuf);
   while(L' ' == wbuf[wcslen(wbuf) - 1])
      wbuf[wcslen(wbuf) - 1] = L'\0';
   printf("устранение завершающих пробелов: '%ls'\n", wbuf);
   revers(wbuf);
```

```
   printf("реверсирование слов: '%ls'\n", wbuf);
   revers(wbuf );
   printf("реверсирование слов: '%ls'\n", wbuf);
}
```

```
$ ./unicode 9
09 ------------------------------------
устранение завершающих пробелов: 'тестовая русскоязычная  строка  в UTF-8 с прямым порядком
слов'
реверсирование слов: 'слов порядком прямым с UTF-8 в строка русскоязычная тестовая'
реверсирование слов: 'тестовая русскоязычная строка в UTF-8 с прямым порядком слов'
----------------------------------------
```

Здесь, для контроля достоверности, мы делаем 2 последовательных реверса (прямой и обратный), чтобы в результате восстановить первоначальный вид строки.

Следующим примером мы сделаем чтение из файла и вывод на терминал локализованных (кирилических) строк. Пишем программу, которая читает из файлов и русскоязычные и англоязычные строки (с одинаковым успехом) и корректно выводит их содержимое на экран. Чтение из файла производим **сразу** в строку широких символов (`wchar_t`, Unicode):

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

#define MAX_TXT 100
int main(int argc, char *argv[]) {
   if (argc != 2) {
      printf("нужно указать имя входного файла\n");
      return 1;
   }
   char *loc = setlocale(LC_ALL, "ru_RU.utf8"); // char *loc = setlocale(LC_ALL, "");
   printf("локализация %s\n", loc);
   FILE *fi = fopen(argv[1], "r");
   if (!fi) {
      printf("ошибка открытия файла %s: %m\n", argv[1]);
      return 1;
   }
   wchar_t buf[MAX_TXT];
   do {
      if (NULL == fgetws(buf, MAX_TXT, fi)) {
         if (feof(fi) != 0) break;                 //EOF
         printf("ошибка чтения: %m\n");
         return 1;
      }
      printf("%ls", buf);
   } while (0 == feof(fi));
   fclose(fi);
   return 0;
}
```

Программа с одинаковым успехом читает и русские и английские тексты:

```
$ ./utype r1.txt
локализация ru_RU.utf8
тестовая строка русского текста
$ ./utype e1.txt
локализация ru_RU.utf8
test string in English
```

Обращаем внимание на то, что строки входного файла, записанные в кодировке UTF-8, считываются в строку `wchar_t buf[]` без каких либо явных преобразований в коде через функции мультибайтных строк `mb*()`. Работа с потоками Unicode-строк будет корректной только после

установки локализаци `setlocale(LC_ALL, "ru_RU.utf8")`.

Обращаем внимание на формат вывода широких строк `printf("%ls", ...)`, причём (важно!) в списке элементов вывода `printf()` могут вперемешку стоять как широкие строки, так и обычные ASCII строки, каждые со своими, естественно, соответствующими форматами (`"%ls"` и `"%s"`).

Детали локализации в C++

Операции со строками

C++, понятно, наследует все возможности C относительно строк, представляемых как массивы `char[]` и `wchat_t[]`. Но C++ вводит новое (и предпочтительнее) объектной представление строк `string` и `wstring`. Большая часть операций со строками, реализующиеся в C функциями API, реализуются для объектов этих классов функциями-методами, за исключением вот таких важных особенностей и отличий от строк в стиле C:

1. Строки C++ можно присваивать операцией = (копировать значение);
2. Строки C++ можно сравнивать типовыми операциями: ==, !=, <, <=, >, >=. Строки сравниваются в лексографическом порядке. Естественно, что итог сравнения одних и тех же строк зависит от выбранной локали;
3. Строки можно конкатенировать (объединять) простым указанием операции + (и, соответственно +=);
4. Существует метод `c_str()`, возвращающий **внутреннее содержимое** строки в форме массива символов (`const char*`);

Как видно и из последнего утверждения, переменные-объекты класса `string/wstring` — это неизменяемые объекты (в том же смысле, как в языке Python и др.). Это не означает константность, это совсем другое:

```
string s = "строка 1";
s = "строка 2"
```

Здесь операцией присвоения переменной `s` будет присвоен **новый** объект, созданный вызовом **конструктора** с инициализирующим значением `"строка 2"`. Предыдущий объект с значением `"строка 1"` будет уничтожен, для него будет вызван **деструктор** при выходе из области определения объекта (блока). Новый и старый объекты будут размещены по разным адресам. В этом смысле и понимается неизменяемость: при модификации значения объекта, новое значение не изменяет старое, а инициализирует новый объект.

Все эти принципы полностью переносятся и на локализованные строки широких символов `wstring`, с той единственной разницей, что `string` является контейнером однобайтовых `char`, а `wstring` — это контейнер 4-х байтовых широких символов `wchar_t`.

Потоки ввода-вывода локализованных символов

Библиотеки C++ определяют (`<iostream>`) отдельные потоки ввода вывода для широких строк, `wcout` (эквивалент `cout`) и `wcin` (эквивалент `cin`). Точно так же как и на языке C, прежде чем осуществлять операции с потоками широких символов, необходимо установить (изменить) локаль программы:

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

void test00(void) {
   locale::global(locale(""));
   wcout << L"строка" << endl;
}

$ ./unicode++ 0
---------------------------------------
строка
```

```
--------------------------------------
```

Это эквивалентно установки той языковой локали программы, которая установлена в системе по умолчанию, переменной окружени LANG:

```
$ LANG=norwegian; ./unicode++ 0
--------------------------------------
??????
--------------------------------------
```

Иногда целесообразно принудительно установить локаль, но это лучше делать в блоке `try {}`, поскольку указание ошибочной строки, указывающей локаль, приведёт к возбуждению исключения:

```
#include <locale>
#include <iostream>
#include <stdexcept>
using namespace std;

void test01(void) {
   locale loc;
   try {
      loc = std::locale("ru_RU.utf8");
   }
   catch(std::runtime_error&) {
      loc = std::locale(loc, "", std::locale::ctype);
   }
   locale::global(loc);
   wcout << L"строка" << endl;

}

$ ./unicode++ 1
--------------------------------------
строка
--------------------------------------
```

Примечание: Все примеры кода компилировались с опцией совместимости с стандартом C++11 :

```
$ g++ xxx.cc -Wall -std=c++11 -o xxx
```

Разрушение ориентации потоков

Каждый поток ввода/вывод может работать эксклюзивно только в одном из режимов (ориентации): работа с символами `char`, или работа с символами `wchar_t`. А поскольку потоки, скажем, `cout` и `wcout` представляют один физический поток вывода, то вывод чего либо (независимо от содержания) в поток `cout` разрушает состояние `wcout` и наоборот. Смешанное использование потоков **невозможно**.

Об этом явно сказано в документации API `<iostream>`:

A program should not mix output operations on `wcout` with output operations on cout (or with other narrow-oriented output operations on stdout): Once an output operation has been performed on either, the standard output stream acquires an orientation (either narrow or wide) that can only be safely changed by calling freopen on stdout.

Аналогичную картину мы уже наблюдали ранее в языке C, что не является неожиданным, поскольку механизмы C++ (API) — это логическая надстройка над базовыми элементами (библиотеке) C, в данном случае над потоками.

В иллюстрацию этого любопытно рассмотреть возникающие при этом эффекты, поскольку подобные вещи могут явиться большой неожиданностью и обескуражить результатом:

```
void test02(void) {
   locale::global(locale(""));
   cout << "строка1" << endl;
   wcout << L"строка2" << endl;
   cout << "строка3" << endl;
```

```
   wcout << L"строка4" << endl;
}
//-------------------------------------------------------
void test03(void) {
   locale::global(locale(""));
   wcout << L"строка1" << endl;
   cout << "строка2" << endl;
   wcout << L"строка3" << endl;
   cout << "строка4" << endl;
}

$ ./unicode++ 2
--------------------------------------
строка2
строка4
--------------------------------------
$ ./unicode++ 3
--------------------------------------
строка1
строка3
--------------------------------------
```

Как легко видеть, в первом случае на терминал выводится только то, что выводится в `wcout`, а во втором — только то, что выводится в `cout`.

Для **восстановления** ориентации потока (классические или широкие символы) его нужно переоткрыть заново:

```
void test04(void) {
   locale::global(locale(""));
   stdout = freopen("/dev/stdout", "w", stdout);
   cout << "строка c1" << endl;
   stdout = freopen("/dev/stdout", "w", stdout);
   wcout << L"строка w2" << endl;
   stdout = freopen("/dev/stdout", "w", stdout);
   cout << "строка c3" << endl;
   stdout = freopen("/dev/stdout", "w", stdout);
   wcout << L"строка w4" << endl;
}

$ ./unicode++ 4
--------------------------------------
строка c1
строка w2
строка c3
строка w4
--------------------------------------
```

Документация `freopen (3)` даёт нам подсказку как это сделать проще: *If filename is a null pointer, the function attempts to change the mode of the stream. Although a particular library implementation is allowed to restrict the changes permitted, and under which circumstances.*

В итоге:

```
void test05(void) {
   locale::global(locale(""));
   stdout = freopen(NULL, "w", stdout);
   cout << "строка1" << endl;
   stdout = freopen(NULL, "w", stdout);
   wcout << L"строка2" << endl;
   stdout = freopen(NULL, "w", stdout);
   cout << "строка3" << endl;
   stdout = freopen(NULL, "w", stdout);
   wcout << L"строка4" << endl;
```

```
}

$ ./unicode++ 5
--------------------------------------
строка1
строка2
строка3
строка4
--------------------------------------
```

Некоторые современные языки

> *Когда же число людей очень увеличилось, они решили построить город и в нём башню высотой до небес, чтобы приобрести себе славу.*
>
> *Но Богу не угодно было их намерение. Он смешал язык строителей так, что они начали говорить на разных языках и перестали понимать друг друга.*
>
> *Кн. Быт. 11:1 – 9*

Выше мы весьма подробно обсуждали то, какие структуры **декларируются** для представления строк в языках C и C++. В более современных языках (грубо можно считать примерно от времени формирования Python версии 3) разработчики, наученные печальным опытом нескольких десятилетий использования C/C++, пошли другим путём: они **декларируют**, что структура представления строки в языке является внутренним делом языка, недоступным и ненужным пользователю. При этом вводится **тип** строки, со своим набором операций, и декларируется что строка является **неизменяемым** объектом.

Вот эта **неизменяемость** строк приводит временами в смущение начинающих работать с этими языками. Но это вовсе не означает что **содержимое** строки невозможно изменить, например, "собака" превратить в "сабака" — но полученная строка это будет совершенно другая строка, размещённая по другим адресам в памяти: содержимое исходной строки **скопировано** в результат с произведенным над ним изменением. Неизменяемость означает что нельзя указать изменить отдельный фрагмент строки «на месте», в предельном случае один символ, указав что-то типа [3] как это делалось в C/C++. И это очень перекликается с не объявлением внутренней структуры строчных переменных — если вы ничего не можете поменять в этой структуре, то зачем вам её знать?

Python

В Python версии 2, бывшим превалирующим (в Linux) на протяжении около 20 лет вообще **не было** соглашений о дефаултной локали в коде:

```
$ python2
Python 2.7.18 (default, Jul  1 2022, 12:27:04)
[GCC 9.4.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import locale
>>> locale.getlocale()
(None, None)
```

Для указания использования кодировки UTF-8 в программе (скрипте) на Python требовалось явное указание этого (2-я строка):

```
$ cat l2.1.py
#!/usr/bin/python2
# -*- coding: utf-8 -*-
print("русскоязычная строка")
$ ./l2.1.py
русскоязычная строка
```

Но стоит вам убрать эту строку спецкомментария, и интерпретатор теряется и не знает что делать с

такой Unicode строкой:

```
$ cat l2.2.py
#!/usr/bin/python2
print("русскоязычная строка")
$ ./l2.2.py
  File "./l2.2.py", line 2
SyntaxError: Non-ASCII character '\xd1' in file ./l2.2.py on line 2, but no encoding declared;
see http://python.org/dev/peps/pep-0263/ for details
```

Но по стандарту Python версии 3 (начавшим распространение с 2008 года) регламентируется что всё в коде представляется в кодировке UTF-8. Сравните:

```
$ python3
Python 3.8.10 (default, Nov 14 2022, 12:59:47)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import locale
>>> locale.getlocale()
('ru_UA', 'UTF-8')
```

И если бы я писал этот фрагмент ещё год назад, я бы тщательно описывал детали различий в работе со строками Python версий 2 и 3 … и так это и было в предыдущих редакциях рукописи. Но на протяжении 2022, наконец, Python 3 становился **стандартом по умолчанию** для одного за другим для всех основных дистрибутивов Linux (и Python 2 даже не инсталлировался в системах по умолчанию… но мог ещё быть установлен вручную из пакетной системы). Поэтому всё дальнейшее изложение я вправе вести исключительно для Python 3.

И теперь для мультиязычных приложений нет сложностей в смысле vekmnbzpsxyjcnb, и не нужны никакие дополнительные явные указания — стандарт Python говорит что все строчные значения предоставляются в кодировке UTF-8:

```
$ cat l3.1.py
#!/usr/bin/python3
print("русскоязычная строка")
print("Hello, 世界")
$ ./l3.1.py
русскоязычная строка
Hello, 世界
```

Более того, и в самом коде (который также представляется в UTF-8) могут использоваться мультиязычные символы, например в именах переменных для большей ясности их значений:

```
$ python3
Python 3.8.10 (default, Nov 14 2022, 12:59:47)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> длина = 123
>>> print(длина)
123
```

Или, если угодно, так:

```
$ cat l3.2.py
#!/usr/bin/python3
import sys

def рекурсивная_функция_фибоначчи(n) :
    if n < 2 : return 1
    else: return рекурсивная_функция_фибоначчи(n - 1) + рекурсивная_функция_фибоначчи(n - 2)

n = int(sys.argv[1])
print("{}".format(рекурсивная_функция_фибоначчи((int(sys.argv[1])))))
$ ./l3.2.py 20
10946
```

Go

Язык Go был официально представлен в 2009 году, поэтому он, естественно[13], изначально ориентирован на использование во всём кодировки UTF-8.

Совершенно естественно, что все символьные константы в Go представлены в Unicode в кодировке UTF-8 (и на вводе и выводе с периферийными устройствами, терминалом). Но, более того, сама запись кода Go хранится в UTF-8, и допускает мультиязычные имена объектов в коде программы. Две небольших иллюстрации к сказанному (больше добавить нечего):

```
$ cat hello.go
package main
import ("fmt", "os")

func main() {
        fmt.Println("ты кто будешь?")
        fmt.Printf("> ")
        буфер := make([]byte, 120)
        длина, _ := os.Stdin.Read(буфер) // возвращается 2 значения
        Ω := длина
        ответ := string(буфер[:Ω-1]) // убрали '\n'
        fmt.Printf("какое длинное имя ... целых %d байт\n", Ω)
        fmt.Printf("привет, %s\n", ответ)
}
$ go run hello.go
ты кто будешь?
> Йося
какое длинное имя ... целых 9 байт
привет, Йося
```

Обращаем внимание на то как Go считает длину введенной текстовой строки: байты но не символы!

И 2-й пример:

```
$ cat circle.go
package main
import ("fmt"; "os"; "math"; "strconv")

var π float64 = math.Pi

func main() {
    bufer := make([]byte, 80)
    for {
        fmt.Printf("радиус вашего круга? : ")
        длина, _ := os.Stdin.Read(bufer)
        str := string(bufer[:длина-1])
        if 0 == len(str) { break };
        радиус, err := strconv.ParseFloat(str, 64)
        if err != nil {
            fmt.Println("ошибка ввода!")
            continue
        }
        fmt.Printf("длина окружности 2*π*радиус = %f\n",
            2*π*радиус)
    }
}
$ go run circle.go
радиус вашего круга? : 10
```

13 Что совершенно естественно, учитывая что один из основных разработчиков Go Роб Пайк — был, в своё время, и разработчиком кодировки UTF-8.

```
длина окружности 2*π*радиус = 62.831853
радиус вашего круга? :
```

Функция `len()` для строки также возвращает число **байт**, но не **символов**. Если же мы захотим получить число символов, в нашем понимании, мы должны проделать некоторое преобразования, например так: `utf8.RuneCountInString(str).`

Rust

Язык программирования Rust — компилируемый и мультипарадигмальный, позиционируется многими как альтернатива C/C++. Язык был официально представлен в 2010 году. Символьный (char) тип Rust (один из «простых» типов), представляет символ Unicode (внутреннее представление данных как u32). Строка (как ещё один тип данных) *гарантированно является корректной последовательностью байтов с точки зрения UTF-8* (это фраза из документации Rust).

```
$ cat mullang.rs
fn greet_world() {
    let regions = [
        String::from("السلام عليكم"),
        String::from("Dobrý den"),
        String::from("Hello"),
        String::from("שָׁלוֹם"),
        String::from("नमस्ते"),
        String::from("こんにちは"),
        String::from("안녕하세요"),
        String::from("你好"),
        String::from("Olá"),
        String::from("Здравствуйте"),
        String::from("Hola"),
    ];

    for region in regions.iter() {
        println!("{}", &region);
    }
}

fn main() {
    greet_world();
}
```

Компиляция[14]:

```
$ rustc mullang.rs
$ ls -l mullang*
-rwxrwxr-x 1 olej olej 3911360 дек 10 22:46 mullang
-rw-r--r-- 1 olej olej     588 ноя  2 23:35 mullang.rs
$ file mullang
mullang: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=d1da0012535cf77bcbeb9a5e9018612d6e8e28fb, for GNU/Linux 3.2.0, with debug_info,
not stripped
```

Выполнение:

```
$ ./mullang
السلام عليكم
Dobrý den
Hello
שָׁלוֹם
```

14 Я специально не использую Rust менеджер `cargo`, а использую вручную компилятор `rustc` для большей прозрачности происходящего. Но и с `cargo`, как рекомендуют использовать Rust, все результаты будут те же.

```
नमस्ते
こんにちは
안녕하세요
你好
Olá
Здравствуйте
Hola
```

Остаётся интерес к тому как Rust отнесётся к объектам в коде, именованных в национальных символах в кодировке UTF-8:

```
$ cat utf8rs.rs
fn main() {
    let строка = "строка";
    println!("вывод: {}", &строка);

    let 안녕하세요 = "안녕하세요";
    println!("вывод: {}", &안녕하세요);
}
```

Здесь компиляция завершится с обильными предупреждениями:

```
$ rustc utf8rs.rs
warning: the usage of Script Group `Cyrillic` in this crate consists solely of mixed script confusables
  --> utf8rs.rs:19:9
   |
19 |     let строка = "строка";
   |         ^^^^^^
   |
   = note: `#[warn(mixed_script_confusables)]` on by default
   = note: the usage includes 'a' (U+0430), 'к' (U+043A), 'о' (U+043E), 'р' (U+0440), 'с' (U+0441), 'т' (U+0442)
   = note: please recheck to make sure their usages are indeed what you want

warning: 1 warning emitted
```

Но это только **предупреждения**, и код компилируется и **успешно** выполняется:

```
$ ls -l utf8rs
-rwxrwxr-x 1 olej olej 3888344 ноя  2 19:30 utf8rs
$ ./utf8rs
вывод: строка
вывод: 안녕하세요
```

Только хотелось бы избавиться от многочисленных предупреждений, если мы делаем это сознательно. Достигается это использованием одной из многочисленных прагм Rust (`mixed_script_confusables`):

```
$ cat utf8rs.rs
#![allow(mixed_script_confusables)]

fn main() {
    let строка = "строка";
    println!("вывод: {}", &строка);

    let 안녕하세요 = "안녕하세요";
    println!("вывод: {}", &안녕하세요);
}
```

И теперь это выглядит так:

```
$ rustc utf8rs.rs
$ ./utf8rs
```

```
вывод: строка
вывод: 안녕하세요
```

Kotlin

Язык программирования Kotlin — статически типизированный, объектно-ориентированный язык, работающий поверх Java Virtual Machine (JVM). Во многих областях (Android и др.) позиционируется как альтернатива Java. Представлен общественности в июле 2011 года.

Kotlin, как и все новые языки программирования, поддерживает символьное представление в UTF-8. Хотя, для обеспечения декларируемой многоплатформенности, может обеспечивать преобразование байтовых последовательностей в любую явно указанную кодировку — пример из документации:

```
val charset = Charsets.UTF_8
val byteArray = "Hello".toByteArray(charset)
println(byteArray.contentToString()) // [72, 101, 108, 108, 111]
println(byteArray.toString(charset)) // Hello
```

Пример, демонстрирующий представление строк в UTF-8 по умолчанию (в Linux, по крайней мере):

```
$ cat stringb.kt
fun main(args: Array<String>) {
   val x : String = args[0]
   println("строка параметров: $x - длиной ${x.length}")
   for (c in x)
      print("$c ")
   println()
   for (b in x.toByteArray())
      print("%02x ".format(b))
   println()
}
$ kotlinc stringb.kt -include-runtime -d stringb.jar
```

Программа попутно демонстрирует соотношение терминов «символ» и «байт» в UTF-8:

```
$ java -jar stringb.jar "latin string"
строка параметров: latin string - длиной 12
l a t i n   s t r i n g
6c 61 74 69 6e 20 73 74 72 69 6e 67
$ java -jar stringb.jar Здравствуйте
строка параметров: Здравствуйте - длиной 12
З д р а в с т в у й т е
d0 97 d0 b4 d1 80 d0 b0 d0 b2 d1 81 d1 82 d0 b2 d1 83 d0 b9 d1 82 d0 b5
$ java -jar stringb.jar 你好
строка параметров: 你好 - длиной 2
你 好
e4 bd a0 e5 a5 bd
$ java -jar stringb.jar السلام عليكم
строка параметров: السلام - длиной 6
ا ل س ل ا م
d8 a7 d9 84 d8 b3 d9 84 d8 a7 d9 85
```

Те, кто знакомы с основами кодирования UTF-8 для Unicode представлений, увидят знакомое: для ASCII строки (латинские символы) - 1 байт на символ, для русских символов - 2 байт на символ, для иероглифической записи - 3 байт на символ … и, в принципе, для некоторых кодировок до 4 байт на символ (хотя потенциально UTF-8 позволяет и 6 байт на символ).

Вторым примером мы проверим допустимость использования национальных алфавитов для записи непосредственно в коде имён объектов (как это было и в Go и Rust):

```
$ cat stringd2.kt
import kotlin.system.*
```

```
fun main() {
    while (true) {
        print("строка? : ")
        var ввод : String // = ""
        try { ввод = readln() }           // чтение строки из консоли
        catch (ex : RuntimeException) {   // EOF: ^D
            println("завершение работы")
            exitProcess(0)
        }
        var len = ввод.length
        if (0 == len) {                   // пустая строка
            println("завершение работы")
            exitProcess(0)
        }
        println("введена строка $ввод длиной $len")
    }
}
```

```
$ kotlinc stringd2.kt -include-runtime -d stringd2.jar
$ java -jar stringd2.jar
строка? : latin string
введена строка latin string длиной 12
строка? : Здравствуйте
введена строка Здравствуйте длиной 12
строка? : 你好
введена строка 你好 длиной 2
строка? : السلام عليكم
введена строка السلام عليكم длиной 12
строка? : السلام
введена строка السلام длиной 6
строка? :
завершение работы
```

Сравнения, поиск, сортировки и другие ...

… Как рыбы попадаются в пагубную сеть, ...

Еккл. 9:12

В этой главе мы посмотрим на некоторые из операций над текстовыми строками, из числа наиболее часто используемых — такие как сравнение, поиск, сортировки, слияния, замены … но главным образом только в той части, в которой эти операции могут отличаться от привычной ASCIIZ практики, и того где они могут преподнести неожиданности.

Самым мощным, как известно, механизмом работы с текстовыми строками является техника регулярных выражений. Но рассмотрение этой техники мы не станем затрагивать в этой главе, и прибережём его для отдельной части позже.

Операции над мультибайтными строками

Совершенно естественно, что этот основной набор операций над строками (сравнения, поиск, сортировка и все другие) выполняется для строк **широких символов** в точности так же, как и для строк традиционных **байтовых символов** (ASCII). Актуален вопрос: а как обстоит дело с этими же операциями для многобайтных представлений UTF-8? Часто для выполнения всего нескольких простых операций над символьными строками преобразование их в строки широких символов оказывается слишком расточительным…

В общем виде, корректный ответ должен состоять в том, что работать с контекстом многобайтных строк UTF-8 **нельзя**. Но, при некоторой осторожности, **некоторые** из таких операций можно (в примерах показаны `string` C++, но абсолютно то же справедливо и относительно `char[]` C):

```
string s1 = "это строка контекстного поиска",
       s2 = "строка",
       s3 = "строка";
```

1. Рассчитывать на вызов `strlen()` для получения длины строки **нельзя**, он возвращает не то значение которое вы ожидаете: `strlen()` возвращает число **байт**, а не число **символов**.
2. **Можно** присваивать (копировать) содержимое строк: `=` в C++ и `strcpy()` в C (или более безопасный вызов `strncpy()`) — копирование тупо идёт байт за байтом, без какого-либо анализа что оно там копирует, до терминального **байта** `'\0'`, который не может встретиться внутри строки независимо от используемого алфавита.
3. Сравнивать строки на эквивалентность (==) **можно**: `if(s2 = s3)` в C++ и `strcmp()` в `C ...`, потому что побайтно (даже без какого-либо символьного смысла) сравниваются на совпадение **полные последовательности** от 1 до 6 байт каждого символа.
4. Сравнивать строки лексографически (<, <=, >, >=) **нельзя**: многобайтные строки разных языков несравнимы.
5. Искать вхождение подстроки **можно**: `s1.find(s2)`, в C эквивалентно `strstr(s1, s2).`
6. Искать вхождение отдельного символа (или байта) **нельзя**: `s1.find_first_of(' ')`, `if(s1[...] == ' ') ...` , аналогично в C: `strchr(s1, ' ')`, `index(s1, ' ')`. Цифровой код разыскиваемого символа может **совпасть** с одним из байтов в последовательности 1-6 байт, представляющих символ, совершенно не обязательно с последним, что вы можете предполагать. А поиск для символов локализованного алфавита это вообще конструкция **синтаксически ошибочная**, она не пройдёт компиляцию: `'Я'` — это вовсе не один байт. Корректно в строке (точнее последовательности байт) искать можно только один единственный символ — `'\0'`, байт конца строки, он не может встречаться в последовательности байт, представляющих UTF-8 символ.
7. Сравнивать отдельные символы строки **нельзя**: отдельный символ это не числовое значение (0...255), а может быть серия из 1...6 последовательных байт.
8. Делать любые **замены** содержимого строки (подстроки на подстроку, символа на подстроку, удаление символа, удаление подстроки, …), в общем случае, **нельзя** (в отдельных случаях возможно, но очень хорошо понимая и контролируя что вы делаете).

На этом мы закончим всё относительно многобайтного представления строк UTF-8, и всё дальнейшее рассмотрение — это некоторые примеры, относящиеся к выполнению операций над `wstring` (или `wchar_t`).

Контейнеры STL широких символов

Другой интересный (не очевидный и не описанный в литературе) аспект — это использование локализованных строк в шаблонной библиотеке STL. Использование контейнерных классов STL в C++ намного упрощает запись кода. Но, предполагая работу с локализованными строками, нужно во всех определениях классов `string` заменить на `wstring`, а `char` — на `wchar_t`. Вся дальнейшая работа остаётся полностью тождественной работе с традиционными строками C/C++, с формальной (полностью эквивалентной) заменой API строк на API широких строк. При этом не забывайте про выбор в коде соответствующей локализации.

В качестве первого простейшего примера рассмотрим анализ строки на то, является ли она палиндромом. Палиндромом называется строка, которая читается одинаково слева-направо и справа-налево, например «12321». Напишем приложение, которое будет анализировать является ли вводимая **строка** палиндромом или нет. Но это приложение должно: а). правильно работать с русскоязычными строками, б). опускать все пробелы, знаки препинания и не алфавитно-цифровые символы встречающиеся в строке, в). преобразовывать буквы при сравнении к единому реестру (в большие, или малые) — пункты б). и в). не мной придуманы и обычно считаются обязательными при рассмотрении палиндромов.

```
$ cat palindrom.cc
#include <iostream>
#include <locale>
#include <unistd.h>
using namespace std;

int main(int argc, char *argv[]) {
   bool debug = argc > 1 && "debug" == string(argv[1]);
```

```
    locale::global(locale(""));
    while (true) {
       if (isatty(STDIN_FILENO) != 0)
          wcout << L"Введите тестируемую строку : ";
       wstring w(L"\n");
       getline(wcin, w);
       if (w.empty()) continue;
       if (wcin.eof()) break;
       if (0 == isatty(STDIN_FILENO))  // переадресация из файла
          wcout << w << endl;
       if (debug) wcout << L"[" << w.size() << L"]: " << w << endl;
       bool poli;
       wchar_t *pb = (wchar_t*)w.c_str(),
               *pe = pb + wcslen(pb) - 1;
       do {
          while (*pb == L' ' || !iswalnum(*pb)) pb++;
          while (*pe == L' ' || !iswalnum(*pe)) pe--;
          poli = towlower(*pb) == towlower(*pe);
          if (debug) wcout << *pb << " ? " << *pe << " = " << (poli ? "+" : "-")<< endl;
       } while (poli && ++pb <= --pe);
       wcout << L"строка " << (poli ? L"" : L"не ") << L"палиндром" << endl;
    }
}
```

Это приложение любопытно тем, что показывает близкие аналогии использования API для широких и традиционных символьных строк: `wcin.eof()`, `wstring.empty()`, `wcslen()`, `iswalnum()` и другие — это всё **прямые аналоги** общеизвестных API для `char`. Всё это сильно упрощает использование широких символов (не нужно знание каких-то новых специфических API).

```
$ ./palindrom
Введите тестируемую строку : строка
строка не палиндром
Введите тестируемую строку : Я иду с мечем судия
строка палиндром
Введите тестируемую строку : Аргентина манит негра
строка палиндром
Введите тестируемую строку : А роза упала на лапу Азора
строка палиндром
Введите тестируемую строку : На в лоб, болван
строка палиндром
Введите тестируемую строку : 404
строка палиндром
Введите тестируемую строку : saippuakivikauppias[15]
строка палиндром
Введите тестируемую строку : Sum summus mus
строка палиндром
Введите тестируемую строку : A man, a plan, a canal. Panama
строка палиндром
Введите тестируемую строку : Olson in Oslo
строка палиндром
Введите тестируемую строку : Madam, I’m Adam
строка палиндром
Введите тестируемую строку : ^C
```

Приложение сделано с некоторой избыточностью: может принимать анализируемые строки не только ручным вводом с терминала, но и переадресацией из предварительно заготовленного файла; позволяет посимвольно проследить фильтрацию и сравнение символов (отладочный режим):

```
$ ./palindrom debug < pl1.txt
```

15 Это не бессмысленный набор букв, а финское слово означающее «продавец мыла», и это самое длинное из известных слов палиндромов в мире.

```
12 2 1
[6]: 12 2 1
1 ? 1 = +
2 ? 2 = +
2 ? 2 = +
строка палиндром
A man, a plan, a canal. Panama
[30]: A man, a plan, a canal. Panama
A ? a = +
m ? m = +
a ? a = +
n ? n = +
a ? a = +
p ? P = +
l ? l = +
a ? a = +
n ? n = +
a ? a = +
c ? c = +
строка палиндром
```

Но самым изощрённым и требовательным тестом на использование методов (и функций) применительно к локализованным строкам широких символов, является, несомненно, применение **обобщённых алгоритмов** к таким строкам. Это иллюстрирует пример (файл `algo.cc`), где мы возьмём произвольный русскоязычный текст (к примеру, фрагмент из стихотворения Иосифа Бродского «Конец прекрасной эпохи») и подвергнем его разнообразным «алгоритмическим» манипуляциям:

```
$ cat algo.cc
#include <iostream>
#include <iomanip>
#include <vector>
#include <set>
#include <algorithm>

using namespace std;

inline wostream& operator <<(wostream& out, const vector<wchar_t>& obj) {
   for (auto p: obj) out << p;
   return out;
}

int main(void) {
   locale loc;
   try {
      loc = std::locale("ru_RU.utf8");
   }
   catch (std::runtime_error&) {
      loc = std::locale (loc, "", std::locale::ctype);
   }
   locale::global(loc);
   wchar_t s[] = L"В этих грустных краях всё рассчитано на зиму: сны,\n"
                 "стены тюрем, пальто; туалеты невест - белизны\n"
                 "новогодней, напитки, секундные стрелки.\n"
                 "Воробьиные кофты и грязь по числу щелочей;\n"
                 "пуританские нравы. Бельё. И в руках скрипачей -\n"
                 "деревянные грелки.\n";
   // copy & find :
   vector<wchar_t> v1(wcslen(s));
   copy(s, s + v1.size(), v1.begin());
   int nb = 0;
   for (auto is = find(v1.begin(), v1.end(), L' '); is != v1.end();
```

```
            is = find(++is, v1.end(), L' ')) nb++;
    wcout << L"в фразе пробелов " << nb << endl;
    vector<wstring> vs = {};
    wchar_t delim[] = L"\n", *state, *token = wcstok(s, delim, &state); // strtok_r()
    for (; token != NULL; token = wcstok(NULL, delim, &state))
       vs.push_back(wstring(token));
    wcout << L"в фразе строк " << vs.size() << L':'<< endl;
    for (auto x : vs) wcout << x << endl;;
    // min & max :
    auto mm = minmax_element(v1.begin(), v1.end());
    wcout << L"диапазон символов: '" << *mm.first << L"' ... '" << *mm.second << L"'" << endl;
    // fill & reverse & rotate & shuffle :
    vector<wchar_t> suv(vs[0].size());
    copy(vs[0].begin(), vs[0].end(), suv.begin());
    wcout << left << setw(14) <<  L"строка: " << suv << endl;
    reverse(suv.begin(), suv.end());
    wcout << left << setw(14) << L"реверс: " << suv << endl;
    rotate(suv.begin(), suv.begin() + suv.size() / 2, suv.end());
    wcout << left << setw(14) << L"ротация: " << suv << endl;
    random_shuffle(suv.begin(), suv.end());
    wcout << left << setw(14) << L"перетасовка: " << suv << endl;
    // set_intersection & set_difference
    set<wchar_t> sus, pns;
    for (wchar_t s: vector<wchar_t>(vs[0].begin(), vs[0].end())) sus.insert(s);
    for (wchar_t s: vector<wchar_t>(vs[1].begin(), vs[1].end())) pns.insert(s);
    vector<wchar_t> outi(300), outd(300);
    auto ret = set_intersection(sus.begin(), sus.end(), pns.begin(), pns.end(), outi.begin());
    wcout << L"общих литер " << (ret - outi.begin()) << L" : " << outi << endl;
    ret = set_difference(sus.begin(), sus.end(), pns.begin(), pns.end(), outd.begin());
    wcout << L"уникальных литер " << (ret - outd.begin()) << L" : " << outd << endl;
}
```

Выполнение:

```
$ ./algo
в фразе пробелов 31
в фразе строк 6:
В этих грустных краях всё рассчитано на зиму: сны,
стены тюрем, пальто; туалеты невест - белизны
новогодней, напитки, секундные стрелки.
Воробьиные кофты и грязь по числу щелочей;
пуританские нравы. Бельё. И в руках скрипачей -
деревянные грелки.
диапазон символов: '
' ... 'ё'
строка:       В этих грустных краях всё рассчитано на зиму: сны,
реверс:       ,ынс :умиз ан онатичссар ёсв хяарк хынтсург хитэ В
ротация:      ёсв хяарк хынтсург хитэ В,ынс :умиз ан онатичссар
перетасовка:  хсыуВ гаыриохва , мнятрнстнасч: тн р и эк изсхуёса
общих литер 14 :  ,авзимнорстуы
уникальных литер 9 : :Вгкхчэяё
```

Сортировки

Алгоритмы сортировок — это настолько хорошо изученный, описанный … и наскучивший всем класс задач, что первоначально именно эту главу планировалось вообще не включать в текст. Это именно тот класс задач, которыми, по бедности воображения, преподаватели замучили студентов...

С другой стороны, именно алгоритмы сортировки дают отличную почву для сравнений и анализа. Поэтому эти алгоритмы любопытно рассмотреть с позиции естественной (лексографической) сортировки именно не англоязычных строк.

Для простоты будем мы сортировать только по возрастанию (поменять порядок сортировки, при

необходимости, элементарно).

```
$ cat sort.cc
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

typedef void (sort_func)(vector<wstring>&);

wostream& operator <<(wostream& wstr, vector<wstring>& v) { // отладка-контроль
   wstr << L"[ ";
   for(auto x : v) wstr << x << L" ";
   return wstr << L"]";
}

int main(int argc, char *argv[]) {
   locale::global(locale(""));
   // предварительные объявления функций сортировки:
   sort_func sort1, sort2, sort3, sort4, sort5, sort6;
   sort_func* variants[] = {
      sort1, sort2, sort3, sort4, sort5, sort6
   };
   vector<wstring> vect = {
      L"Фёдоров", L"Петров", L"Сидоров", L"Иванов", L"Чапаев", L"Фурманов"
   };
   wcout << vect << endl
         << L"-------------------------------------------------" << endl;
   for (unsigned i = 0; i < sizeof(variants) / sizeof(variants[0]); i++) {
      vector<wstring> vcopy(vect);
      variants[i](vcopy);
      wcout << vcopy << endl;
   }
}

//-----------------------------------------------------------------------------------------
// Быстрая сортировка. Сложность в среднем O(n*log(n)), но в худшем случае O(n*n)
void sort1(vector<wstring>& v) {
   sort(v.begin(), v.end());
}
//-----------------------------------------------------------------------------------------
bool sort_function(wstring& f, wstring& s) { return f < s; }

void sort2(vector<wstring>& v) { // использование функции как предиката
   sort(v.begin(), v.end(), sort_function);
}
//-----------------------------------------------------------------------------------------
struct sort_class {   // функтор сравнения
   bool operator()(wstring& f, wstring& s) { return f < s; }
};

void sort3(vector<wstring>& v) {
   sort(v.begin(), v.end(), sort_class());
}
//-----------------------------------------------------------------------------------------
// Сортировка подпоследовательности. Гарантированная сложность O(n*log(n)) в любом случае.
// Обычно сортировка в куче выполняется в 2-5 раз медленнее быстрой сортировки sort().
void sort4(vector<wstring>& v) {
   partial_sort(v.begin(), v.end(), v.end());
}
//-----------------------------------------------------------------------------------------
```

```
// Сортировка слиянием. Сложность O(n*log(n)) или O(n*log(n)*log(n)), если без дополнительной
памяти
void sort5(vector<wstring>& v) {
   stable_sort(v.begin(), v.end());
}
//-----------------------------------------------------------------------------------------
// Сортировка в куче (heap) - вызывают функции, непосредственно работающие с кучей
// (то есть с бинарным деревом, используемым в реализации этих алгоритмов).
// Сложность O(n+n*log(n))
void sort6(vector<wstring>& v) {
   make_heap(v.begin(), v.end());
   sort_heap(v.begin(), v.end());
}
//-----------------------------------------------------------------------------------------
```

Здесь показана работа 4-х предустановленных в библиотеке C++ алгоритмов сортировки:

```
$ ./sort
[ Фёдоров Петров Сидоров Иванов Чапаев Фурманов ]
------------------------------------------------
[ Иванов Петров Сидоров Фурманов Фёдоров Чапаев ]
[ Иванов Петров Сидоров Фурманов Фёдоров Чапаев ]
[ Иванов Петров Сидоров Фурманов Фёдоров Чапаев ]
[ Иванов Петров Сидоров Фурманов Фёдоров Чапаев ]
[ Иванов Петров Сидоров Фурманов Фёдоров Чапаев ]
[ Иванов Петров Сидоров Фурманов Фёдоров Чапаев ]
```

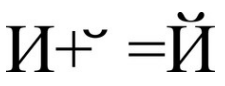
Варианты 2 и 3 используют в качестве предиката сравнения функцию и функциональный объект, соответственно, и демонстрируют случаи, когда мы можем произвольно менять порядок сортировки (по возрастанию, по убыванию, или вообще по любому произвольному критерию).

Примечание: Обратите внимание на позицию литеры 'Ё' в лексографической последовательности Unicode, то, как в очередной раз удружили русскому алфавиту западные «партнёры». Для уточнения проделаем ещё один тест:

```
void test06(void) {
   wchar_t arr[] = { L'Ё', L'А', L'И', L'Й', L'Я', L'а', L'и', L'й', L'я', L'ё' };
   locale::global(locale(""));
   for (unsigned i = 0; i < sizeof(arr) / sizeof(arr[0]); i++)
      wcout << arr[i] << L"->" << hex << (unsigned)arr[i] << L" | ";
   wcout << endl;
}

$ ./unicode++ 6
--------------------------------------
Ё->401 | А->410 | И->418 | Й->419 | Я->42f | а->430 | и->438 | й->439 | я->44f | ё->451 |
--------------------------------------
```

Отметьте, что заглавная буква 'Ё' **предшествует** всему остальному алфавиту, а малая 'ё' — **следует** за всем остальным алфавитом. Это достаточно существенный «подарок», его нужно иметь в виду, и можно поиметь весьма неожиданные ошибки. Но этот артефакт, при необходимости, можно легко устранить (если его предполагать), корректируя предикат сравнения, как показано было в вариантах 2 и 3 показанного выше примера.



Кроме того, есть ещё одна тонкая особенность, связанная с отображением непротяжённых (модифицирующих) символов (знаков ударений, диакритических знаков и др.), предлагаемых в Unicode (как вариант). Часто в качестве примера приводится русский символ 'Й'.

Но в тесте выше мы видим, что в представлениях Linux символ 'Й' отображается как **монолитный** символ, имеющий свой код, отличающийся от 'И'. И проблема использования модифицирующих символов, опять же, это больше проблема Windows, чем Unicode представления, и останавливаться на ней мы не станем.

Литература и сетевые ресурсы

1. Брайан У. Керниган, Деннис М. Ритчи, Язык программирования C, 3-е издание

   https://monster-book.com/yazyk-programmirovaniya-c-kernigach-ritchi
2. Юникод - https://ru.wikipedia.org/wiki/Юникод
3. Кодовая страница - https://ru.wikipedia.org/wiki/Кодовая_страница
4. Кириллица в Юникоде - https://ru.wikipedia.org/wiki/Кириллица_в_Юникоде
5. А.Гриффитс, GCC. Полное руководство. Platinum Edition. М.: «ДиаСофт», 2004, ISBN 966-7992-33-0, стр. 624
6. Черновик стандарта C99 (ISO/IEC 9899:TC3 Septermber 7, 2007)

   https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf
7. У.Ричард Стивенс, Стивен А.Раго, UNIX. Профессиональное программирование, 3-е издание, СПб.: «Символ-Плюс», 2013, ISBN: 978-5-93286-216-2, стр. 1104
8. The Open Group Base Specifications Issue 7, 2018 edition IEEE Std 1003.1™-2017 (Revision of IEEE Std 1003.1-2008), стандарты POSIX https://pubs.opengroup.org/onlinepubs/9699919799/
9. N1570 Committee Draft — April 12, 2011 последний черновик стандарта C11 25 апреля 2011г. https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf
10. std::wcout - https://cplusplus.com/reference/iostream/wcout/
11. Примеры палиндромов - https://ru.wikipedia.org/wiki/Палиндром
12. Алгоритмы http://www.redov.ru/kompyutery_i_internet/rukovodstvo_po_standartnoi_biblioteke_shablonov_stl/p10.php
13. Обобщенные алгоритмы в алфавитном порядке - http://cpp.com.ru/lippman/c21.html
14. UTF-16 - https://ru.wikipedia.org/wiki/UTF-16
15. UTF-8 - https://ru.wikipedia.org/wiki/UTF-8
16. UTF-8, a transformation format of ISO 10646 - регламент стандарта UTF-8

    https://www.rfc-editor.org/rfc/rfc3629
17. The Go Programming Language Specification, Version of June 29, 2022

    https://go.dev/ref/spec
18. Standard library, Version: go1.19.4 - https://pkg.go.dev/std
19. The Rust Programming Language - https://doc.rust-lang.org/book/second-edition/
20. Rust на примерах - https://doc.rust-lang.ru/stable/rust-by-example/index.html
21. Руководство по языку Kotlin - https://kotlinlang.ru/

Часть 2. Регулярные выражения в программном коде

Когда же число людей очень увеличилось, они решили построить город и в нём башню высотой до небес, чтобы приобрести себе славу.

Но Богу не угодно было их намерение. Он смешал язык строителей так, что они начали говорить на разных языках и перестали понимать друг друга.

Кн. Быт. 11:1 – 9

Регулярные выражения являются способом описания текстовых шаблонов для сопоставления. Они являются фундаментальной базой таких инструментов GNU как `grep`, `egrep`, `sed`, `awk` и редакторов `ed`, `vi`, `vim`, `Emacs`. Из языков программирования регулярные выражения исторически достигли своего полного развития в Perl, и в настоящее время включены практически во все современные языки программирования: Perl, Ruby, Go, ...

Но языки C (особенно) и C++ не обладают развитыми средствами обработки символьной информации, поэтому реализация регулярных выражений их средствами противопоказана. Однако (первоначально из-за необходимости реализации базовых GNU утилит) обработка регулярных выражений (в той или иной мере полноты) в них также реализована. И может быть использована в собственном коде.

И в завершение предисловия… чего нет этом тексте:

Предметом данных заметок ни в коем случае не является **составление и использование** шаблонов регулярных выражений — на этот предмет есть великое множество отличных публикаций. Точно так же, не станем мы и затрагивать вопросы внутренней **реализации обработки** регулярных выражений (исключая минимальное упоминание, важное для дальнейшего изложения). Точно так же, совершенно не будут затронуты вопросы **различия в синтаксисе** разных диалектов (стандартов) языка регулярных выражений.

Интересом данных заметок, собственно, будут составлять только два вопроса:

1. Как развернуть (что-то инсталлировать и как-то настроить) API для работы с регулярными выражениями в инфраструктуре того или иного рассматриваемого языка программирования;
2. В какой мере применимы эти механизмы техники регулярных выражений к локализованной (не англоязычной) текстовой информации (когда сопоставляемая строка, или шаблон в форме регулярного выражения, или оба содержат иноязычную строку). Это и является основным предметом настоящих заметок.

Мне бы было крайне интересно рассмотреть поведение операций регулярных выражений на текстах, скажем, на китайском языке … или на арабском. Но, поскольку я не знаю ни того, ни другого, и не могу оценить корректность получаемых результатов — то рассматривать это я буду на русскоязычном материале, и надеяться что это естественным образом переносится **на все** языки мира.

Общие замечания относительно регулярных выражений

Некоторые люди во время решения некой проблемы думают: «Почему бы мне не использовать регулярные выражения?». После этого у них уже две проблемы…

Jamie Zawinski

Прежде всего, стоит остановиться на том факте (на который не часто обращают внимание), что регулярные выражения появились не как ещё один инструмент для практики «от сохи» (как, например, язык BASIC), а как предмет изучения, имеющий в фундаменте основу из теоретических разделов абстрактной математики [1]:

Истоки регулярных выражений лежат в теории автоматов, теории формальных языков и классификации формальных грамматик по Хомскому.

Эти области изучают вычислительные модели (автоматы) и способы описания и классификации формальных языков. В 1940-х гг. Уоррен Маккалок и Уолтер Питтс описали нейронную систему, используя простой автомат в качестве модели нейрона.

Математик Стивен Клини позже описал эти модели, используя свою систему математических обозначений, названную «регулярные множества».

Кен Томпсон встроил их в редактор QED, а затем в редактор ed под UNIX. С этого времени регулярные выражения стали широко использоваться в UNIX и UNIX-подобных утилитах, например в expr, awk, Emacs, vi, lex, Perl.

То есть, теория регулярных выражений, как математическая абстракция, появилась **раньше** самого первого компьютера фон Неймана в реале[16]. А обработка сопоставления с регулярным выражением — это хорошо известная в теории работа конечного автомата[17].

Громоздкость показанного далее рассмотрения регулярных выражений существенно осложняется тем, что стандарт POSIX предусматривает 2 вида (уровня) синтаксиса регулярных выражений: базовый (Basic Regular Expression, BRE) и расширенный (Extended Regular Expressions, ERE). Программы типа `grep`, `sed` и строчный редактор `ed` по умолчанию используют **базовый** уровень регулярных выражений. Программы типа `egrep` и `awk`, и язык `Perl` используют **расширенные** регулярные выражения (на самом деле тонких диалектов ещё больше, как мы перечислим вскоре). Выбранный режим обработки может изменяться, например, опциями запуска для утилиты `grep`:

```
$ grep –help
...
  -E, --extended-regexp     ШАБЛОН – расширенное регулярное выражение (ERE)
  -F, --fixed-regexp        ШАБЛОН – строки, разделённые символом новой строки
  -G, --basic-regexp        ШАБЛОН – простое регулярное выражение (BRE)
  -P, --perl-regexp         ШАБЛОН – регулярное выражения языка Perl
...
```

При сопоставлении с образцом в программном коде используемый уровень будет определяться набором **флагов**.

Кроме того, может быть достаточно много уточняющих деталей работы, например, учитывать или нет регистр символов. Со стороны утилит это тоже определяется опциями командной строки, например для `grep: -i`. В программном коде такие уточнения также должны делаться набором всё тех же **флагов**, объединённых по «ИЛИ».

В результате, конкретный программный код обрастает большим набором разнородных флагов, которые в разных пакетах (библиотеках) определяются разными множествами возможностей, например так:

```
RE_BACKSLASH_ESCAPE_IN_LISTS | RE_CHAR_CLASSES |
RE_NO_BK_BRACES | RE_NO_BK_PARENS | RE_NO_BK_VBAR | RE_INTERVALS
```

Поэтому добиться полностью **эквивалентного поведения** различных экземпляров кода, использующих **разные** библиотеки для работы с регулярными выражениями, порой оказывается достаточно хлопотно, а временами и просто невозможно.

Как написал в своей книге Майкл Фицджеральд (см. библиографию в конце):

Большинство из указанных реализаций регулярных выражений в чем-то сходны, а в чем-то различаются. Я не могу подробно обсудить все отличия в столь маленькой книге, но о многих расскажу. Любые попытки задокументировать все различия между всеми реализациями наверняка привели бы меня в больницу.

Внутренняя реализация обработки регулярных выражений в общем случае достаточно сложна, хорошо изучена и опирается на теорию конечных автоматов [2]:

Квантификаторы и объединение вызывают к жизни такое количество путей срабатывания сложных шаблонов, что использование "обычного" алгоритма невозможно. Должен быть применен более действенный подход. Лучший путь — построить автомат

16 Нужно признать, хотя это и не любят афишировать, что для изощрённого использования регулярных выражений хорошо бы иметь достаточно глубокое математическое образование.

17 Мы не станем углубляться в эту тему, но это следует держать в уме.

и смоделировать его работу. Для описания поискового шаблона, заданного регулярным выражением, вы можете использовать недетерминированный и детерминированный конечные автоматы.

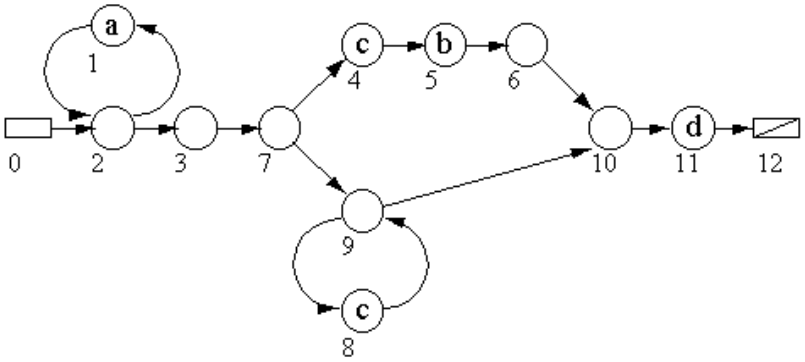


Рис. Недетерминированный конечный автомат для шаблона `a*(cb|c*)d`

Рассмотрение реализации никоим образом не входит в круг нашего рассмотрения (это отдельный серьёзный предмет), но из общего упоминания отметим то, что первой фазой отождествления всегда является **построение** конечного автомата. А это означает, что мы (т. е. авторы языковой библиотеки) сможем смоделировать в каждом случае такой автомат под шаблон отождествления именно на том языке, код которого рассматривается. И такая модель может быть **предварительно скомпилирована** для последующего многократного применения к **разным** входным строкам. Это окажется весьма важным в дальнейшем рассмотрении…

Как это работает в утилитах GNU

Прежде, чем рассматривать как это делается в программном коде, проследим как отдельные (и весьма многочисленные) утилиты GNU, использующие регулярные выражения, работают с локализованными строками…

```
$ export LANG=en_US.utf8
$ locale | head -n2
LANG=en_US.utf8
LC_CTYPE="en_US.utf8"

$ echo "слово слава слива" | grep сл
слово слава слива
$ echo "слово слава слива" | grep сл[о,а]
слово слава слива
$ echo "слово слава слива" | grep сл.ва
слово слава слива
```

(В этом разделе там, где прошедшие сопоставление подстроки отмечаются в терминале цветом, эти результаты показаны жирным шрифтом.)

Редактор `sed`:

```
$ echo "слово слава слива" | sed  s/л/к/g
сково скава скива
$ echo "слово слава слива" | sed  s/л[о,а,и]/кр/g
скрво скрва скрва
$ echo "раз два три" | sed  s/' './_/g
раз_ва_ри
$ echo "раз два три четыре" | sed "/[^ ]* [ ]*[^ ]* [ ]*\([^ ]*\).*/s//\1/"
три
```

Теперь AWK (в части работы с регулярными выражениями):

```
$ echo "раз два три" | awk '{ print $1, $3 }'
раз три
```

Здесь показаны далеко не все утилиты UNIX / POSIX / Linux, предназначенные для использования регулярных выражений в командной строке и из скриптов `shell` (`bash`, `dash` и др.). Всё это — давно и постоянно используемая скриптовая техника UNIX. Детали такого использования выходятт за рамки наших интересов в данном тексте...

Итогом этой краткой преамбулы мы демонстрируем то, что классические GNU утилиты показывают корректную работу с русскоязычными строками в UTF-8, и даже когда локаль терминальной сессии искусственно установлена другая, как в показанном примере, в `en_US.utf8`.

Но:

```
$ export LANG=en_US.iso88591
$ locale | head -n2
LANG=en_US.iso88591
LC_CTYPE="en_US.iso88591"
$ echo вашшче | egrep "ш{2,}"
вашшче
```

Как это работает из программного кода

Кроме использования из утилит системы, регулярные выражения могут быть использованы **изнутри** программного кода. Использование техники регулярных выражений каждый раз проходит через две последовательные фазы (которые не так отчётливо видны при работе из командной строки):

1. Разбор текстового шаблона (образца) с которым будет производиться сопоставление, построение конечного автомата;
2. Само сопоставление целевого текста с образцом.

По трудоёмкости (времени выполнения) первая фаза (часто неявная) может быть много значительнее второй, которая может оказаться проще.

Поскольку из программного кода чаще всего приходится сопоставлять с одним и тем же образцом различные текстовые цели (в цикле), то чаще всего, в программном коде эти фазы разделяются: образец предварительно **компилируется** во внутреннюю форму, после чего эта форма может многократно применяться для сопоставления. Но используются также и API которые делают это онлайн, как единая неделимая операция (напоминая то как это делают консольные утилиты).

Большая часть рассмотрения будет посвящена тому как это делается в классических (по крайней мере для Linux) языках C и C++. В C и C++ реализованы и доступны для использования несколько различных реализаций обработки регулярных выражений. Но **логика** каждой из реализаций остаётся неизменной:

- Заданный шаблон (образец) сопоставления предварительно, как правило, **компилируется**;
- После чего с компилированным шаблоном могут **сопоставляться** сколь угодно текстовых выражений;

Позже мы кратко рассмотрим как это же делается в новых, современных языках программирования: Python, Go, Rust, Kotlin...

Регулярные выражения в C

> *«Вселенная – некоторые называют её Библиотекой – состоит из огромного, возможно, бесконечного числа шестигранных галерей, с широкими вентиляционными колодцами, ограждёнными невысокими перилами. Из каждого шестигранника видно два верхних и два нижних этажа – до бесконечности»*
>
> *Хорхе Луис Борхес «Вавилонская Библиотека»*

В C для работы с локализованными строчными переменными предусмотрен тип `wchar_t` (эквивалент `char`) и представление локализованных строк как массивов `wchar_t[]`. Особенностью C является то, что среди **многих** альтернативных инструментов работы с регулярными

выражениями **нет** таких, которые работали бы со строками в представлении `wchar_t*`. И относится сказанное как к встроенным библиотекам GNU, так и к пакетам от сторонних производителей.

Но предметом настоящих заметок является именно исследование того, как происходит (возможна) сопоставление с образцами именно локализованных иноязычных строк. Возможность использования инструментов регулярных выражений связана с тем, что при сопоставлении **строк** `char[]`, выраженных в Unicode кодировке UTF-8 (как это имеет место в Linux), сравниваются не осмысленные **символы**, а (бессмысленные) последовательности **байт**, представляющих многобайтные изображения символов (от 1-го до 6-ти байт кодирования UTF-8 на символ). С таким же успехом могло бы искаться вхождение «текстовой строки» `"\1\2\3"`. При этом важно, как будет показано далее, что результатом сопоставления являются просто байтовые **позиции** начала и конца найденного соответствия.

И теперь мы можем перейти к последовательному рассмотрению различных доступных реализаций.

Одной из первых, ранних реализаций GNU использовалась заимствованная ещё из OS Solaris реализация, определения которой находятся <regexp.h>, функции: `compile()`, `advance()`, `step()`. Ещё до сих пор в публикациях встречаются описания этой реализации, которые выглядят как-то так:

```
$ cat regexp.c
#include <stdio.h>
#define INIT      char *sp = instring;
#define GETC()    (*sp++)
#define PEEKC()   (*sp)
#define UNGETC(c) (--sp)
#define RETURN(c) return c;
#define ERROR(c)  printf("error %d\n", c);
#include <regexp.h>

#define SIZE 80
int main(int argc, char *argv[]) {
   char pattern[SIZE] = "s",
        buf[SIZE] = "test";
   if (!compile(*argv, pattern, &pattern[SIZE], '\0')) {
      return 1;
   }
   if (!step(buf, pattern))
      printf("no match\n");
   else {
      for (char* p = loc1; p < loc2; p++) // где loc1 и loc2 — глобальные переменные пакета
         printf( "%c", *p );
      printf( "\n" );
   }
   return 0;
}
```

Всё это (как видно из заголовка примера) построено на макроопределениях, и в описаниях (`man`) сказано:

> *У этого файла неприятный интерфейс. Программы, которые включают данный файл, перед оператором #include <regexp.h> должны содержать определение пяти макросов, перечисленных ниже. Макросы используются функцией compile.*

Здесь всё просто для понимания общей логики работы с регулярными выражениями, в этом полезность такого примера. Но не ведитесь на эту простоту...

Ещё ко времени подготовки первых редакций этого текста, на уровне 2016 года, такой код компилировался без ошибок, но с такими вот характерными предупреждениями, которые заставляют задуматься:

```
$ gcc -Wall regexp.c -o regexp
In file included from regex0.c:9:0:
```

```
/usr/include/regexp.h:31:2: предупреждение: #warning "<regexp.h> will be removed in the next
release of the GNU C Library." [-Wcpp]
 #warning "<regexp.h> will be removed in the next release of the GNU C Library."
  ^
/usr/include/regexp.h:32:2: предупреждение: #warning "Please update your code to use <regex.h>
instead (no trailing 'p')." [-Wcpp]
 #warning "Please update your code to use <regex.h> instead (no trailing 'p')."
  ^
```

Объяснения мы находим в комментариях самого заголовочного файла <regexp.h>:

> *The contents of this header file were standardized in the Single Unix Specification, Version 2 (1997) but marked as LEGACY; new applications were already being encouraged to use <regex.h> instead. POSIX.1-2001 removed this header.*

И к сегодня, в конечном итоге, даже попытка компиляции подобного кода завершится грубыми ошибками с сообщениями вида:

```
$ gcc regexp.c
In file included from regexp-.c:10:
/usr/include/regexp.h:30:2: error: #error "The GNU C Library no longer implements <regexp.h>."
   30 | #error "The GNU C Library no longer implements <regexp.h>."
      |  ^~~~~
/usr/include/regexp.h:31:2: error: #error "Please update your code to use <regex.h> instead (no
trailing 'p')."
   31 | #error "Please update your code to use <regex.h> instead (no trailing 'p')."
      |  ^~~~~
```

Это **устаревший механизм**, пришедший в GNU ещё из OS Solaris, и сохранившийся много лет только из соображений синтаксической совместимости. Он представляет только исторический интерес … даже если вы столкнётесь со ссылками на него в публикациях и обсуждениях, что всё ещё имеет место.

Следующий механизм, как и сообщают сообщения из `<regexp.h>`, определён в файле `<regex.h>` — это и есть механизм («родной» для POSIX API), введенный POSIX.1-2001. Логика работы всё та же: предварительная компиляция образца (шаблона), после чего сколько угодно раз сопоставления с образцом. Его можно использовать двояким образом, оба варианта показаны ниже (все примеры, по возможности, записаны так, чтобы эквивалентные вещи обозначались теми же именами). Первый вариант использует **расширения** GNU для <regex.h>:

```
$ cat regex1.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#define __USE_GNU
#include <regex.h>

int main(int argc, char *argv[]) {
#define SIZE 80
   char buf[SIZE] = "123,4567,898709",
        pattern[SIZE] = "^([^,]*),([^,]*),([^,]*)$";
   if (argc > 1) strncpy(pattern, argv[1], SIZE - 1);
   struct re_pattern_buffer *weight;
   weight = (struct re_pattern_buffer*)malloc(sizeof(struct re_pattern_buffer));
   if (!weight) {
      printf("allocate error %m\n");
      return 1;
   }
   re_set_syntax(RE_BACKSLASH_ESCAPE_IN_LISTS | RE_CHAR_CLASSES |
      RE_NO_BK_BRACES | RE_NO_BK_PARENS | RE_NO_BK_VBAR | RE_INTERVALS);
   const char *err = re_compile_pattern(pattern, strlen(pattern), weight);
   if (err) {
      printf("compile error: %s\n", err);
      free(weight);
      return 2;
   }
   struct re_registers regs;
```

```
    memset(&regs, 0, sizeof(regs));
    while (fgets(buf, sizeof(buf) - 1, stdin)) {
       if (buf[strlen(buf) - 1] == '\n') buf[strlen(buf) - 1] = '\0';
       if (0 == strlen(buf)) break; // выход
       int p = re_match(weight, buf, strlen(buf), 0, &regs);
       if (p <= 0) {
          printf("no match found\n--------------------\n");
          continue;
       }
       for (int c = 0; (c < p) && (regs.start[ c ] >= 0); c++) {
          printf("%d/%d : ", regs.start[c], regs.end[c]);
          for (int i = regs.start[c]; i < regs.end[c]; i++)
             printf("%c", buf[i]);
          printf("\n");
       }
       printf("--------------------\n");
    }
    free(weight);
    return 0;
}
```

Сборка … и убеждаемся что здесь для сборки не требуются никакие (кроме `libc`) внешние библиотеки :

```
$ gcc -Wall regex1.c -o regex1
$ ldd regex1
        linux-vdso.so.1 (0x00007ffcf8df7000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007fcc31992000)
        /lib64/ld-linux-x86-64.so.2 (0x00007fcc31bb1000)
```

И некоторая минимальная проверка выполнением[18] (примеры которые мы будем использовать далее для сравнения вариантов):

```
$ ./regex1 "^([^,]*),([^,]*),([^,]*)$"
123,4567,898709
0/15 : 123,4567,898709
0/3 : 123
4/8 : 4567
9/15 : 898709
--------------------
qwerty
no match found
--------------------
a,b,c
0/5 : a,b,c
0/1 : a
2/3 : b
4/5 : c
--------------------
слово,ещё слово,снова слово
0/50 : слово,ещё слово,снова слово
0/10 : слово
11/28 : ещё слово
29/50 : снова слово
--------------------
й,ё,Ё
0/8 : й,ё,Ё
0/2 : й
```

18 Особую тщательность дополнительно нужно проявлять при тестировании (всех и любых примеров) на символы: 'Ё', 'ё', 'Й', 'й'. Связано это с тем, что в таблицах Unicode 'Ё' лексографически предшествует всему русскому алфавиту, а 'ё' — завершает этот алфавит. А 'Й' и 'й' могут порождать проблемы непротяжённых (модифицирующих) символов … но это более характерно Windows и Unicode кодированию UTF-16.

```
3/5 : ё
6/8 : Ё
--------------------
$ ./regex1 "([0-9]{1,3}).([0-9]{1,3}).([0-9]{1,3}).([0-9]{1,3})"
192.168.1.3
0/11 : 192.168.1.3
0/3 : 192
4/7 : 168
8/9 : 1
10/11 : 3
```

Следующий представленный вариант использует ту же библиотеку <regex.h>, но уже в её POSIX нотации:

```
$ cat regex2.c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <regex.h>

int main(int argc, char *argv[]) {
#define SIZE 80
   char buf[SIZE],
        errbuf[SIZE] = "",
        pattern[SIZE] = "[0-9]";
#define MATCHSIZE 20
   regmatch_t regs[MATCHSIZE];
   regex_t re, *pre = &re;
   if (argc > 1) strncpy(pattern, argv[1], SIZE - 1);
   int err = regcomp(pre, pattern, REG_ICASE | REG_EXTENDED);
   if (err) {
       regerror(err, pre, errbuf, SIZE);
       printf("%s\n", errbuf);
       return 1;
   }
   while (fgets(buf, sizeof(buf) - 1, stdin)) {
      if (buf[strlen(buf) - 1] == '\n') buf[strlen(buf) - 1] = '\0';
      if (0 == strlen(buf)) break; // выход
      if (REG_NOMATCH == regexec(pre, buf, MATCHSIZE, regs, 0)) {
         printf("no match found\n--------------------\n");
         continue;
      }
      for (int c = 0; regs[c].rm_so != -1; c++) {
         printf("%d/%d : ", regs[c].rm_so, regs[c].rm_eo);
         for (int i = regs[c].rm_so; i < regs[c].rm_eo; i++)
            printf("%c", buf[i]);
         printf("\n");
      }
      printf("--------------------\n");
   }
   regfree(pre);
   return 0;
}
```

Примечание: Здесь, как и в предыдущем варианте, нет необходимости (как это всегда бывает в коде C обязательно при работе с широкими **локализованными** строками wchar_t[]) устанавливать соответствующую языковую локаль:

```
setlocale( LC_ALL, "" );
```

Поиск и сравнения производятся не в терминах осмысленных **символов** иностранного языка (многобайтных UTF-8), а в терминах обессмысленных **байт** в последовательностях UTF-8 представлений символов (но это, тем не менее, работает).

Сборка, как и в предыдущем случае, не требует ничего дополнительно определять в опциях:

```
$ gcc -Wall regex2.c -o regex2
$ ./regex2 "^([^,]*),([^,]*),([^,]*)$"
123,4567,898709
123,4567,898709
0/15 : 123,4567,898709
0/3 : 123
4/8 : 4567
9/15 : 898709
--------------------
qwerty
no match found
--------------------
a,b,c
0/5 : a,b,c
0/1 : a
2/3 : b
4/5 : c
--------------------
й,ё,Ё
0/8 : й,ё,Ё
0/2 : й
3/5 : ё
6/8 : Ё
--------------------
слово,ещё слово,снова слово
0/50 : слово,ещё слово,снова слово
0/10 : слово
11/28 : ещё слово
29/50 : снова слово
--------------------
$ ./regex2 "([0-9]{1,3}).([0-9]{1,3}).([0-9]{1,3}).([0-9]{1,3})"
это подсеть 192.168.1.0
22/33 : 192.168.1.0
22/25 : 192
26/29 : 168
30/31 : 1
32/33 : 0
--------------------
```

Результат сопоставления (в обоих вариантах и показанных далее) представляется как **массив** совпадений (пусть и выраженных по-разному), 1-й элемент которого описывает общее соответствие, а последующие — это совпадения последовательных отмеченных подвыражений (\1, \2, \3, … в терминологии регулярных выражений):

```
$ ./regex1 "([0-9]*).([0-9]*).([0-9]*).([0-9]*)"
192.168.1.3
0/11 : 192.168.1.3
0/3 : 192
4/7 : 168
8/9 : 1
10/11 : 3
--------------------
```

И, для дополнительного подтверждения сказанного, проделаем операции в корейской языковой локали (лишь бы эта локаль была с UTF-8 кодировкой Unicode):

```
$ export LANG=ko_KR.utf8
$ locale | head -n2
LANG=ko_KR.utf8
LC_CTYPE="ko_KR.utf8"
$ echo "русский текст на корейский манер"
```

```
русский текст на корейский манер
$ echo вашшче | egrep "ш{2,}"
вашшче
$ ./regex2 арт
аппарат апартамент аплодисмент
19/25 : арт
--------------------
```

Но такое («тупое» побайтное) сопоставление имеет побочные эффекты и требует хорошего понимания происходящего: можно искать в строках совпадений многобайтных последовательностей **подстрок**, но нельзя корректно выполнять всё, что формулируется в терминологии **символов**, трактуемых как единичные **байты**:

```
$ ./regex2 "Ё."
яываэё Ёйвап
13/16 : Ё
--------------------
```

Конец сопоставленной последовательности в данном случае должен был бы завершаться на 17-м **байте**, а не 16-м.

В итоге, мы можем считать, что операции сопоставления с регулярными выражениями вполне можно использовать в коде языка C, но с ограничением некоторых случаев сопоставления, и при хорошем понимании что и как при этом происходит.

Но такой вот POSIX API является **не единственной** альтернативой для использования из-под C ...

PCRE

Следующий инструмент в нашем обзоре, широко используемый **пакет** (его библиотеки) — это PCRE (**P**erl **C**ompatible **R**egular **E**xpressions). Этот механизм использует совместимый с Perl **синтаксис** регулярных выражений. Считается, что синтаксис PCRE намного мощнее и гибче, чем любой из вариантов регулярных выражений POSIX и чем у многих других библиотек регулярных выражений. Но мы, как говорилось ранее, не будем рассматривать различия в синтаксисах диалектов регулярных выражений (которых великое множество), а сосредоточимся только на том как **использовать** тот или иной механизм.

Но и здесь мы находим 2 линии развития:

- PCRE, который развивается с 1997, и был завершён 15 июня 2021 на версии 8.43 (предположительно это будет последняя версия этой линии);
- PCRE2, форк PCRE, с пересмотренным API и несколько изменённым синтаксисом регулярных выражений, который был выпущен в 2015 году и активно развивается по настоящее время;

В любом из этих вариантов использования мы должны начать с инсталляции соответствующего инструментария. И начнём с PCRE...

Инсталляция (из стандартного репозитория, в данном случае это Mint 20.3):

```
$ apt show libpcre3 libpcre3-dev
...
$ aptitude search pcre3 | grep ^i
i  libpcre3 - устаревшая библиотека поддержки регулярных выражений, совместимых с Perl5 — файлы времени исполнения
i  libpcre3-dev - Old Perl 5 Compatible Regular Expression Library - development files
i A libpcre32-3 - Old Perl 5 Compatible Regular Expression Library - 32 bit runtime files
$ aptitude show libpcre3
Пакет: libpcre3
Версия: 2:8.39-12ubuntu0.1
...
Описание: устаревшая библиотека поддержки регулярных выражений, совместимых с Perl5 — файлы времени исполнения
Библиотека предоставляет функции для работы с регулярными выражениями. Синтаксис и семантика выражений сделаны максимально похожими на регулярные выражения языка Perl 5.
```

```
В новых пакетах следует использовать только более новые пакеты pcre2, а уже существующие пакеты
необходимо постепенно перевести на использование pcre2.
Пакет содержит динамически загружаемую версию библиотеки.
```

Используя этот инструментарий приложение, подобное предыдущим, может быть сделано так:

```
$ cat regex3.c
#include <stdio.h>
#include <string.h>
#include <pcre.h>

int main(int argc, char *argv[]) {
#define SIZE 80
   char buf[SIZE] = "test test test test ",
        pattern[SIZE] = "s";
   if (argc > 1) strncpy(pattern, argv[1], SIZE - 1);
   const unsigned char *locale_tables = pcre_maketables();
   const char *error;
   int erroffset;
   pcre *re = pcre_compile((char*)pattern, 0, &error, &erroffset, locale_tables);
   if (!re) {
      printf("pattern compile error: %s\n", error);
      return 1;
   }
#define MATCHSIZE 20
   int ovector[MATCHSIZE * 2];
   while (fgets(buf, sizeof(buf) - 1, stdin)) {
      if (buf[strlen(buf) - 1] == '\n') buf[strlen(buf) - 1] = '\0';
      if (0 == strlen(buf)) break; // выход
      int count = pcre_exec(re, NULL, (char*)buf, strlen(buf), 0, 0, ovector, MATCHSIZE);
      if (count < 0) {
         printf("no match found\n--------------------\n");
         continue;
      }
      for (int c = 0; (c < 2 * count) && (ovector[c] >= 0); c += 2) {
         printf("%d/%d : ", ovector[c], ovector[c + 1]);
         for (int i = ovector[c]; i < ovector[c + 1]; i++)
            printf("%c", buf[i]);
         printf("\n");
      }
      printf("--------------------\n");
   }
   pcre_free(re);
   return 0;
}
```

Здесь та же история, что и ранее: сначала нам предстоит скомпилировать шаблон регулярного выражения, а затем сколько угодно раз сопоставлять его с входными строками. Глубоко в недрах документации библиотеки утверждается, что если последний параметр вызова компиляции `pcre_compile()` равен `NULL`, то используется собственная таблицей символов `pcre_default_tables`, которая определена в исходном файле `chartables.c` и вкомпилирована в модуль библиотеки. Но эту таблицу можно и изменить вызовом `pcre_maketables()`, которая не предусматривает параметров и использует **текущую** установленную локаль операционной системы.

Для сборки обязательна отдельная библиотека lib`pcre.so` (и в ряде дистрибутивов её лучше указать в командной строке после файла исходного кода … на самом деле — после объектного файла):

```
$ gcc -Wall regex3.c -l pcre -o regex3
```

Проверяем то, что из этого получилось:

```
$ ./regex3 "Ё"
это Ёлка
7/9 : Ё
--------------------
а это ёлка
no match found
--------------------
31.12.2016 NewYear - Ёлка
21/23 : Ё
--------------------
```

(Литеры 'Ё' и 'ё', если помните, имеют в Unicode особое соотношение с остальным русским алфавитом, а поэтому заслуживают особой проверки … как и 'Й' и 'й'.)

```
$ ./regex3 "^([^,]*),([^,]*),([^,]*)$"
123,4567,898709
0/15 : 123,4567,898709
0/3 : 123
4/8 : 4567
9/15 : 898709
--------------------
a,b,c
0/5 : a,b,c
0/1 : a
2/3 : b
4/5 : c
--------------------
й,ё,Ё
0/8 : й,ё,Ё
0/2 : й
3/5 : ё
6/8 : Ё
--------------------
слово,ещё слово,снова слово
0/50 : слово,ещё слово,снова слово
0/10 : слово
11/28 : ещё слово
29/50 : снова слово
--------------------
слово , word, ещё слово
0/36 : слово , word, ещё слово
0/11 : слово
12/17 :  word
18/36 :  ещё слово
--------------------
```

В итоге нашего рассмотрения видим, что все представленные в C методы сопоставления с образцом могут работать с локализованными строками в **мультибайтном** представлении UTF-8. Связано это с тем, что осуществляется поиск **подстрок байт**, даже если каждый из этих байт и бессмысленный в терминологии Unicode многобайтного символа.

Но эта переносимость на локализованные символы весьма условна — следует проявлять большую осторожность (и даже настороженность) в тех случаях, когда шаблон оперирует терминами не подстрок, а **символов** (например числом повторений символов) потому что символы-байты в данном представлении неадекватны. Сравните:

```
$ ./regex3 "t{2,}"
etty
1/3 : tt
--------------------
$ ./regex3 "ш{2,}"
вашшче
no match
```

```
--------------------
```

Но:

```
$ ./regex3 "(аш){2,}"
ашаш
0/8 : ашаш
4/8 : аш
--------------------
$ ./regex3 "(аш)+"
шабаш
6/10 : аш
6/10 : аш
--------------------
```

Дальше мы переходим к аналогичному использованию более новой реинкарнации: PCRE2… Здесь нам также нужно вручную стандартным образом доустановить пакеты, например, такой, для полноты картины, набор:

```
$ aptitude search pcre2 | grep ^i
i  libpcre2-16-0 - New Perl Compatible Regular Expression Library - 16 bit runtime files
i  libpcre2-32-0 - New Perl Compatible Regular Expression Library - 32 bit runtime files
i  libpcre2-8-0 - New Perl Compatible Regular Expression Library- 8 bit runtime files
i A libpcre2-dev - New Perl Compatible Regular Expression Library - development files
i A libpcre2-posix2 - New Perl Compatible Regular Expression Library - posix-compatible runtime
files
i  pcre2-utils - New Perl Compatible Regular Expression Library — utilities
```

Приложение функционально эквивалентное тому, что мы собирали выше для PCRE:

```
$ cat regex6.c
#include <stdio.h>
#include <string.h>
#define PCRE2_CODE_UNIT_WIDTH 8
#include <pcre2.h>

int main(int argc, char *argv[]) {
#define SIZE 120
  unsigned char pattern[SIZE] = ".";
  if (argc > 1) strncpy((char*)pattern, argv[1], SIZE - 1);
  size_t pattern_size = strlen((char*)pattern);
  uint32_t options = 0;
  size_t erroffset;
  int errcode;
  pcre2_code *re;
  char buf[SIZE];
  re = pcre2_compile(pattern, pattern_size, options, &errcode, &erroffset, NULL);
  if (re == NULL) {
    pcre2_get_error_message(errcode, (unsigned char*)buf, SIZE);
    printf("%d\t%s\n", errcode, buf);
    return 1;
  }
#define MATCHSIZE 20
  uint32_t ovecsize = MATCHSIZE * 2;
  pcre2_match_data *match_data = pcre2_match_data_create(ovecsize, NULL);
  while (fgets(buf, sizeof(buf) - 1, stdin)) {
    if (buf[strlen(buf) - 1] == '\n') buf[strlen(buf) - 1] = '\0';
    if (0 == strlen(buf)) break; // выход
    int rc = pcre2_match(re, (unsigned char*)buf, strlen(buf),
                         0, options, match_data, NULL);
    if(rc == 0) {
      printf("offset vector too small: %d\n--------------------\n", rc);
      continue;
    }
```

```
    else if (rc < 0) {
      printf("no match found\n--------------------\n");
      continue;
    }
    else if(rc > 0) {
      size_t* ovector;
      ovector = pcre2_get_ovector_pointer(match_data);
      for(int i = 0; i < rc; i++) {
        printf("%ld/%ld : ", ovector[2 * i], ovector[2 * i + 1]);
        char* start = buf + ovector[2 * i];
        size_t slen = ovector[2 * i + 1] - ovector[2 * i];
        printf("%.*s\n", (int)slen, (char*)start);
      }
    }
    printf("--------------------\n");
  }
  pcre2_match_data_free(match_data);
  pcre2_code_free(re);
  return 0;
}
```

Видим, что используемый API существенно поменялся, но логика приложения сохраняется всё та же. Прежде включения заголовочного файла <pcre2.h> **требуется** определить разрядность представления единиц, с которыми будет работать отождествление. Разрядность может быть указана как 8, 16, 32. В данном примере мы указываем 8 и библиотека регулярных выражений будет работать с байтовыми единицами, т. е. То же представление UTF-8 как и в предыдущем случае. Значение 16, как я понимаю, предназначено для Unicode представления UTF-16 в операционной системе Windows, и это мы рассматривать не будем. К разрядности 32 мы вернёмся очень скоро.

Сборку приложения осуществляем командой (здесь нас интересует указание подключаемой библиотеки):

```
$ gcc -Wall regex6.c -l pcre2-8 -o regex6
$ ldd regex6
        linux-vdso.so.1 (0x00007ffd649eb000)
        libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007efe3f8b2000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007efe3f6c0000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007efe3f69d000)
        /lib64/ld-linux-x86-64.so.2 (0x00007efe3f970000)
```

Проверка выполнением (на интерпретации результатов не будем пока останавливаться, мы вернёмся к этому вскоре):

```
$ ./regex6 "^([^,]*),([^,]*),([^,]*)$"
и,й,ё
0/8 : и,й,ё
0/2 : и
3/5 : й
6/8 : ё
--------------------
```

PCRE и POSIX нотация

PCRE изначально нацелен на эквивалентность функций с нотацией (синтаксисом) регулярных выражений Perl. Но в библиотеках PCRE2 предусмотрен API для обеспечения обработки регулярных выражений в нотации POSIX (хотя утверждается что синтаксис POSIX беднее Perl). Выглядит это так:

```
$ cat regex5.c
#include <stdio.h>
#include <string.h>
#include <pcre2posix.h>
```

```
int main(int argc, char *argv[]) {
#define SIZE 120
   char buf[SIZE] = "test test test test ";
   char pattern[SIZE] = "s";
   if (argc > 1) strncpy(pattern, argv[1], SIZE - 1);
   regex_t preg;
   if (pcre2_regcomp(&preg, (char*)&pattern,
       REG_UCP | REG_UTF | REG_ICASE) != 0) {
      printf("pattern compile error: %m\n");
      return 1;
   }
   while (fgets(buf, sizeof(buf) - 1, stdin)) {
      if (buf[strlen(buf) - 1] == '\n') buf[strlen(buf) - 1] = '\0';
      if (0 == strlen(buf)) break; // выход
#define MATCHSIZE 20
      regmatch_t pmatch[MATCHSIZE];
      int rc = pcre2_regexec(&preg, (char*)buf, MATCHSIZE,
                             (regmatch_t*)&pmatch, REG_NOTBOL);
      if (rc == REG_NOMATCH ) {
         printf("no match found\n--------------------\n");
         continue;
      }
      else if (rc != 0) {
         pcre2_regerror(rc, &preg, buf, SIZE);
         printf("error: %s\n--------------------\n", buf);
         continue;
      }
      for (int c = 0; c < MATCHSIZE; c ++) {
         if (-1 == pmatch[c].rm_so) break;
         printf("%d/%d : ", pmatch[c].rm_so, pmatch[c].rm_eo);
         for (int i = pmatch[c].rm_so; i < pmatch[c].rm_eo; i++)
            printf("%c", buf[i]);
         printf("\n");
      }
      printf("--------------------\n");
   }
   pcre2_regfree(&preg);
   return 0;
}
```

Обращаем внимание на другой заголовочный файл `<pcre2posix.h>` определяющий изменения API. Кроме того, меняется и динамическая библиотека с которой мы компонуем приложение:

```
$ gcc -Wall regex5.c -l pcre2-posix -o regex5
$ ldd regex5
        linux-vdso.so.1 (0x00007ffdbc3ef000)
        libpcre2-posix.so.2 => /lib/x86_64-linux-gnu/libpcre2-posix.so.2 (0x00007f2299778000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2299586000)
        libpcre2-8.so.0 => /lib/x86_64-linux-gnu/libpcre2-8.so.0 (0x00007f22994f5000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f22997aa000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f22994d2000)
```

Вот как выглядит выполнение:

```
$ ./regex5 "([0-9]*)\.([0-9]*)\.([0-9]*)\.([0-9]*)"
192.168.1.3
0/11 : 192.168.1.3
0/3 : 192
4/7 : 168
8/9 : 1
10/11 : 3
```

```
--------------------
192.168.1
no match found
--------------------
```

Широкие символы Unicode

Библиотеки C не имеют возможности работы с **контекстом** мультибайтных локализованных строк представленных в кодировке UTF-8. Для корректной работы с контекстом Unicode текстов POSIX вводит тип широких символов `wchar_t`. Сам тип широких локализованных символов (`wchar_t`) появился в стандарте C89, но, в полной мере с API поддержки его развернулось только в стандарте C99. В таком представлении **каждый и любой** символ представляется 4-байтным значением.

Библиотеки PCRE2 предоставляют **корректный** механизм работы с широкими символами, свободный от коротко упоминавшихся ранее артефактов, связанных с мультибайтным представлением. Вот как может выглядеть пример эквивалентный по функциональности показанному ранее `regex6.c` :

```
$ cat regex7.c
#include <stdio.h>
#include <wchar.h>
#include <locale.h>
#include <string.h>
#define PCRE2_CODE_UNIT_WIDTH 32
#include <pcre2.h>

void c2w(const char *c, wchar_t *w) {
   int n = -1;
   while (n != 0)
      c += (n = mbtowc(w++, c, MB_CUR_MAX));
}

int main(int argc, char *argv[]) {
  setlocale(LC_ALL, ""); // только после этого работают преобразования c2w()!
#define SIZE 120
  wchar_t wpattern[SIZE] = L".";
  if (argc > 1) c2w(argv[1], wpattern);
  size_t wpattern_size = wcslen(wpattern);
  uint32_t options = 0;
  size_t erroffset;
  int errcode;
  wchar_t wbuf[SIZE];
  pcre2_code* re = pcre2_compile((unsigned int*)wpattern, wpattern_size,
                                 options, &errcode, &erroffset, NULL);
  if (re == NULL) {
    pcre2_get_error_message(errcode, (unsigned int*)wbuf, SIZE);
    printf("%d\t%ls\n", errcode, wbuf);
    return 1;
  }
#define MATCHSIZE 20
  uint32_t ovecsize = MATCHSIZE * 2;
  pcre2_match_data *match_data = pcre2_match_data_create(ovecsize, NULL);
  char buf[SIZE];
  while (fgets(buf, sizeof(buf) - 1, stdin)) {
    if (buf[strlen(buf) - 1] == '\n') buf[strlen(buf) - 1] = '\0';
    if (0 == strlen(buf)) break; // выход
    c2w(buf, wbuf);
    int rc = pcre2_match(re, (unsigned int*)wbuf, wcslen(wbuf),
                         0, options, match_data, NULL);
    if(rc == 0) {
      printf("offset vector too small: %d\n--------------------\n", rc);
      continue;
```

```
    }
    else if (rc < 0) {
      printf("no match found\n--------------------\n");
      continue;
    }
    else if(rc > 0) {
      size_t* ovector;
      ovector = pcre2_get_ovector_pointer(match_data);
      for(int i = 0; i < rc; i++) {
        printf("%ld/%ld : ", ovector[2 * i], ovector[2 * i + 1]);
        wchar_t* start = wbuf + ovector[2 * i];
        size_t slen = ovector[2 * i + 1] - ovector[2 * i];
        for (int j = 0; j < slen; j++)
            printf("%lc", *(start + j));
        printf("\n");
      }
    }
    printf("--------------------\n");
  }
  pcre2_match_data_free(match_data);
  pcre2_code_free(re);
  return 0;
}
```

Собираем приложение так:

```
$ gcc -Wall regex7.c -l pcre2-32 -o regex7
$ ldd regex7
        linux-vdso.so.1 (0x00007ffd517b4000)
        libpcre2-32.so.0 => /lib/x86_64-linux-gnu/libpcre2-32.so.0 (0x00007f172bfbe000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f172bdcc000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f172bda9000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f172c067000)
```

А теперь **сравним** работу приложений `regex6` (оперирующее с мультибайтовыми представлениями UTF-8) и `regex7` (оперирующее с `wchar_t` представлением каждого Unicode символа):

```
$ ./regex6 "^([^,]*),([^,]*),([^,]*)$"
й,g,ё
0/7 : й,g,ё
0/2 : й
3/4 : g
5/7 : ё
--------------------
$ ./regex7 "^([^,]*),([^,]*),([^,]*)$"
й,g,ё
0/5 : й,g,ё
0/1 : й
2/3 : g
4/5 : ё
--------------------
```

В чём разница? А в том, что 1-е приложение (`regex6`) оперирует с **байтами**, ничего не зная о том как эти байты складываются в символы, а 2-е приложение (`regex7`) оперирует с **символами** Unicode, независимо от того, скольким байтами представляется очередной символ: 1 байт для ASCII, 2 байта для кирилицы, или 4 байта для какого-то экзотического иероглифического письма (обратите внимание на последовательность нумерации в выводе). Понятно, что 2-й вариант — абсолютно корректен, а 1-й — достаточно условный («на грани фола»).

Акцентируем внимание, что **все** показанные предыдущие варианты работы с регулярными выражениями, исключая этот последний, **не позволяют** организовать абсолютно корректное

обслуживание многоязычных Unicode текстов.

Регулярные выражения в C++

C++ — это язык для разработки и использования элегантных и эффективных абстракций.

Бьёрн Страуструп

Естественно, **все** перечисленные выше инструменты применимы и в коде на языке C++ (поскольку язык C++ является **надмножеством** языка C, а не каким-то совершенно новым языком). Хотя библиотека PCRE (см. выше) гораздо чаще обсуждается и применяется относительно C++, чем к самому C (так было до стандарта C++11).

Тем не менее, C++ предлагает свои инструменты работы с регулярными выражениями — определения в файле `<regex>`, работающие по честному с строками локализованных (широких) символов `wstring`. Но использовать эти возможности можно только с компилятором (или при указании такого его режима), поддерживающим, как минимум, стандарт C++11. В противном случае, вы сразу же получите сообщение об ошибке:

```
$ g++ regex1++.cc -oregex1++
In file included from /usr/include/c++/5.3.1/regex:35:0,
                 from regex1++.cc:3:
/usr/include/c++/5.3.1/bits/c++0x_warning.h:32:2: ошибка: #error This file requires compiler
and library support for the ISO C++ 2011 standard. This support must be enabled with the -
std=c++11 or -std=gnu++11 compiler options.
```

Приложение, сделанное во многом эквивалентным показанным в предыдущих разделах, но с использованием этого нового механизма. Мы планируем производить **контекстный** поиск по соответствию образцу — а это означает что мы должны использовать широкие Unicode символы, строки из `wchar_t`, или тип `wstring`:

```
$ cat regex1++.cc
#include <iostream>
#include <locale>
#include <regex>
using namespace std;

int main(int argc, char *argv[]) {
   locale::global(locale ("ru_RU.utf8"));
   wstring wp = L"строка";
   if (argc > 1) {
      wchar_t warg[strlen(argv[1]) + 1];
      mbstowcs(warg, argv[1], strlen(argv[1]) + 1);
      wp = wstring(warg);
   }
   wregex pattern(wp, regex_constants::icase | regex_constants::awk);
   while (true) {
      wstring buf;
      getline(wcin, buf);
      if (wcin.eof() || buf.empty()) break;
      wcmatch match;                // результат сопоставления
      if (!regex_search((wchar_t*)buf.c_str(), match, pattern)) {
         wcout << "no match found" << endl
               << "--------------------" << endl;
         continue;
      }
      for (unsigned i=0; i<match.size(); ++i) {
         wcout << match.position(i) << L"/" << match[i].length()
               << match.position(i) + match[i].length()
               << L" : " << match[i] << endl;
      }
      wcout << "--------------------" << endl;
   }
```

```
}
```

Насколько теперь всё стало проще и короче! И это понятно — поддержка `<regex>` сделана через STL, в частности, результат сопоставлений `wcmatch` — это `vector<wstring>`. В этом и оборотная сторона этого инструмента: при ошибках в определениях типов в вашем коде создаются целые огромные простыни сообщений об ошибках, которые почти невозможно истолковывать.

Сборка (про стандарт C++11 мы уже сделали замечание):

```
$ g++ -Wall -std=c++11 regex1++.cc -o regex1++
```

Работает это уже знакомым нам способом:

```
$ ./regex1++ "([0-9]*)\.([0-9]*)\.([0-9]*)\.([0-9]*)"
192.168.1.3
0/11 : 192.168.1.3
0/3 : 192
4/7 : 168
8/9 : 1
10/11 : 3
--------------------
адрес IP: 192.168.1.3
10/21 : 192.168.1.3
10/13 : 192
14/17 : 168
18/19 : 1
20/21 : 3
--------------------
```

Как и следовало ожидать (как планировалось), такой код совершенно корректно работает с локализованными текстами:

```
$ ./regex1++ "^([^,]*),([^,]*),([^,]*)"
слово1,слово2,слово3,слово4
0/20 : слово1,слово2,слово3
0/6 : слово1
7/13 : слово2
14/20 : слово3
--------------------
```

Обратите внимание на начальные позиции (групп) и длины: все эти величины выражены в **символах**, а не **байтах**!

Интересно заглянуть в заголовки `<regex>`. В частности, на вариации **стиля** сопоставления с образцом:

```
        static constexpr flag_type icase = regex_constants::icase;
        static constexpr flag_type nosubs = regex_constants::nosubs;
        static constexpr flag_type optimize = regex_constants::optimize;
        static constexpr flag_type collate = regex_constants::collate;
        static constexpr flag_type ECMAScript = regex_constants::ECMAScript;
        static constexpr flag_type basic = regex_constants::basic;
        static constexpr flag_type extended = regex_constants::extended;
        static constexpr flag_type awk = regex_constants::awk;
        static constexpr flag_type grep = regex_constants::grep;
        static constexpr flag_type egrep = regex_constants::egrep;
```

Это показывает сколько **различающихся** диалектов синтаксиса регулярных выражений размножилось в природе, когда один и тот же образец будет восприниматься по-разному, давая в итоге различный результат. В коде своего сопоставления вы можете определить это флагами при создании шаблона, например так:

```
    wregex pattern(wp, regex_constants::icase | regex_constants::awk);
```

Здесь, заодно, показана установка флага **нечувствительности** к регистру текста (так как по умолчанию шаблон различает регистр):

```
$ ./regex1++ слово
В начале было Слово, и Слово было у Бога, и Слово было Бог
14/19 : Слово
--------------------
```

Если же вас интересует не поиск соответствия, а **полное соответствие** выражения образцу, то вместо `regex_search()` используется `regex_match()`. В простейшем виде это выглядит как-то так:

```
$ cat fio.cc
#include <iostream>
#include <locale>
#include <regex>
using namespace std;

int main( int argc, char *argv[] ) {
   locale::global( locale ( "ru_RU.utf8" ) );
   wcmatch m = wcmatch {};
   while( true ) {
      wstring buf;
      getline( wcin, buf );
      if( wcin.eof() ) break;
      if( !regex_match( (wchar_t*)buf.c_str(), m, wregex { LR"((\w+) (\w+) (\w+))" } ) )
         wcout << L"ошибочный формат" << endl;
      else
         wcout << L"фамилия='" << m[ 1 ].str()
               << L"' имя='" << m[ 2 ].str()
               << L"' отчество='" << m[ 3 ].str() << L'\'' << endl;
   }
}

$ g++ -Wall -std=c++11 fio.cc -o fio
$ ./fio
Иванов Иван Иванович
фамилия='Иванов' имя='Иван' отчество='Иванович'
$ ./fio < fio.dat
фамилия='Иванов' имя='Иван' отчество='Иванович'
фамилия='Петров' имя='Пётр' отчество='Петрович'
фамилия='Сидоров' имя='Глеб' отчество='Кузьмич'
фамилия='Чапаев' имя='Василий' отчество='Иванович'
```

Если же вам нужен **циклический поиск всех** (или некоторых) последовательных сопоставлений с образцом, то библиотека вводит новые типы (классы) — несколько **итераторов** поиска сопоставлений (с различными типами сопоставляемой строки):

Тип	Определение
`cregex_iterator`	`regex_iterator<const char*>`
`wcregex_iterator`	`regex_iterator<const wchar_t*>`
`sregex_iterator`	`regex_iterator<std::string::const_iterator>`
`wsregex_iterator`	`regex_iterator<std::wstring::const_iterator>`

Инкремент такого итератора ищет следующее сопоставление в строке (соответствующего представления). Конечным итератором считается итератор, производимый конструктором без параметров. Пример предыдущего примера контекстного поиска (`regex1++.cc`), переписанного в терминах итераторов, может выглядеть так:

```
$ cat regex2++.cc
#include <iostream>
#include <locale>
#include <regex>
using namespace std;
```

```
int main(int argc, char *argv[]) {
   locale::global(locale ("ru_RU.utf8"));
   wstring wp = L"строка";
   if(argc > 1) {
      wchar_t warg[strlen(argv[1]) + 1];
      mbstowcs(warg, argv[1], strlen(argv[1]) + 1);
      wp = wstring(warg);
   }
   wregex pattern(wp, regex_constants::icase | regex_constants::awk);
   while (true) {
      wstring buf;
      getline(wcin, buf);
      if (wcin.eof() || buf.empty()) break;
      wsregex_iterator beg = wsregex_iterator(buf.begin(), buf.end(), pattern),
                       end = wsregex_iterator();
      wcout << L"Число сопоставлений " << distance(beg, end) << endl;
      for (auto i = beg; i != end; i++)                  // сопоставление
         for(long unsigned int j = 0; j < i->size(); j++) // группа
            wcout << i->position(j) << L"/" << i->length(j)
                  << L" : " << i->str(j) << endl;
      wcout << "--------------------" << endl;
   }
}
```

Сборка:

```
$ g++ -Wall -std=c++11 regex2++.cc -o regex2++
```

Как уже сказано, этот код, в отличие от всех предыдущих, ищет не единичное, а последовательно **все** возможные (не пересекающиеся) сопоставления. Некоторое количество примеров, потому что они не так очевидны:

```
$ ./regex2++ Слово
В начале было Слово, и Слово было у Бога, и Слово было Бог
Число сопоставлений 3
14/19 : Слово
23/28 : Слово
44/49 : Слово
--------------------
$ ./regex2++ Бог
В начале было Слово, и Слово было у Бога, и Слово было Бог
Число сопоставлений 2
36/39 : Бог
55/58 : Бог
--------------------
$ ./regex2++ Бог$
В начале было Слово, и Слово было у Бога, и Слово было Бог
Число сопоставлений 1
55/58 : Бог
--------------------
$ ./regex2++ "([0-9]*)\.([0-9]*)\.([0-9]*)\.([0-9]*)"
адрес IP: 192.168.1.3 в сети адрес IP: 192.168.1.0
Число сопоставлений 2
10/21 : 192.168.1.3
10/13 : 192
14/17 : 168
18/19 : 1
20/21 : 3
39/50 : 192.168.1.0
39/42 : 192
43/46 : 168
47/48 : 1
49/50 : 0
```

```
--------------------
192.168.1.3
Число сопоставлений 1
0/11 : 192.168.1.3
0/3 : 192
4/7 : 168
8/9 : 1
10/11 : 3
--------------------
192.168.1
Число сопоставлений 0
--------------------
```

(Детали использования итераторов сопоставлений см. в справочнике [11] по библиотеке регулярных выражений C++.)

Этот новый инструмент работы с регулярными выражениями предоставляет не только возможность простого сопоставления с образцом, но и ряд других возможностей, например, контекстную замену подстроки. Проверяем как это работает со строками широких локализованных символов:

```
$ cat regexr.cc
#include <iostream>
#include <locale>
#include <regex>
using namespace std;

wstring warg(char* arg) {
   wchar_t wa[strlen(arg) + 1];
   mbstowcs(wa, arg, strlen(arg) + 1);
   return wstring(wa);
}

int main(int argc, char *argv[]) {
   locale::global(locale ("ru_RU.utf8"));
   wstring wpat = 1 == argc ? L"слово" : warg(argv[1]),
           replacement = argc < 3 ? L"понятие" : warg(argv[2]);
   wcout << wpat << L" -> " << replacement << endl;
   basic_regex<wchar_t> pattern(wpat); // образец сопоставления
   while(true) {
      wstring buf;
      getline(wcin, buf);
      if(wcin.eof() || buf.empty()) break;
      wcout << regex_replace(buf, pattern, replacement) << endl;
   }
}
```

И результаты того как это работает:

```
$ g++ -Wall -std=c++11 regexr.cc -o regexr
$ ./regexr Слово понятие
Слово -> понятие
В начале было Слово, и Слово было у Бога, и Слово было Бог
В начале было понятие, и понятие было у Бога, и понятие было Бог
$ ./regexr "c{1,}" z
c{1,} -> z
классик
клazик
российский
рozийzкий
сказ класс российский
zказ клаz рozийzкий
```

Поздние языки программирования

Более современные языки программирования, конечно, никак не могли обойти стороной регулярные выражения, и (разным образом) предоставляют **встроенные** механизмы работы с ними (в отличие от C/C++ для которых мы в каждом случае дополнительно устанавливали сторонний инструментарий). Поэтому нам остаётся только кратко проиллюстрировать как это выглядит… (Мы не будем обсуждать ни составление конкретных шаблонов сопоставления, ни полного перечня предоставляемых API — только то как подключить механизм регулярных выражений и начать его использовать.)

Python

Ещё в начале этого года (2011) я бы рассматривал этот вопрос начиная с Python 2 ... Но в этом (2022) году произошло очень важное событие: в **основных** дистрибутивах Linux, как итог 14-летних усилий (2008 — 2022 … не так просто это оказалось!) **дефаултной** версией стал Python 3, а Python 2 даже **по умолчанию** не устанавливается в инсталляции системы, хотя и присутствует в стандартном репозитории для ручной установки, на любителя или в целях совместимости с ранним программным обеспечением. Это фактически означает конец 22 летней (2000 — 2022) славной истории Python 2. Поэтому мы станем рассматривать только Python 3...

Обработка текстовой информации вообще, и регулярные выражения как самый мощный механизм такой обработки, всё это очень органично вписывается в области использования Python. Поэтому ничего удивительного в том, что поддержка регулярных выражений входят в Python давно и естественным образом. Более того, Python 3 декларирует использование по умолчанию кодировки UTF-8, не только для представления содержимого текстовых констант, но, даже, например, в написании **имён переменных** программы.

В Python работа с регулярными выражениями реализована в стандартном модуле `re`, который входит в стандартный дистрибутив Python (т. е. ничего не нужно специально предпринимать для его использования):

```
$ python3
Python 3.8.10 (default, Nov 14 2022, 12:59:47)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import re
>>> re.__version__
'2.2.1'
>>> quit()
```

Основные функции предлагаемые модулем `re`:

Функция	Её смысл
`re.search(pattern, string)`	Найти в строке `string` первую строчку, подходящую под шаблон `pattern`;
`re.fullmatch(pattern, string)`	Проверить, подходит ли строка `string` под шаблон `pattern`;
`re.split(pattern, string, maxsplit=0)`	Аналог `str.split()`, только разделение происходит по подстрокам, подходящим под шаблон `pattern`;
`re.findall(pattern, string)`	Найти в строке `string` все непересекающиеся шаблоны `pattern`;
`re.finditer(pattern, string)`	Итератор всем непересекающимся шаблонам `pattern` в строке `string` (выдаются match-объекты);
`re.sub(pattern, repl, string, count=0)`	Заменить в строке `string` все непересекающиеся шаблоны `pattern` на `repl`;

Несколько примеров того как используются операции модуля `re`:

```
$ cat regexp.py
#!/usr/bin/python3
import re

match = re.search(r'\d\d\D\d\d', r'Телефон 123-12-12')
print(match[0] if match else 'Not found')

match = re.search(r'\d\d\D\d\d', r'Телефон 1231212')
print(match[0] if match else 'Not found')

match = re.fullmatch(r'\d\d\D\d\d', r'12-12')
print('YES' if match else 'NO')
match = re.fullmatch(r'\d\d\D\d\d', r'T. 12-12')
print('YES' if match else 'NO')

print(re.split(r'\W+', 'Где, скажите мне, мои очки??!'))

print(re.findall(r'\d\d\.\d\d\.\d{4}',
                 r'Эта строка написана 19.01.2018, а могла бы и 01.09.2017'))

for m in re.finditer(r'\d\d\.\d\d\.\d{4}',
                     r'Эта строка написана 19.01.2018, а могла бы и 01.09.2017'):
    print('Дата', m[0], 'начинается с позиции', m.start())

print(re.sub(r'\d\d\.\d\d\.\d{4}',
             r'DD.MM.YYYY',
             r'Эта строка написана 19.01.2018, а могла бы и 01.09.2017'))
```

И то как это срабатывает:

```
$ ./regexp.py
23-12
Not found
YES
NO
['Где', 'скажите', 'мне', 'мои', 'очки', '']
['19.01.2018', '01.09.2017']
Дата 19.01.2018 начинается с позиции 20
Дата 01.09.2017 начинается с позиции 45
Эта строка написана DD.MM.YYYY, а могла бы и DD.MM.YYYY
```

Если функции `re.search`, `re.fullmatch` не находят соответствие шаблону в строке, то они возвращают значение `None` (а функция `re.finditer` не выдаёт ничего). А если соответствие найдено (что нас зачастую и интересует), то возвращается `match`-объект. Это достаточно сложная структура, содержащая в себе кучу полезной информации о соответствии шаблону (полный набор атрибутов нужно смотреть в документации):

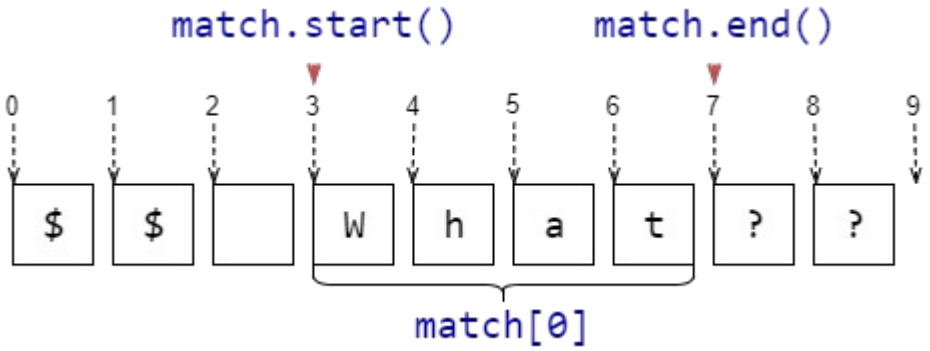
```
>>> match = re.search(r'\d\d\D\d\d', r'Телефон 123-12-12')
>>> type(match)
<class 're.Match'>
>>> print(match)
<re.Match object; span=(9, 14), match='23-12'>
>>>
```

Если шаблон регулярного выражения предстоит использовать неоднократно, его полезно предварительно **компилировать** (как уже было сказано ранее):

```
$ python3
Python 3.8.10 (default, Nov 14 2022, 12:59:47)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import re
```

```
>>> regex = re.compile('\s+')
>>> regex.split('А роза упала на лапу Азора')
['А', 'роза', 'упала', 'на', 'лапу', 'Азора']
>>> type(regex)
<class 're.Pattern'>
```

Метод match	Описание	Пример
match[0], match.group()	Подстрока, соответствующая шаблону	match = re.search(r'\w+', r'$$ What??') match[0] # -> 'What'
match.start()	Индекс в исходной строке, начиная с которого идёт найденная подстрока	match = re.search(r'\w+', r'$$ What??') match.start() # -> 3
match.end()	Индекс в исходной строке, который следует сразу за найденной подстрока	match = re.search(r'\w+', r'$$ What??') match.end() # -> 7



Как уже показано выше, отрабатывать (тестировать) работу с регулярными выражениями в Python можно и в режиме интерпретатора:

```
$ python3
Python 3.8.10 (default, Nov 14 2022, 12:59:47)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> text = """
... 100 ИНФ  Информатика
... 213 МАТ  Математика
... 156 АНГ  Английский
... """
>>> import re
>>> regex_num = re.compile('\d+')
>>> regex_num.findall(text)
['100', '213', '156']
>>>
```

И теперь, рассмотревши разнообразие API в Python из области регулярных выражений, и убедившись насколько предрасположен Python к подобного образа деятельности, сделаем диалоговое приложение, подобное таким же на других языках программирования:

```
$ cat regexp2.py
#!/usr/bin/python3
import sys
import re

if len( sys.argv ) > 1:
    pattern = sys.argv[ 1 ]
else:
    pattern = r'\d\d\d'
try:
    rex = re.compile( pattern, flags = re.IGNORECASE )
except Exception as exc:
```

```
    print( "error: {}".format( exc ) )
    sys.exit( 1 )

while(True):
    buf = sys.stdin.readline()[ :-1 ]
    if 0 == len( buf ):             # завершение работы
        break
    match = rex.search( buf )
    if match:
        for i in range( 0, len( match.groups() ) + 1 ):
            print("{}/{} : {}".format(
                    match.span( i )[ 0 ], match.span( i )[ 1 ],
                    match.group( i ) ) )
    else:
        print( "no match found" )
    print( "--------------------" )

re.purge()
sys.exit( 0 )
```

Поскольку Python язык интерпретирующий (в значительной мере), то компиляция здесь не нужна, и сразу переходим к выполнению:

```
$ ./regexp2.py "([0-9]*)\.([0-9]*)\.([0-9]*)\.([0-9]*)"
это IP: 192.168.1.3
8/19 : 192.168.1.3
8/11 : 192
12/15 : 168
16/17 : 1
18/19 : 3
--------------------
192.168.1
no match found
--------------------
192.168.1.3
0/11 : 192.168.1.3
0/3 : 192
4/7 : 168
8/9 : 1
10/11 : 3
--------------------
```

Ещё один короткий пример, для того чтобы утвердиться в том, что Python (его API регулярных выражений) считает позиции отождествлений (для строк мультибайтных локализованных символов) не в **байтах**, но в **символах** (шаблон программы по умолчанию: `r'\d\d\d'` — 3 цифровых символа):

```
$ ./regexp2.py
1234
0/3 : 123
--------------------
число 987
6/9 : 987
--------------------
число this 987
11/14 : 987
--------------------
```

Go

Язык Go (компилирующий) в значительной мере является продолжателем линии языка C (и у основания этих языков стоят одни и те же лица, только с разрывом в 40 лет). Но, в противовес C/C+

+, язык Go изначально вводит для обработки символьной информации встроенный (builtin) тип строки `string` и обширный API строчной обработки. Поэтому язык никак не мог оставить в стороне и работу с регулярными выражениями — GoLang имеет в своей **стандартной** библиотеке пакет обработки регулярных выражений, пакет `regexp`.

```
$ cat regexg.go
package main
import ("fmt"; "regexp")

func main() {
        matched, _ := regexp.MatchString("cat", "black cat meow")
        fmt.Println(matched)

        re, _ := regexp.Compile("cat")
        res := re.FindAllString("black cat meowcat", -1)
        fmt.Println(res)
}

$ go run regexp.go
true
[cat cat]
```

Показанное выше выполнение примера похоже на интерпретацию (как это было в случае Python), но на самом деле это полноценная компиляция, выполняющаяся онлайн, с последующим выполнением (команда: `go run ...`). Приведенный пример кода настолько понятен, что не требует дополнительных объяснений.

Наконец, сделаем, для сравнения, приложение, которое ведёт себя подобно тем, которые мы писали для C/C++. Более того, закончив отлаживать код доверим его форматирование («красоту», «кодестайл») самой системе Go:

```
$ go fmt regexg2.go
regexg2.go
```

После чего получим следующее (так выглядит «кодестайл» как его понимает GoLang при автоматическом форматировании):

```
$ cat regexg2.go
package main

import (
       "bufio"
       "fmt"
       "os"
       "regexp"
)

func main() {
       pattern := "."
       if len(os.Args) > 1 {
              pattern = os.Args[1]
       }
       re, err := regexp.Compile(pattern)
       if err != nil {
              fmt.Println(err)
              return
       }
       sc := bufio.NewScanner(os.Stdin)
       for sc.Scan() {
              buf := sc.Text()
              if 0 == len(buf) { // выход
                     break
              }
```

```
                res := re.FindAllSubmatch([]byte(buf), -1)
                if nil == res {
                        fmt.Println("no match found\n")
                        fmt.Printf("--------------------\n")
                        continue
                }
                ind := re.FindAllSubmatchIndex([]byte(buf), -1)
                for i := 0; i < len(res); i++ {
                        for j := 0; j < len(res[i]); j++ {
                                fmt.Printf("%d/%d : %s\n",
                                        ind[i][2*j], ind[i][2*j+1],
                                        res[i][j])
                        }
                }
                fmt.Printf("--------------------\n")
        }
}
```

Компилируем:

```
$ go build -o regexg2 regexg2.go
$ ls -l regexg2
-rwxrwxr-x 1 olej olej 2121284 дек 26 13:30 regexg2
$ file regexg2
regexg2: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, Go
BuildID=A5CwsdQ21MTPb8lByu9w/8i2a9bKVB2l4xHZoM9ig/lVjjVSMLAGftLi63mCJE/QxqWPp6scnWtlfZK0_VU,
with debug_info, not stripped
$ ldd regexg2
        не является динамическим исполняемым файлом
```

Сборка статическая... Современная система GoLang может осуществлять и динамическую сборку, со связыванием с динамическими библиотеками периода выполнения. Но мы не будем этим заниматься...

И несколько примеров на выполнение:

```
$ ./regexg2 "([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})\.([0-9]{1,3})"
192.168.1.3 принадлежит сети 192.168.1.0
0/11 : 192.168.1.3
0/3 : 192
4/7 : 168
8/9 : 1
10/11 : 3
44/55 : 192.168.1.0
44/47 : 192
48/51 : 168
52/53 : 1
54/55 : 0
--------------------
сеть 192.168.1.0
9/20 : 192.168.1.0
9/12 : 192
13/16 : 168
17/18 : 1
19/20 : 0
--------------------
$ ./regexg2 Слово
В начале было Слово, и Слово было у Бога, и Слово было Бог
25/35 : Слово
40/50 : Слово
76/86 : Слово
--------------------
```

В последнем результате Go, работающий со строками UTF-8, даёт смещения позиций совпадения в **байтах**, в отличие, например, от варианта C++, работающего с широкими символами (wchar_t), и дающего смещения в **символах**:

```
$ ./regex2++ Слово
В начале было Слово, и Слово было у Бога, и Слово было Бог
Число сопоставлений 3
14/19 : Слово
23/28 : Слово
44/49 : Слово
--------------------
```

Результаты очень **подобные** тому, что мы получали на C/C++, но они несколько **отличаются**, потому что, как это уже отмечалось, существует достаточно много **диалектов** регулярных выражений, отличающиеся синтаксисом задания шаблонов сопоставления, и различные инструментальные, языковые средства используют разные диалекты. В итоге, к сожалению, в реальных проектах часто приходится модифицировать вид шаблона под каждый конкретный проект — это ещё одна неприятная особенность использования регулярных выражений.

Rust

Язык Rust: а). ориентирован во многом как язык **системного** программирования, б). обеспечивающего повышенную **надёжность** за счёт семантики владения объектами, и в). претендующего на повышенную **производительность** скомпилированного кода (не ниже, а иногда и лучше, чем традиционный C). Эти принципы не особенно способствуют наличию развитых средств текстовой обработки. Но и в нём, примерно с 2014 года, было введено библиотечное расширение (crate в терминологии Rust) `regex` работы с регулярными выражениями. Это лишний раз указывает на то какую оценку этой технике придают разработчики языков.

Таким образом, для использования регулярных выражений нам необходимо загрузить **сторонний** крейт Rust для выполнения работы. Но с Rust это не так просто... Я **не знаю** (пока?) способа загрузить и использовать нужный крейт при использовании консольного компилятора `rusrc`, и о такой возможности, как это сделать, задаётся множество вопросов в Интернет. Но, к счастью, инсталляции Rust развиваются с самого начала как интегральная инфраструктура, включающая в свой состав **менеджер** проектов cargo. В его функции, помимо прочего разного, входит построение и обслуживание проектов, в том числе и загрузка любых требуемых внешних крейтов из сетевого репозитория.

```
$ cargo -V
cargo 1.66.0 (d65d197ad 2022-11-15)
$ cargo --help
Rust's package manager
...
Some common cargo commands are (see all commands with –list):
...
    build, b    Compile the current package
...
    clean       Remove the target directory
...
    new         Create a new cargo package
...
    add         Add dependencies to a manifest file
    run, r      Run a binary or example of the local package
...
```

Создадим **новый** (пустой) проект в котором будем работать:

```
$ cargo new regexrs2
     Created binary (application) `regexrs2` package

$ tree regexrs2
regexrs2
├── Cargo.toml
```

```
└── src
    └── main.rs

1 directory, 2 files
```

Здесь `Cargo.toml` — параметры проекта, в том числе и его внешние зависимости, а `main.rs` — главный файл проекта, с которого будет производится старт нашего приложения (изначально при создании это эталонный шаблон «Hello World»). Перечень доступных в репозитории крейтов (не все они библиотеки, здесь же и отдельные приложения) можно отобрать по интересующему контексту:

```
$ cd regexrs2
$ cargo search regex
regex = "1.7.0"                    # An implementation of regular expressions for Rust. This implementation uses finite automata …
lazy-regex = "2.3.1"               # lazy static regular expressions checked at compile time
proc-macro-regex = "1.1.0"         # A proc macro regex library
regex-automata = "0.2.0"           # Automata construction and matching using regular expressions.
easy-regex = "0.11.7"              # Make long regular expressions like pseudocodes
readable-regex = "0.1.0-alpha1"    # Regex made for humans. Wrapper to build regexes in a verbose style.
human_regex = "0.2.3"              # A regex library for humans
regex_static = "0.1.1"             # Compile-time validated regex, with convenience functions for lazy and static regexes.
webforms = "0.2.2"                 # Provides form validation for web forms
hashtag-regex = "0.1.1"            # A simple regex matching hashtags accoding to the unicode spec: http://unicode.org/reports/tr…
... and 891 crates more (use --limit N to see more)
```

Опускаясь в каталог созданного проекта, начинаем в нём выполнять операции, первой из которых будет — добавить зависимости (крейты) в проект (1-й раз это не быстрая операция, потому что скачиваются индексы всего репозитория):

```
$ cd regexrs2
$ cargo add regex
    Updating crates.io index
      Adding regex v1.7.0 to dependencies.
             Features:
             + aho-corasick
             + memchr
             + perf
             + perf-cache
             + perf-dfa
             + perf-inline
             + perf-literal
             + std
             + unicode
             + unicode-age
             + unicode-bool
             + unicode-case
             + unicode-gencat
             + unicode-perl
             + unicode-script
             + unicode-segment
             - pattern
             - unstable
             - use_std
```

После этого в файл параметров проекта `Cargo.toml` прописаны зависимости (причём, при такой форме команды, прописывается крейт последней доступной версии, хотя в команде мы можем указать и конкретную требуемую версию):

```
$ cat Cargo.toml | grep -A1 dependencies
[dependencies]
```

```
regex = "1.7.0"
```

Точно того же действия можно добиться (и так рекомендуют в литературе) ручным прописыванием строк зависимостей с их версиями (хотя автору кажется предпочтительным именно использование менеджера `cargo`).

А в файл исходного кода заменим содержимое но своё:

```
$ cat regexrs2/src/main.rs
use regex::Regex;

const TO_SEARCH: &'static str = "
On 2010-03-14, foo happened. On 2014-10-14, bar happened.
";

fn main() {
    let re = Regex::new(r"(\d{4})-(\d{2})-(\d{2})").unwrap();
    for caps in re.captures_iter(TO_SEARCH) {
        println!("year: {}, month: {}, day: {}",
                 caps.get(1).unwrap().as_str(),
                 caps.get(2).unwrap().as_str(),
                 caps.get(3).unwrap().as_str());
    }
}
```

Находясь всё в том же корневом каталоге проекта делаем построение (компиляцию) созданного проекта:

```
$ cargo build
   Compiling memchr v2.5.0
   Compiling regex-syntax v0.6.28
   Compiling aho-corasick v0.7.20
   Compiling regex v1.7.0
   Compiling regexrs2 v0.1.0 (/home/olej/2022/own.BOOKs/Localiz/regex.cod/regexrs2)
    Finished dev [unoptimized + debuginfo] target(s) in 6.59s
```

Всё! Мы создали первое готовое приложение, работающее с регулярными выражениями:

```
$ ./target/debug/regexrs2
year: 2010, month: 03, day: 14
year: 2014, month: 10, day: 14
```

Дальше мы создадим на Rust приложение подобное предыдущим, которое позволяет экспериментировать изменяя в диалоге как шаблон сопоставления (как параметр командной строки), так и целевую сопоставляемую строку:

```
$ cat regexrs1/src/main.rs
use std::env;
use std::io;
use regex::Regex;

fn main() {
   let arguments: Vec<String> = env::args().collect();
   let pattern = if arguments.len() > 1 {
      arguments[1].clone()
   } else {
      String::from(r"(\d{4})-(\d{2})-(\d{2})")
   };
   loop {
      let mut input = String::new();
      io::stdin().read_line(&mut input).expect("ошибка ввода");
      if 1 == input.len() { break }; // выход
      let buf = &input[0 .. input.len() - 1];
      let re = Regex::new(&pattern).expect("ошибка шаблона");
      if 0 == re.captures_iter(buf).count() {
```

```
            println!("no match found");
            println!("--------------------");
            continue;
         }
         for caps in re.captures_iter(buf) {
            for i in 0..caps.len() {
               let cap = caps.get(i).unwrap();
               println!("{}/{} : {}",
                        cap.start(), cap.end(), cap.as_str());
            }
            println!("--------------------");
         }
      }
   }
```

Построение проекта:

```
$ cargo clean
$ cargo build
   Compiling memchr v2.5.0
   Compiling regex-syntax v0.6.28
   Compiling aho-corasick v0.7.20
   Compiling regex v1.7.0
   Compiling regexrs1 v0.1.0 (/home/olej/2022/own.BOOKs/Localiz/regex.cod/regexrs1)
    Finished dev [unoptimized + debuginfo] target(s) in 7.45s
```

Выполнение (с параметрами подобными более ранним примерам для сравнения):

```
$ ./target/debug/regexrs1 "([0-9]*)\.([0-9]*)\.([0-9]*)\.([0-9]*)"
адрес IP: 192.168.1.3 в сети адрес IP: 192.168.1.0
15/26 : 192.168.1.3
15/18 : 192
19/22 : 168
23/24 : 1
25/26 : 3
54/65 : 192.168.1.0
54/57 : 192
58/61 : 168
62/63 : 1
64/65 : 0
--------------------
192.168.1.3
0/11 : 192.168.1.3
0/3 : 192
4/7 : 168
8/9 : 1
10/11 : 3
--------------------
192.168.1
no match found
--------------------
```

Статическая компиляция

Разработчики Rust, как уже было замечено ранее, уделяют исключительное внимание производительности кода Rust (и преуспевают в этом: временами скорость приложений Rusr больше чем у эквивалентных GCC приложений C/C++!). С другой стороны, компиляция шаблонов регулярных выражений — это крайне большой объём работы, особенно для сложных шаблонов, это построение кода решающего автомата, как было замечено выше.

Но, в подавляющем большинстве практических случаев, полный вид шаблона регулярного выражения известен (и выверен) уже к началу написания кода, зачастую уже к этому времени мы

знаем синтаксис тех текстовых строк которые предстоит анализировать. А если это так, то шаблон регулярного выражения можно скомпилировать ещё **до начала** выполнения программы. Вот что пишет один из авторов разработки Rust:

Я решительно не имею в виду компиляцию регулярных выражений «заранее». Я имею в виду более буквальный перевод: регулярное выражение преобразуется в собственный код Rust, когда вы компилируете свою программу на Rust. То есть существует (практически) нулевая стоимость компиляции регулярного выражения во время выполнения. Возможно, что более важно, поскольку он скомпилирован в собственный код Rust, ваше регулярное выражение всегда будет работать быстрее.

Для реализации такой возможности используется библиотека (крейт) `lazy_static`. Используем его для преобразования только-что написанного приложения в статическую форму (я не стану здесь усложнять компилируемый шаблон, поскольку его вид не влияет никак на рассматриваемый предмет):

```
$ cargo new regexrs3
     Created binary (application) `regexrs3` package
$ cd regexrs3
$ cargo search lazy_static --limit 5
lazy_static = "1.4.0"           # A macro for declaring lazily evaluated statics in Rust.
lazy-static-include = "3.1.3"   # This crate provides `lazy_static_include_bytes`
                                # and `lazy_static_include_str` macros to replac…
static_init = "1.0.3"           # Safe mutable static and non const static initialization,
                                # and code execution at program startup…
staticinit = "1.0.0"            # Safe mutable static and non const static initialization,
                                # and code execution at program startup…
lazy-regex = "2.3.1"            # lazy static regular expressions checked at compile time
... and 320 crates more (use --limit N to see more)
$ cargo add lazy_static
    Updating crates.io index
      Adding lazy_static v1.4.0 to dependencies.
             Features:
             - spin
             - spin_no_std
$ cargo add regex
    Updating crates.io index
      Adding regex v1.7.0 to dependencies.
             Features:
             + aho-corasick
             + memchr
             + perf
             + perf-cache
             + perf-dfa
             + perf-inline
             + perf-literal
             + std
             + unicode
             + unicode-age
             + unicode-bool
             + unicode-case
             + unicode-gencat
             + unicode-perl
             + unicode-script
             + unicode-segment
             - pattern
             - unstable
             - use_std
$ cat Cargo.toml | grep -A2 dependencies
[dependencies]
lazy_static = "1.4.0"
regex = "1.7.0"
```

Теперь всё готово для отработки самого кода (в этом новом проекте `cargo`):

```
$ cat src/main.rs
use std::io;
use lazy_static::lazy_static;
use regex::Regex;

lazy_static! {
   static ref RE: Regex = Regex::new(r"\d\d\d").unwrap();
}

fn main() {
   loop {
      let mut input = String::new();
      io::stdin().read_line(&mut input).expect("ошибка ввода");
      if 1 == input.len() { break }; // выход
      let buf = &input[0 .. input.len() - 1];
      if 0 == RE.captures_iter(buf).count() {
         println!("no match found");
         println!("--------------------");
         continue;
      }
      for caps in RE.captures_iter(buf) {
         for i in 0..caps.len() {
            let cap = caps.get(i).unwrap();
            println!("{}/{} : {}",
                     cap.start(), cap.end(), cap.as_str());
         }
      }
      println!("--------------------");
   }
}
```

Сборка (или периодическая пересборка для наглядности):

```
$ cargo clean
$ cargo build
   Compiling memchr v2.5.0
   Compiling regex-syntax v0.6.28
   Compiling lazy_static v1.4.0
   Compiling aho-corasick v0.7.20
   Compiling regex v1.7.0
   Compiling regexrs3 v0.1.0 (/home/olej/2022/own.BOOKs/Localiz/regex.cod/regexrs3)
    Finished dev [unoptimized + debuginfo] target(s) in 7.02s
$ ls -l ./target/debug/regexrs3
-rwxrwxr-x 2 olej olej 16322296 дек 27 01:20 ./target/debug/regexrs3
$ file target/debug/regexrs3
target/debug/regexrs3: ELF 64-bit LSB shared object, x86-64, version 1 (SYSV), dynamically
linked, interpreter /lib64/ld-linux-x86-64.so.2,
BuildID[sha1]=9b2884baab8d7c483bc2a8d990f234079081ae0a, for GNU/Linux 3.2.0, with debug_info,
not stripped
$ ldd target/debug/regexrs3
       linux-vdso.so.1 (0x00007ffe791a7000)
       libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f89a78a2000)
       libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f89a787f000)
       libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f89a7879000)
       libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f89a7687000)
       /lib64/ld-linux-x86-64.so.2 (0x00007f89a7bfd000)
```

И проверка выполнением:

```
$ ./target/debug/regexrs3
567
```

```
0/3 : 567
--------------------
67
no match found
--------------------
5A7
no match found
--------------------
123456
0/3 : 123
3/6 : 456
--------------------
```

Kotlin

Регулярные выражения предоставляются Kotlin с его стандартной библиотекой, не требует дополнительных инсталляций.

```
$ cat regexk1.kt
import kotlin.system.*

fun main(args: Array<String>) {
    val pattern : String = args[0]
    val regex = pattern.toRegex()
    while (true) {
        var ввод : String = readln()
        if (0 == ввод.length)  // выход
            exitProcess(0)
        var matches : Sequence<MatchResult>? = regex.findAll(ввод)
        if (!matches!!.any()) {
            println("no match found")
            println("--------------------")
            continue
        }
        for (match in matches)
            for (grp in match.groups)
                println("${grp?.range?.first}/${grp?.range?.last} : ${grp?.value}")
        println("--------------------")
    }
}
```

Запускать это можно разным образом… Для начала первый и простейший способ — используя JRE (Java Runtime Environment) языка Java … наследником которого является Kotlin. Компиляция:

```
$ kotlinc regexk1.kt -include-runtime -d regexk1.jar
```

Выполнение:

```
$ java -jar regexk1.jar "(\d)(\d\d)"
== 987 == 654
3/5 : 987
3/3 : 9
4/5 : 87
10/12 : 654
10/10 : 6
11/12 : 54
--------------------
c1v2n3
no match found
--------------------
```

Для сравнения тут же запустим с теми же параметрами обсуждавшийся выше вариант C++:

```
$ ./regexg2 "(\d)(\d\d)"
== 987 == 654
3/6 : 987
3/4 : 9
4/6 : 87
10/13 : 654
10/11 : 6
11/13 : 54
--------------------
```

Обращаем внимание на 2 обстоятельства:

1. Так тот, так и другой вариант считает позиции в текстовой строке UTF-8 по символам, но не по байтам.
2. Конечная позиция каждого сопоставления в варианте Kotlin не 1 меньше чем в варианте C++. Но это связано только с тем, что библиотека Kotlin предоставляет финальную позицию как «последний символ строки», а в C++ — как «байт за окончанием строки», терминальный байт '\0'… При желании то и другое можно поправить, если хорошо понимать откуда берутся эти цифры.
3. Но из-за такой вот дуальности, а также понимания того откуда она происходит, я менять код пока не буду… «у каждой избушки свои погремушки».

Запуск Kotlin программ

Выше показан один из возможных способов запуска кода, написанного на Kotlin — используя среду выполнения языка Java (виртуальную машину JVM). Повторим команды сборки и запуска:

```
$ kotlinc regexk1.kt -include-runtime -d regexk1.jar
$ java -jar regexk1.jar "(\d)(\d\d)"
== 987 == 654
3/5 : 987
3/3 : 9
4/5 : 87
10/12 : 654
10/10 : 6
11/12 : 54
--------------------
```

Другой способ — это использование для запуска непосредственно Kotlin без использования среды Java:

```
$ kotlinc regexk1.kt -d regexk1k.jar
$ kotlin -classpath regexk1k.jar Regexk1Kt "(\d)(\d\d)"
== 987 == 654
3/5 : 987
3/3 : 9
4/5 : 87
10/12 : 654
10/10 : 6
11/12 : 54
--------------------
```

(Здесь сознательно изменено имя создаваемого архива JAR, чтобы мы могли позже сравнить итоги.)

Возникает вопрос: какие параметры строки запуска мы должны при этом использовать? 1-й параметр (`-classpath ...`) — это **имя файла** архива JAR созданного в результате компиляции. А 2-й параметр — это **имя класса** (а в Java, а значит и Kotlin как его приемнике — всё является классом!) содержащего в своём составе статический метод `main()` запуска программы. Если вы в своей программе не используете заголовочной строки пакета: `package foo` — то это имя класса будет образовываться как конкатенация имени и расширения имени файла исходного кода, записанных с большой буквы. Его можно подсмотреть так:

```
$ jar tf regexk1k.jar
META-INF/MANIFEST.MF
Regexk1Kt.class
META-INF/main.kotlin_module
```

Третий способ состоит в том, чтобы **установить** и использовать Kotlin Native. Kotlin Native — это технология, которая компилирует исходный код Kotlin в двоичные данные целевой платформы[19], не требующие поддержки виртуальных машин. Скомпилированные двоичные данные могут запускаться непосредственно на целевой платформе. В основном она включает в себя внутренний компилятор на основе LLVM и родные библиотеки времени выполнения Kotlin. Kotlin Native включён в состав изделия начиная с версии 1.3:

```
$ kotlin -version
Kotlin version 1.7.20-release-201 (JRE 11.0.17+8-post-Ubuntu-1ubuntu220.04)
```

Итого:

```
$ kotlinc-native regexk1.kt -o regexk1
$ ls -l regexk1.kexe
-rwxrwxr-x 1 olej olej 2084152 дек 27 22:46 regexk1.kexe
$ file regexk1.kexe
regexk1.kexe: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked,
interpreter /lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.16,
BuildID[sha1]=35bf922f1e11448be31a25a523aaef7a6dfa741a, not stripped
$ ldd regexk1.kexe
        linux-vdso.so.1 (0x00007ffc8f0ca000)
        libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f9f5c8e7000)
        libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f9f5c798000)
        libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f9f5c775000)
        libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f9f5c75a000)
        libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f9f5c568000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f9f5c915000)
$ ./regexk1.kexe "(\d)(\d\d)"
== 987 == 654
3/5 : 987
3/3 : 9
4/5 : 87
10/12 : 654
10/10 : 6
11/12 : 54
--------------------
```

В конечном итоге, можем сравнить (не в смысле предпочтений, а только для констатации фактов):

```
$ ls -l regexk1*
-rw-rw-r-- 1 olej olej 4694586 дек 28 00:39 regexk1.jar
-rwxrwxr-x 1 olej olej 2084152 дек 28 00:40 regexk1.kexe
-rw-rw-r-- 1 olej olej    2213 дек 28 00:39 regexk1k.jar
-rw-rw-r-- 1 olej olej     706 дек 27 22:28 regexk1.kt
```

Два варианта JAR различаются размерами в 2121 раз! Но это не должно вызывать удивления, если мы рассмотрим содержимое архивов:

```
$ unzip -l regexk1k.jar
Archive:  regexk1k.jar
  Length      Date    Time    Name
---------  ---------- -----   ----
       78  1980-01-01 00:00   META-INF/MANIFEST.MF
     3084  1980-01-01 00:00   Regexk1Kt.class
       39  1980-01-01 00:00   META-INF/main.kotlin_module
---------                     -------
     3201                     3 files
```

19 Kotlin Native работает на множестве аппаратных платформ и операционных систем.

И вариант собранный под исполнение виртуальной машиной JVM:

```
$ unzip -l regexk1.jar | grep class$ | wc -l
2802
$ unzip -l regexk1.jar | head
Archive:  regexk1.jar
  Length      Date    Time    Name
---------  ---------- -----   ----
       78  1980-01-01 00:00   META-INF/MANIFEST.MF
     3084  1980-01-01 00:00   Regexk1Kt.class
       39  1980-01-01 00:00   META-INF/main.kotlin_module
      596  1980-01-01 00:00   kotlin/collections/ArraysUtilJVM.class
     2721  1980-01-01 00:00   kotlin/jvm/internal/AdaptedFunctionReference.class
      898  1980-01-01 00:00   kotlin/jvm/internal/CallableReference$NoReceiver.class
     4173  1980-01-01 00:00   kotlin/jvm/internal/CallableReference.class
```

Этот вариант содержит, кроме того же (судя по размеру) класса байт-кода `Regexk1Kt.class`, ещё 2801 классов исполняющей системы (runtime).

Ну и, для полноты картины, сравним скорость подготовки (компиляции) приложений в разных вариантах:

```
$ time kotlinc regexk1.kt -include-runtime -d regexk1.jar
real    0m6,722s
user    0m21,178s
sys     0m1,711s
$ time kotlinc regexk1.kt -d regexk1k.jar
real    0m5,353s
user    0m20,444s
sys     0m1,031s
$ time kotlinc-native regexk1.kt -o regexk1
real    0m23,788s
user    0m29,527s
sys     0m1,926s
```

При компиляции в нативный код приходится сильно потрудиться … в 4.5 раза дольше. При сборке реально крупных проектов это может действительно стать досадной проблемой.

Использование регулярных выражений

Регулярные выражения, будучи одной из форм выражения программы деятельности конечных автоматов, являются в умелых руках чрезвычайно мощным, и часто недооценённым инструментом. Из-за своей непривычной формы они кажутся чем-то чрезмерно сложным, но это не совсем так: они не столько сложны, сколько необычны. После их изучения, даже не слишком обстоятельного, их использование становится достаточно простым и понятным.

Собственно, сама техника составления и использования регулярных выражений не является предметом настоящих заметок. Но в перечне литературы, в конце текста, показаны книги, изданные в русских переводах, которых более чем достаточно для самого замысловатого использования регулярных выражений.

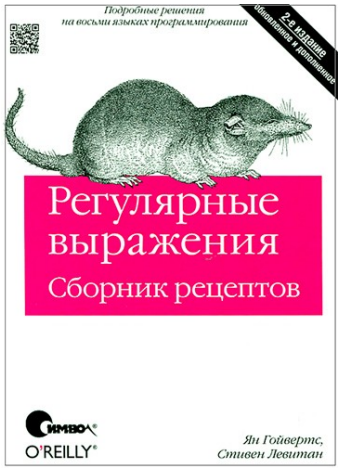
Литература и сетевые ресурсы

4. Regular Expressions, The Open Group Base Specifications Issue 7, 2018 edition
   https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap09.html
5. Реализация механизма обработки регулярных выражений на языке C++
   http://rus-linux.net/nlib.php?name=/MyLDP/algol/cpattern/Regular_Expressions_in_C_ru.html
6. Юникод — https://ru.wikipedia.org/wiki/Юникод
7. Кириллица в Юникоде — https://ru.wikipedia.org/wiki/Кириллица_в_Юникоде
8. UTF-8 — https://ru.wikipedia.org/wiki/UTF-8
9. UTF-8, a transformation format of ISO 10646, регламент стандарта UTF-8
   https://www.rfc-editor.org/rfc/rfc3629
10. REGEXP(5) — https://housecomputer.ru/os/unix/program/REGEXP.5.shtml.htm
11. regexp (5), Solaris man — http://www.opennet.ru/man.shtml?category=5&topic=regexp
12. Regular Expressions in C, 2020-02-01 — https://lloydrochester.com/
13. Регулярные выражения в C++: Использование библиотеки PCRE
    http://www.opennet.ru/base/dev/pcre_cpp.txt.html
14. Библиотека регулярных выражений — https://ru.cppreference.com/w/cpp/regex
15. Функция regex_search Visual Studio 2015
    https://learn.microsoft.com/en-us/cpp/standard-library/regex-functions?view=msvc-170&redirectedfrom=MSDN&viewFallbackFrom=vs-2017
16. RegEx Pal, онлайн тестер регулярных выражений — https://www.regexpal.com
17. Регулярные выражения в Python от простого к сложному. Подробности, примеры, картинки, упражнения — https://habr.com/ru/post/349860/
18. re — Regular expression operations (документация) — https://docs.python.org/3/library/re.html
19. Примеры применения регулярных выражений в Python
    https://pythonru.com/primery/primery-primeneniya-regulyarnyh-vyrazheniy-v-python
20. Регулярные выражения в Python за 5 минут: теория и практика для новичков и не только
    https://proglib.io/p/regulyarnye-vyrazheniya-v-python-za-5-minut-teoriya-i-praktika-dlya-novichkov-i-ne-tolko-2022-04-05
21. Путешествие в golang regexp — https://tproger.ru/articles/puteshestvie-v-golang-regexp
22. Регулярные выражения с Go: часть 1
    https://code.tutsplus.com/ru/tutorials/regular-expressions-with-go-part-1--cms-30403
23. Go. Standard library, regexp (документация) — https://pkg.go.dev/regexp@go1.19.4
24. Rust. Crate regex (документация) — https://docs.rs/regex/latest/regex/
25. regex v1.7.0 — https://crates.io/crates/regex
26. Crate regex — https://docs.rs/regex/latest/regex/
27. All Crates (репозиторий крейтов Rust) — https://crates.io/crates
28. Regular Expressions in Kotlin, December 2, 2021
    https://translated.turbopages.org/proxy_u/en-ru.ru.afa03f0b-639a18ef-8cdd49f0-74722d776562/https/www.baeldung.com/kotlin/regular-expressions
29. Основы Kotlin. Регулярные выражения RegExp
    https://www.fandroid.info/6-5-osnovy-kotlin-doktor-regexp/2/
30. Класс String в Kotlin – подробно про строки, 13.04.2022

https://java-blog.ru/kotlin/klass-string-v-kotlin-podrobno-pro-stroki

31. Friedl J. / Фридл Дж. - Mastering Regular Expressions / Регулярные выражения (3-е издание), 2008г., СПб "Символ-Плюс", ISBN: 5-93286-121-5, 608 страниц

32. Ян Гойвертс, Стивен Левитан, Регулярные выражения. Сборник рецептов, 2-е издание, 2015г., СПб "Символ-Плюс", ISBN: 978-5-93286-221-6, 704 страницы



33. Майкл Фицджеральд, Регулярные выражения. Основы, 2015г., "Вильямс", ISBN: 978-5-8459-1953-3, 144 страниц