

Про



и почему проблем не стало меньше

Почти реферат

640 килобайт хватит всем

Или 16 бит хватит всем. В 1988 году Джозеф Бекер написал [основополагающую работу](#), с которой, пожалуй, можно отсчитывать историю стандарта Unicode. Вероятно, этой работой мы в итоге обязаны тому, что 16-битная кодировка была выбрана в WinAPI (Windows NT 3.1 – 1993 год), в Java (1995 год) и, впоследствии, 16-битный формат вошёл и в C# например.

Выбранное для этого стандарта название Unicode было призвано символизировать единый, универсальный и однозначный способ кодирования символьной информации. Иногда кое-где у нас порой последовательность символов Unicode именуют Unitext. Забегая вперёд отметим, что это кажется правильным, чтобы подчеркнуть отличие от привычных нам строк символов в привычных нам же языках программирования.

Что в указанной работе Бекера ещё важно: предложено разделить собственно символ – [графему](#) – от его начертания – [глифа](#) (или набора глифов, поскольку символ может состоять из нескольких элементов). Можно также сказать, что кодирование Unicode определяется как способ задать уникальный номер (code point или пойнт-код) для любого начертания, используемого для письма.

Предложенный Бекером способ кодирования нашёл отражение в формате UCS – Universal Coded Character Set, способ кодирования с фиксированной длиной кода (fixed-length encoding). UCS – это первая редакция стандарта до версии 1.1 включительно. Для нас важно, что в те далёкие времена не определялись так называемые surrogate code points (в русском техническом это термин «суррогатные пары»).

Тогда всем казалось, что достаточно сделать переход от 8 бит к 16 битам и это решит проблемы человечества с кодировками используемых языков общения.

Тем не менее, науке с тех пор известен не только UCS-2, но и UCS-4 – 32-х битный вариант. Последний пришлось придумать, когда в информационную сферу массово пришли азиаты с их объёмными азбуками и — внезапно! — 16 бит категорически перестало хватать.

Здесь сразу отметим очевидные преимущества UCS-2 и UCS-4 – постоянная длина кодов и простые алгоритмы кодирования, а потому высокая скорость обработки текста в таких кодировках. Это же преимущество изрядно компенсировалось расточительностью в использовании памяти, особенно для европейских и близких к европейским азбук. Причём это очевидно для UCS-4, и верно (хотя и не столь очевидно) для UCS-2.

Итак, оказалось, что 16-бит недостаточно, а 32 бита *обычно* слишком много. И уже где-то примерно в 1996 году произошло изобретение UTF-16, который, в отличие от той же UCS-2, был в состоянии работать с кодами большей длиной, чем 16-бит, и был, по сути, кодом переменной длины. Случился плавный и незаметный переход от кодировок с фиксированной длиной к кодировкам переменной длины, то есть, к UTF-16. Именно UTF-16 используется для представления «широких» символов в Windows, равно как символьный тип в C# и Java, и в некоторых юникодных или использующих Unicode C++ библиотеках (Qt, ICU). О родных для C++ так называемых «широких» символах `wchar_t` и строках `std::wstring` надо будет поговорить отдельно, поскольку бытует поверье, что вот это в C++ сделано не вполне очевидным и прямым способом (как с тем же платформенно-зависимым типом `int`).

В настоящее время стандарт Unicode определяет 109449 символов, причём около 74500 символов входят в так называемую группу CJK (China-Japan-Korea).

С момента появления UTF-16 начинается эпоха Unicode 2.0. UTF – это Unicode Transformation Format, формат кодирования с переменной шириной (variable-width encoding).

В качестве небольшого необязательного отступления

ISO/IEC-10646 и Unicode – это не одно и то же. В целом, коды символов и способы кодирования совпадают в этих стандартах (более того, каждая редакция Unicode содержит ссылку на соответствующий ей 10646 документ), но Unicode дополнительно определяет ограничения на реализацию стандарта на разных платформах, алгоритмы, описания функциональных символов, которые не входят в ISO-10646.

И ещё. Тем кто застал ASCII и MS-DOS будет небезынтересно узнать, что ISO-8859-1 входит в ISO-10646 в качестве первых 256 пойнт-кодов.

Немного определений

Code point или пойнт-код (википедия, например, [использует](#) такой русскоязычный термин при описании OKS-7, но эта же калька кажется пригодной и в данном случае) – это уникальный номер, представляющий любой символ, который мы используем или могли бы использовать при письме. Кодовый элемент (code unit) – это базовый элемент закодированной последовательности, представляет из себя минимальную комбинацию бит, которая может представлять текстовый элемент; своего рода дескриптор символа. Звучит сложно, но: для UTF-8 code unit состоит из 8 бит, для UTF-16 – из 16 бит, и для UTF-32 – из 32-х бит. Абстрактный символ (abstract character) – это единица информации, которая используется для организации, представления или управления текстовыми данными. Закодированный символ (encoded character) – это сопоставление абстрактного символа и code

point. Один и тот же внешне одинаковый символ технически может иметь в Unicode разные кодировки или способы кодирования, например, греческая буква Ω (омега) или Ом — символ сопротивления в радиотехнике (например, 5Ω). И наоборот, один и тот же символ может иметь разные способы кодирования: используемое в испанском языке сочетание **Í** может кодироваться как один поинт-код U+0140 или два поинт-кода U+006C U+00B7 (причем второй вариант стандарт определяет как предпочтительный).

Пользовательский символ (user-perceived character) — это всё, что конечный пользователь воспринимает или готов воспринимать как отдельный письменный символ.

Пример: **ю́** – U+044E кириллическая строчная буква **ю**, за которой следует комбинирующий код ударения U+0301, то есть, собственно символ как его видит пользователь состоит из двух поинт-кодов.

Кластер графем (grapheme cluster) – это последовательность закодированных символов, которые следует воспринимать вместе (например, **ch**).

Когда пользователь интересуется длиной текстовой строки его, как правило, интересует длина в пользовательских символах (user-perceived characters). Это настолько важный момент, что в некоторых API способ вычисления длины текста специфицируется отдельно (например, Twitter API содержит специальное упоминание о том, как именно высчитывается длина сообщения — твита).

Ещё одно необязательное, но важное отступление

Важный момент стандарта Unicode, но не вполне очевидный с первого взгляда: отказ от рассмотрения текста как простого массива символов, то есть, привычных нам строк. Правильнее верить, что мы работаем с массивом глифов, а не графем. Такой подход немножко ломает стереотипы того, как надо программировать обработку и преобразование текста (определение длины строки, поиск подстроки в строке, разбиение, вставка символов и подстрок и прочее). Кажется, что с точки зрения производительности кода, например, для подсчёта количества символов, правильнее использовать коды с фиксированной длиной или пошире (UCS-2, UTF-16, UTF-32), чтобы на всё хватило, и кодировать сразу целые символы (графемы). Однако, Unicode определяет, что не все поинт-коды соответствуют закодированным символам, некоторые могут быть *не-символами* (символы перемещения курсора, служебная информация и прочее) То есть, просто так всё равно подсчитать не получится. Поэтому, например, вычисление длины текста в юникодных символах в общем случае не делается за известное константное время.

Несколько примеров на Java (потому что на Java проще)

В немецком языке есть буква — эсцет.

Даже простой toupper/tolower может изменить длину строки.

BOM – Byte Order Mark

Эта штука появилась вследствие того, что некоторые варианты кодирования, а, точнее, все, кроме UTF-8, имеют размер кодового элемента больше одного байта. То есть, строго говоря, *только для UTF-8 BOM*, в принципе, **не** нужен. Нужно только джентльменское соглашение

«вот мы тут везде используем UTF-8». Да и в нём — как будет показано ниже — тоже нет особой необходимости, поскольку UTF-8 мало того, что легко детектируется, так он ещё и обладает свойством самосинхронизации. Об этом мы тоже скажем позднее.

А вот для UTF-16 и UTF-32 BOM иметь почти обязательно (можно, конечно, тоже как-то договориться, но на практике ошибок делается всё равно слишком много). Более того, по понятным причинам для UTF-16 и UTF-32 нужно целых два BOM'а: для **Little Endian** и для **Big Endian**; то есть, у нас должны быть определены маркеры UTF-16BE, UTF-16LE, UTF-32BE, UTF-32LE.

Итак, у нас в наличии вот такой список BOM'ов:

UTF-8	0xEF 0xBB 0xBF
UTF-16BE	0xFE 0xFF
UTF-16LE	0xFF 0xFE
UTF-32BE	0x00 0x00 0xFE 0xFF
UTF-32LE	0xFF 0xFE 0x00 0x00

В этом смысле сложилась интересная ситуация в Windows, которая, как мы теперь уже знаем, использует в своём API UTF-16 для представления юникодных символов и строк; они в WinAPI часто сопровождаются буквосочетанием WC(S) – Wide Character (String). При этом Windows не накладывает никаких ограничений на способы кодирования текстовой информации в прикладных программах, можно использовать как UTF-16BE, так и UTF-16LE, если выбрать и установить подходящий BOM. Но внутренний формат в Windows – всегда UTF-16LE, который используется в WinAPI для юникодных версий системных вызовов, он же используется для задания имён файлов в файловых системах NTFS и FAT (если используемый вариант последней поддерживает длинные имена файлов).

Особенности UTF-8

- Символ кодируется последовательностью байтов
- Байты последовательности, начиная со второго, всегда начинаются с битов 10
- Если первый байт последовательности единственный, то он всегда начинается с бита 0, то есть символ кодируется одним байтом и соответствует коду ASCII
- Если символ не ASCII, то первые биты первого байта содержат столько единиц, сколько байтов в последовательности (включая первый байт), после чего идёт бит 0
- Все последующие значащие биты склеиваются в последовательность битов и интерпретируются как код символа

Таким образом, способ кодирования UTF-8 позволяет в потоке однозначно определить, что это а) Unicode, б) найти границы символов. Другие способы кодирования (UTF-16 и UTF-32) такими свойствами не обладают. К недостаткам UTF-8 относят необходимые расходы на перенос служебной информации.

Есть точка зрения, что эффективность и простота UTF-8 побудила Google определить `wchar_t` как тип длиной в 1 байт в Android NDK, чтобы в конце концов заставить пользователей отказаться от широких символов в пользу UTF-8. Авторитет Google сомнению не подлежит, но этот шаг всё равно не отменяет наличие `wchar_t` и проблем, с этим типом связанных. Потому что



Использование UTF-8 с `char` и `std::string`

Итак, с появлением UTF-16 и «широких» символом возникла необходимость добавить в C/C++ поддержку нового стандарта. По соображениям обратной совместимости и из-за равенства `sizeof(char) == 1` исправлять `char` не стали (очень хорошо!), а просто добавили: новый тип строкового литерала `L`, тип `wchar_t` для «широких» символов и, соответственно, `std::wstring` для работы со строками таких символов. Как показала практика, решение вышло умеренно удачным.

Во-первых, `wchar_t` на разных платформах имеет разную длину. В Linux он, как правило, 32-битный, в Windows размер `wchar_t` равен 16 битам, а в Android он и вовсе равен одному байту. Написание переносимого кода получило ещё одно препятствие. Плюс `wchar_t` и `std::wstring`, оказалось, имеют вирусную природу: очень трудно использовать в одной программе одновременно и `char` (`std::string`) и `wchar_t` (`std::wstring`) из-за сложностей с взаимным перекодированием и использованием смешанных внешних API (частично с обычными символами, частично — с «широкими»).

В этом смысле UTF-8 представляет собой хорошую альтернативу новым типам символов и строк:

1. Минимальный размер кода символа — 1 байт, то есть, UTF-8 укладывается в единицы типов `char`, а это очень удобно
2. У нас остались привычные типы `char` и `std::string`
3. Прозрачно совместим с ASCII
4. Корректная работа алгоритмов поиска подстроки в строке (не найдётся строка с началом в середине символа, например)
5. Достаточно компактен
6. Там, где требуется использовать «широкие» символы или строки символов можно выполнять перекодирование на лету, пользуясь уже известными API; затраты на это обычно не слишком велики (см., например, рекомендации в [3] и [5])

В недостатки использования UTF-8 можно записать необходимость перестраивать строку при операциях вставки и удаления символа, то есть, использование UTF-8 может существенно

замедлить выполнение кода, активно использующего такие операции. Более того, как мы уже знаем из примеров выше и общих рассуждений, преобразования текстовых строк в общем случае являются операциями не тривиальными и, скорее всего, для каждого языка нужно будет программировать поддержку отдельно.

В целом, нет никаких объективных причин предпочесть один определённый способ кодирования Unicode. Но, по сумме факторов именно UTF-8 является наиболее удачным вариантом для использования, по крайней мере, в C/C++ точно. В общем, если не знаешь, что выбрать, а выбрать надо — выбирай UTF-8.

Литература

1. <http://unicode.org/>
2. <http://unicode.org/history/unicode88.pdf>
3. <http://utf8everywhere.org/>
4. <http://www.italiancpp.org/2016/04/20/unicode-localization-and-cpp-support/>
5. <https://habrahabr.ru/company/xakep/blog/257895/>