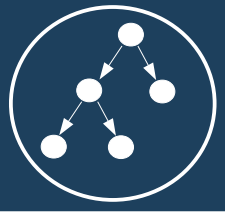


Data Structures and Algorithms

Алгоритмы.
Строки. Способы хранения на ПК

Строка определение

Строка — последовательность символов алфавита. Строки играют очень важную роль в представлении и обработке информации. В современном мире большинство наиболее важной информации закодировано и хранится в виде строк(книги, сайты, языки общения). Строки являются настолько привычным и удобным способом представления информации, что часто используются для представления не символьных по своей природе данных (например в геномике ДНК сводиться к последовательности из 4 символов). Поэтому алгоритмы обработки строк играют важную роль в программировании.

Как строки хранятся в памяти ПК

Строка представляется в виде массива (символов, байт). Размер строки равен размеру массива.

Преимущества: размер строки известен заранее, поэтому копирование элемента, получение размера строки работает быстро. Легко получить любой символ по его индексу (даже при отсчете от конца строки). Легко отслеживать возможность выхода за границы строки.

Недостатки: проблемы с хранением строк произвольной длинны. Ограничение максимальной длинны строки (в больше случаем до 4 Гб). При использовании алфавита с переменным размером символа (например, UTF-8), в размере хранится не количество символов, а именно размер строки в байтах, поэтому количество символов необходимо считать отдельно.

Как строки хранятся в памяти ПК

Строка представляется в виде участка памяти заполненного символами (их байт - представление). Конец строки обозначается специальным символом (например нуль-терминированный строки, где таким символом является символ с кодом 0).

Преимущества: возможность представления строки без создания отдельного типа данных (просто хранить последовательность байт в выделенном участке памяти). Отсутствие ограничения на максимальный размер строки. Простота получения суффикса строки. Простота передачи строк в функции (передаётся указатель на первый символ).

Недостатки: долгое выполнение операций получения длины и конкатенации строк. Отсутствие средств контроля за выходом за пределы строки, в случае повреждения завершающего байта возможность повреждения больших областей памяти, что может привести к непредсказуемым последствиям — потере данных, краху программы и даже всей системы. Невозможность использовать символ завершающего байта в качестве элемента строки. Невозможность использовать некоторые кодировки с размером символа в несколько байт (например, UTF-16), так как во многих таких символах, например Ā (0x0100), один из байтов равен нулю (в то же время, кодировка UTF-8 свободна от этого недостатка).

Как строки хранятся в памяти ПК

Строка представляется в виде списка символов (или их байт представления).
Преимущества: Есть возможность контроля длинны строки. Легкость конкатенации строк.
Недостатки: Низкая скорость работы.

Как строки хранятся в памяти ПК

Массив → [H,e,l,l,o, ,w,o,r,l,d]

Последовательность в памяти → H,e,l,l,o, ,w,o,r,l,d,Ø

Список → H → e → l → l → o → → w → o → r → l → d

В некоторых языках программирования может применяться смешанный подход.

Хранение строк на ПК

Название языка программирования	Версия	Способ хранения
Python	3.9	Массив байт. На уровне интерпретатора кодировка UTF-32. На уровне использования, оптимизационная на основе данных строки.
Java	11	Массив байт кодировка UTF-16
Fortran	2008	Массив байт. Кодировка указывается явно при создании (ASCII или UTF-8,16). Поддерживаются нуль-терминированные строки (для совместимости с C).

Способ представления символа

Исторически первым стандартом для хранения символов можно назвать ASCII (1963 год). Предложена кодировка в которой использовались 7 бит (старший бит изначально использовался для контроля ошибок). Символу ставилось в соответствие целое число (записывается в двоичной системе счисления с помощью 7 бит) количество символов 128. Для указания какому символу, какое целое число соответствует использовалась таблица (часто называлась таблица ASCII).

Изначально в таблице хранились символы для представления:

- десятичных цифр;
- латинского алфавита;
- знаков препинания;
- управляющих символов.

В последствии 8-й бит стали использовать для представления символов «национальных алфавитов». Т.е. для каждого языка можно было использовать символы от 128 до 255.

ASCII таблица пример

Символ	Десятичное	Двоичное	Шестнадцатеричное
A	65	1000001	41
B	66	1000010	42
C	67	1000011	43
D	68	1000100	44
E	69	1000101	45
F	70	1000110	46
G	71	1000111	47
H	72	1001000	48
I	73	1001001	49
J	74	1001010	4A
K	75	1001011	4B
L	76	1001100	4C
M	77	1001101	4D
N	78	1001110	4E
O	79	1001111	4F

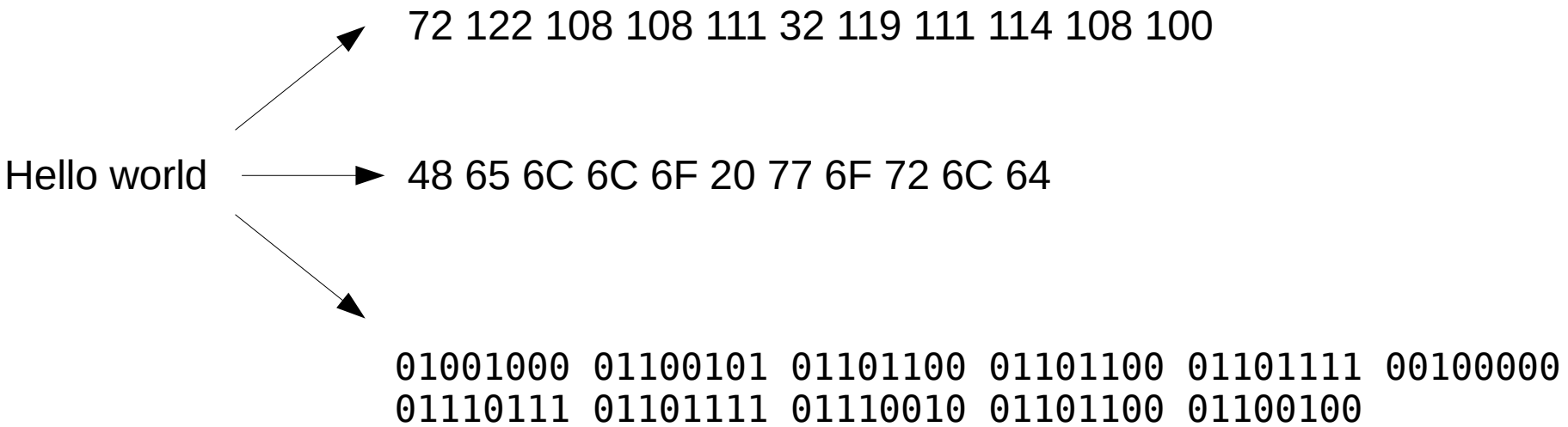
Символ	Десятичное	Двоичное	Шестнадцатеричное
	32	0100000	20
!	33	0100001	21
"	34	0100010	22
#	35	0100011	23
$	36	0100100	24
%	37	0100101	25
&	38	0100110	26
'	39	0100111	27
(	40	0101000	28
)	41	0101001	29
*	42	0101010	2A
+	43	0101011	2B
,	44	0101100	2C
-	45	0101101	2D
.	46	0101110	2E

Важные особенности ASCII

- Коды символов цифр «0»-«9» в двоичной системе счисления начинаются с 0011, а заканчиваются двоичными значениями чисел. Например, 0101 - число 5, а 0011 0101 — символ «5». Зная об этом, можно преобразовать двоично-десятичные числа (BCD) в ASCII-строку с помощью простого добавления слева 0011 к каждому двоично-десятичному полубайту.

- Буквы «A»-«Z» верхнего и нижнего регистров различаются в своём представлении только одним битом, что упрощает преобразование регистра и проверку на принадлежность кода к диапазону значений. Буквы представляются своими порядковыми номерами в алфавите, записанными пятью цифрами в двоичной системе счисления, перед которыми стоит 010 (для букв верхнего регистра) или 011 (для букв нижнего регистра).

Пример кодирования строки



Python

Реализация алгоритма на Python

Кодирование строки в последовательность байт

```
def code_to_ASCII(text):
    code_points = []
    for sym in text:
        code_point = ord(sym)
        if code_point > 128:
            raise ValueError("String contains erroneous characters ")
        code_points.append(code_point)
    return bytes(code_points)
```

Декодирование строки из последовательности байт

```
def decode_from_ASCII(byte_sequince):
    result = ""
    for b in byte_sequince:
        result = result+chr(b)
    return result
```

Недостатки ASCII

ASCII кодировка обладает рядом недостатков. Это малый диапазон представляемых символов (255) что не позволяет разместить полностью все символы некоторых национальных алфавитов. При учете, что первые 128 символов зарезервированы стандартом это количество еще меньше. Это привело к появлению большого количества несовместимых между собой кодировок. Требовалось расширение количества символов. Это было реализовано в Unicode.

Unicode

Unicode - стандарт кодирования символов, включающий в себя знаки почти всех письменных языков мира. В настоящее время (2021 год) стандарт является преобладающим.

По своей сути стандарт Unicode определяет таблицу символов размером 1 114 112 символов и алгоритмы и способы соответствия позиции и символа. Разделено это общее пространство на 17 блоков, по 65 536 символов в каждом. Такой блок называется плоскостью. Каждая плоскость содержит свою группу символов. В нулевой все основные символы.

Плоскости Юникода:

- Плоскость 0 (0000—FFFF): Основная многоязычная плоскость
- Плоскость 1 (10000—1FFFF): Дополнительная многоязычная плоскость
- Плоскость 2 (20000—2FFFF): Дополнительная идеографическая плоскость
- Плоскость 3 (30000—3FFFF): Третичная идеографическая плоскость
- Плоскости 4—13 (40000—DFFFF) не используются
- Плоскость 14 (E0000—EFFFF): Специализированная дополнительная плоскость
- Плоскость 15 (F0000—FFFFF) Дополнительная область для частного использования — A
- Плоскость 16 (100000—10FFFF) Дополнительная область для частного использования — B

Способы записи и представления

Универсальный набор символов перечисляет допустимые по стандарту Юникод символы и присваивает каждому символу код в виде неотрицательного целого числа, записываемого обычно в шестнадцатеричной форме с префиксом U+, например, U+040F. Семейство кодировок определяет способы преобразования кодов символов для передачи в потоке или в файле.

Юникод имеет несколько форм представления (англ. Unicode transformation format, UTF):

- UTF-8
- UTF-16 (UTF-16BE, UTF-16LE)
- UTF-32 (UTF-32BE, UTF-32LE).

Порядок байтов

Один символ может быть представлен с помощью нескольких байт. Как следствие можно задать два способа описания порядка следования байт. Систему, совместимую с процессорами x86, называют little endian, а с процессорами m68k и SPARC — big endian.

Для определения порядка байтов используется метка порядка байтов (англ. Byte order mark - BOM). В начале файла записывается код указывающий и на кодировку и на порядок следования байт.

UTF-8
EF BB BF
UTF-16BE
FE FF
UTF-16LE
FF FE
UTF-32BE
00 00 FE FF
UTF-32LE
FF FE 00 00

Кодировка UTF-8

UTF-8 — представление Юникода, обеспечивающее наибольшую компактность и обратную совместимость с 7-битной системой ASCII; текст, состоящий только из символов с номерами меньше 128, при записи в UTF-8 превращается в обычный текст ASCII и может быть отображён любой программой, работающей с ASCII; и наоборот, текст, закодированный 7-битной ASCII может быть отображён программой, предназначенной для работы с UTF-8. Остальные символы Юникода изображаются последовательностями длиной от 2 до 4 байт, в которых первый байт всегда имеет маску 11xxxxxx, а остальные — 10xxxxxx.

Таким образом один символ в UTF-8 может занимать 1,2,3 или 4 байта.

Кодировка UTF-8 (определение сколько кол-вы байт)

Первое что нужно выполнить при кодировании символа с помощью UTF-8 это определить, сколько байт нужно для его представления.

Диапазон номеров символов	Требуемое количество байт
00000000-0000007F	1
00000080-000007FF	2
00000800-0000FFFF	3
00010000-0010FFFF	4

Например символ (U+263A) требует для своего представления 3 байта, так как лежит в диапазоне от 800-FFFF.

Кодировка UTF-8 (Установка старших бит в первом байте)

Нужно установить старшие биты первого байта в соответствии с необходимым количеством байт, определённом на первом этапе:

- 0xxxxxxx — если для кодирования потребуется один байт;
- 110xxxxx — если для кодирования потребуется два байт;
- 1110xxxx — если для кодирования потребуется три байт;
- 11110xxx — если для кодирования потребуется четыре байт.

Для символа (U+263A) требуется 3 байта, значит для первого байта следует установить значение равное 1110xxxx.

Как видно если для хранения символа нужен 1 байт, то старший бит равен 0. Таким образом мы получаем 7 битную ASCII кодировку. UTF-8 обратно совместим с ASCII.

Кодировка UTF-8 (Установка старших бит в остальных байтах)

Если для кодирования требуется больше одного байта, то в остальных байтах два старших бита всегда устанавливаются равными 10 (10xxxxxx). Это позволяет легко отличать первый байт в потоке, потому что его старшие биты никогда не равны 10.

Количество байт	Шаблон
1	0xxxxxxx
2	110xxxxx 10xxxxxx
3	1110xxxx 10xxxxxx 10xxxxxx
4	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

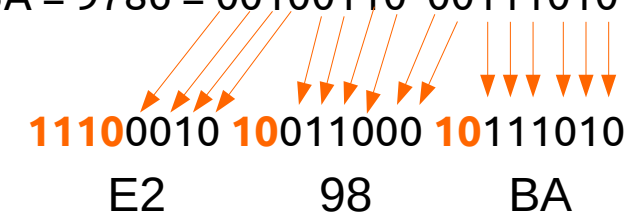
Для символа (U+263A) требуется 3 байта, значит требуемый шаблон кодирования равен 1110xxxx 10xxxxxx 10xxxxxx .

Кодировка UTF-8 (Установка значащих бит)

Установить значащие биты в байте в соответствии с номером символа Юникода, выраженном в двоичном виде. Начать заполнение с младших битов номера символа, поставив их в младшие биты последнего байта, продолжить справа налево до первого байта. Свободные биты первого байта, оставшиеся незадействованными, заполнить нулями.

Для символа (U+263A)

263A = 9786 = 00100110 00111010



Для указания, что файл или поток содержит символы Юникода, в начале файла или потока может быть вставлен маркер последовательности байтов (англ. Byte order mark, BOM), который в случае кодирования в UTF-8 принимает форму трёх байтов: EF BB BF = 11101111 10111011 10111111

Java

Реализация алгоритма на Java

Методы для кодирования одного символа в UTF-8

```
public static int numberOfBytes(int codePoint) {
      if (codePoint <= 0x7F) {
            return 1;
      } else if (codePoint <= 0x7FF) {
            return 2;
      } else if (codePoint <= 0xFFFF) {
            return 3;
      }
      return 4;
}

public static byte[] codeSymbol(int codePoint) {
      int numberOfBytes = numberOfBytes(codePoint);
      if (numberOfBytes == 1) {
            return new byte[] { (byte) codePoint };
      }
      byte[] result = new byte[numberOfBytes];
      int mask = 0b111111;
      int firstByte = 0b10000000;
      for (int i = numberOfBytes - 1; i > 0; i--) {
            result[i] = (byte) ((codePoint & mask) | 0b10000000);
            codePoint = codePoint >> 6;
            firstByte = (firstByte >> 1) | firstByte;
      }
      result[0] = (byte) (firstByte | codePoint);
      return result;
}
```

Методы для кодирования строки в UTF-8

```
public static byte[] codeToUTF8(String text) throws IOException {
    ByteArrayOutputStream os = new ByteArrayOutputStream();
    char[] symbols = text.toCharArray();
    for (int i = 0; i < symbols.length; i++) {
        int codePoint = Character.codePointAt(symbols, i);
        os.write(codeSymbol(codePoint));
    }
    return os.toByteArray();
}
```

Методы для декодирования символа в UTF-8

```java
public static int numberOfNextBytes(byte firstByte) {
	int nb = 1;
	if ((firstByte & 0b10000000) == 0) {
		return 1;
	}
	for (int i = 1; i <= 4; i++) {
		firstByte = (byte) (firstByte << 1);
		if ((firstByte & 0b10000000) == 0) {
			nb = i;
			break;
		}
	}
	return nb;
}

public static int getSymbolCodePoint(byte[] byteArray, int start, int end) {
	if (start == end) {
		return byteArray[start];
	}
	int result = (255 & byteArray[start]) & ((1 << (6 - (end - start))) - 1);
	for (int i = start + 1; i <= end; i++) {
		result = result << 6;
		result = result | ((255 & byteArray[i]) & 0b111111);
	}
	return result;
}
```

Методы для декодирования строки в UTF-8

```
public static String decodeFromUTF8(byte[] bytes) {
     int start = 0;
     int end = start + numberOfNextBytes(bytes[start]) - 1;
     String result = Character.toString(getSymbolCodePoint(bytes, start, end));
     for (;;) {
          start = end + 1;
          if (start >= bytes.length) {
               break;
          }
          end = start + numberOfNextBytes(bytes[start]) - 1;
          result += Character.toString(getSymbolCodePoint(bytes, start, end));
     }
     return result;
}
```

Кодировка UTF-16

UTF-16 - один из способов кодирования символов из Юникода в виде последовательности 16-битных слов. При этом каждый символ записывается одним или двумя словами (суррогатная пара).

Принцип кодирования UTF-16

В UTF-16 символы кодируются двухбайтовыми словами с использованием всех возможных диапазонов значений (от 0 до $FFFF_{16}$). При этом можно кодировать символы Unicode в диапазонах $0000_{16}..D7FF_{16}$ и $E000_{16}..FFFF_{16}$. Исключенный отсюда диапазон $D800_{16}..DFFF_{16}$ используется как раз для кодирования так называемых суррогатных пар — символов, которые кодируются двумя 16-битными словами.

Символы Unicode до $FFFF_{16}$ включительно (исключая диапазон для суррогатов) записываются как есть 16-битным словом (2 байта).

Символы же в диапазоне $10000_{16}..10FFFF_{16}$ (больше 16 бит) кодируются по следующей схеме:

Из кода символа вычитается 10000_{16}. В результате получится значение от нуля до $FFFFF_{16}$, которое помещается в разрядную сетку 20 бит.

Старшие 10 бит (число в диапазоне $0000_{16}..03FF_{16}$) суммируются с $D800_{16}$, и результат идёт в ведущее (первое) слово, которое входит в диапазон $D800_{16}..DBFF_{16}$.

Младшие 10 бит (тоже число в диапазоне $0000_{16}..03FF_{16}$) суммируются с $DC00_{16}$, и результат идёт в последующее (второе) слово, которое входит в диапазон $DC00_{16}..DFFF_{16}$.

Пример кодирование в UTF-16

Символ (U+263A)= 9786 = 00100110 00111010

Код этого символа меньше $FFFF_{16}$ при этом не попадает в диапазон ($D800_{16}$..$DFFF_{16}$) поэтому просто записывается как есть 26 3A (помним о big endian). В начале файла нужно записать FE FF.

Fortran

Реализация алгоритма на Fortran

Функции для кодирования в UTF-16

```
function number_of_words(code_point)
        integer(4),intent(in)::code_point
        integer(4)::number_of_words
        number_of_words = 1
        if(code_point > Z'10000') then
            number_of_words = 2
        end if
    end function number_of_words

    function code_to_UTF16(code_points_array)
        integer(4),intent(in),dimension(:),allocatable::code_points_array
        integer(2),dimension(:),allocatable::code_to_UTF16
        integer(4)::i,j,n_words,temp_code_point
        n_words = 0
        do i = 1,size(code_points_array)
            n_words=n_words + number_of_words(code_points_array(i))
        end do
        allocate(code_to_UTF16(n_words))
        j=1
        do i = 1,size(code_points_array)
            n_words = number_of_words(code_points_array(i))
            if (n_words == 1) then
                code_to_UTF16(j) = int(code_points_array(i),kind=2)
                j=j+1
            else
                temp_code_point = code_points_array(i) - int(Z'10000')
                code_to_UTF16(j) = int(Z'D800' + ISHFT(temp_code_point,-10),kind=2)
                j=j+1
                code_to_UTF16(j) = int(Z'DC00' + IAND(temp_code_point,Z'3FF'),kind=2)
                j=j+1
            end if
        end do
    end function code_to_UTF16
```

Функции для кодирования строки в UTF-16

```
function code_text_to_UTF16(text_line)
        integer, parameter::ucs4  = selected_char_kind ('ISO_10646')

        character(len=*,kind=ucs4),intent(in)::text_line
        integer(2),dimension(:),allocatable::code_text_to_UTF16
        integer(4),dimension(:),allocatable:: code_points_array
        integer(4)::i, text_size
        text_size = len_trim(text_line,kind=ucs4)
        allocate (code_points_array(text_size))
        do i = 1,size(code_points_array)
            code_points_array(i) = ICHAR(text_line(i:i),kind=ucs4)
        end do
        code_text_to_UTF16 = code_to_UTF16(code_points_array)
        deallocate(code_points_array)
    end function code_text_to_UTF16
```

Функции для декодирования из UTF-16

```
function number_of_characters(hex_word_array)
        integer(2),dimension(:),allocatable,intent(in)::hex_word_array
        integer(4)::number_of_characters
        integer(4)::i,temp_code
        number_of_characters = 0
        do i = 1, size(hex_word_array)
            temp_code = IAND(int(hex_word_array(i)),Z'FFFF')
            if(temp_code>=Z'D800' .and. temp_code<=Z'DFFF') then
                number_of_characters=number_of_characters + 1
            end if
        end do
        number_of_characters = size(hex_word_array) - number_of_characters/2
    end function number_of_characters
```

Функции для декодирования из UTF-16

```
function decode_from_UTF16(hex_word_array)
        integer(2),dimension(:),allocatable::hex_word_array
        integer(4)::n
        integer(4)::i,j,temp_code
        integer(4),dimension(:),allocatable::decode_from_UTF16
        n = number_of_characters(hex_word_array)
        allocate(decode_from_UTF16(n))
        i = 1
        j = 1
        do while(i <= size(hex_word_array))
            temp_code = IAND(int(hex_word_array(i)),Z'FFFF')
            if(temp_code>=Z'D800' .and. temp_code<=Z'DFFF') then
                decode_from_UTF16(j) = ISHFT(IAND(temp_code,Z'3FF'),10)
                i=i+1
                temp_code = IAND(int(hex_word_array(i)),Z'FFFF')
                decode_from_UTF16(j) = IOR(decode_from_UTF16(j),IAND(temp_code,Z'3FF'))+Z'10000'
                i=i+1
                j=j+1
            else
                decode_from_UTF16(j) = temp_code
                i=i+1
                j=j+1
            end if
        end do
    end function decode_from_UTF16
```

Функции для декодирования строки из UTF-16

```
function decode_text_from_UTF16(hex_word_array)
        integer, parameter::ucs4  = selected_char_kind ('ISO_10646')
        integer(2),dimension(:),allocatable,intent(in)::hex_word_array

        character(kind=ucs4,len=:),allocatable::decode_text_from_UTF16
        integer(4),dimension(:),allocatable::code_points_array
        integer(4)::i,text_size
        code_points_array = decode_from_UTF16(hex_word_array)
        text_size = size(code_points_array)
        allocate(character(len=text_size,kind=ucs4)::decode_text_from_UTF16)
        do i = 1,text_size
            decode_text_from_UTF16(i:i) = CHAR(code_points_array(i),kind=ucs4)
        end do
    end function decode_text_from_UTF16
```

UTF-32

UTF-32 (англ. Unicode Transformation Format) - один из способов кодирования символов Юникода, использующий для кодирования любого символа ровно 32 бита.

Символ UTF-32 является прямым представлением его кодовой позиции. Главное преимущество UTF-32 перед кодировками переменной длины заключается в том, что символы Юникод непосредственно индексируем. Получение n-ой кодовой позиции является операцией, занимающей одинаковое время. Напротив, коды с переменной длиной требует последовательного доступа к n-ой кодовой позиции. Это делает замену символов в строках UTF-32 простой, для этого используется целое число в качестве индекса, как обычно делается для строк ASCII.

Пример кодирование в UTF-32

Символ (U+263A)= 9786 = 00100110 00111010

Для записи в формате UTF-32 нужно просто записать его номер в виде 32 битного числа.

00000000 00000000 00100110 00111010 = 00 00 26 3A

Если используется little endian то в файл нужно записать FF FE 00 00 3A 26 00 00

Список литературы

1) ASCII стандарт
2) UTF-8 стандарт
3) UTF-16 стандарт
4) Unicode