V

23 марта 2020

Символы и кодировка: историческая справка I

В третьей лекции был введён символьный тип **char** и понятие *кодировки* в языках программирования. В качестве базовой кодировки была приведена ASCII. Первоначально она включала в себя только 128 символов, затем была расширена до 256. Одним из основных достоинств этой кодировки на заре развития ЭВМ (60-е — 70-е года XX века) являлось компактное хранение текстовой информации: на один символ тратился только один байт. Что было важным, в том числе и для быстродействия обработки текстов (меньше байт загружать => быстрее обработка), в годы жестких ограничений на объём памяти компьютеров и различных съёмных носителей.
Поскольку ASCII включала только англоязычный алфавит, в других странах были разработаны собственные кодировки. Как правило, общая схема *региональных* кодировок была следующей: брался 1 байт без знака (ака 8 бит на ведущих

архитектурах ЭВМ), первые 128 символов сохраняли совместимость с ASCII-таблицей, в оставшиеся (коды от 128 до 255) добавлялись локальные языковые символы. Соответственно, сохранялась возможность пользоваться как английским, так и региональным алфавитом. Первое следствие из такой схемы, что региональные коды символов никак не могли быть совместимы между собой.
В 1986 годы в Академии Наук СССР группа исследователей разработала кодировку CP866 (Code page 866, также известная как «DOS Cyrillic Russian») для представления символов из кириллицы. Кроме русского языка, эта кодировка позволяла хранить и тексты в некоторых других языках, основанных на кириллице (для примера, болгарский язык). CP866 стала основной кодировкой для руссифицированной версии операционной системы DOS. И этот факт до сих пор приносит некоторые неудобства в изучении языка C++ на ОС

Символы и кодировка: историческая справка III

Windows. Сама по себе кодировка отличалась алфавитным порядком следования русских букв.
Затем, при выходе ОС Windows (середина 90-х) компанией Microsot была разработана собственная кодировка для кириллицы — **windows-1251** или «cp1251». Основнная на том же принципе (хотим весь региональный алфавит в одном байте) она включала более полный набор кириллических символов в сравнении с CP866. Помимо русского языка позволяла реализовывать сербский, украинский, белорусский и македонский алфавиты. Эта кодировка стала стандартной для каждой руссифицированной версии ОС Windows. При этом коды русских букв не совпадают с CP866.
И первое столкновение двух последних кодировок происходит, когда на руссскоязычной ОС Windows реализуются **консольные** программы (ещё раз определение: программы, взаимодейтсвующие с пользователем в текстомом режиме).

Символы и кодировка: историческая справка IV

Так, все текстовые файлы в подобной региональной версии ОС Windows создаются в кодировке windows-1251. Но при этом, по умолчанию *командная строка* (она же — «command line», «cmd.exe») интерпретирует все переданные ей символы (их коды) согласно кодировке CP866. Именно это является причиной того, что без дополнительных манипуляций (ссылка) вместо русских букв в командной строке Windows показываются совсем неожидаемые символы.
Кроме того, на реализацию однобайтовыйх кодировок накладываются особенности того, как тип **char** представлен в ОС. На настоящий момент, в OS Windows на архитектурах **x86** и **x86-64** тип **char** является *знаковым* и позволяет хранить в переменных значения от -128 до 127 включительно. Поэтому, при просмотре кода русских символов в данной ОС, будут показываться отрицательные значения. При этом, сами

Символы и кодировка: историческая справка V

таблицы кодировок предполагают, что каждому символу сопоставлено только положительное значение.
Однобайтовое представление символов не было идеальным. В частности, оно не всегда позволяло реализовать полный алфавит некоторых восточных стран. Кроме того, с развитием мультиязычного программного обеспечения и интернета, возникла необходимость хранения достаточно произвольных наборов текста, где могли бы встречаться комбинации различных алфавитов. Для примера, любой мессенджер на смартфонах позволяет вам распостранять текст практически на любом алфавите. Такое было бы невозможно, продолжай все использовать региональные кодировки.
Для решения такой проблемы (хотим хранить текст в заранее неизвестном алфавите) были разработаны **многобайтовые** кодировки. Скажем, есть кодировки **UTF16** и **UTF32**, которые используют для хранения символов 2 и 4 байта,

Символы и кодировка: историческая справка VI

соответственно. Однако наибольшее распространение в современных языках программирования получила кодировка **UTF8**.

C++ и UTF8

Язык C++ позволяет хранить, передавать и показывать любой текст в кодировке **UTF8**, но его стандартная библиотека не предоставляет функций для манипулирования отдельными символами. Для этого стоит использовать отдельные библиотеки.

Символы и кодировка

Что за **UTF8**?

- текстовый символ может состоять от 1 до 4 байт
- на представление уникального символа выделено не более 21 бита
- вводится понятие «code point», равное 1 байту (8 битам)
- включает в себя все возможные региональные алфавиты
- Коды от 0 до 127 совпадают с кодировкой ASCII
- Больше информации: https://en.wikipedia.org/wiki/UTF-8

Вне ОС Windows

На отличных от Windows операционных системах (Mac OS, Linux-based OS) кодировка **UTF8** является кодировкой по умолчанию для любых текстовых документов. Если будете работать в них, приведённая ин-фа будет полезна, как и знание, что любая русская буква в UTF8 занимает 2 байта.

Работа с текстом

Базовая строка в C++ - это одномерный массив значений типа **char**, в котором присутствует специальный символ (значение типа **char**) — '\0'. он же известен как *символ конца строки* или *нулевой символ*. Его целочисленный код — 0.
С помощью символа конца строки вычисляют её длину — количество ненулевых значений типа **char** **до первого** нулевого символа. Длина строки не обязана совпадать с *длиной массива*, используемого для её хранения.
Литералом строкового типа является группа текста, заключённая в двойные кавычки. Напомним, что литералом является некоторое неизменяемое значение определённого типа данных. Строки неоднократно использовались в примерах при вызовах функций **printf/scanf**.

Работа с текстом

Исходя из определения строк, для хранения и манипуляции с ними в C++ используются либо статические массивы типа **char**), либо указатели на динамические массивы. Рассмотрим основные способы задания строк в программах.

```
1 char phrase1[] = "String example!";
2 printf("Saved string: %s\n", phrase1);
```

По строкам примера:

1. создаём статический массив и сохраняем в нём указанную строку. Так как массив инициализируется сразу, размер можно не указывать. При этом, длина сохранённой строки равна 15 символам, а размер массива — 16 элементов. Нулевой символ при задании строки через двойные кавычки подставляется неявно;
2. для вывода строки на консоль можно воспользоваться функцией **printf** со спецификатором «**%s**». Из массива **pharase1** будут выводиться все символы (в смысле типа **char**) до тех пор, пока не встретится нулевой символ.

Работа с текстом

Для демонстрации связи строк и массива рассмотрим следующий пример.

```
1 char phrase2[] = { 'A', 'B', 'C', '\0' };
2 // Тоже самое, что и:
3 // char phrase2[] = "ABC";
4 printf("Second symbol is {{%c}}\n", phrase2[1]);
```

По строкам примера:

- в 1-ой строке явно задаём массив типа **char** на четыре элемента и *нулевой символ* ставим явно руками. Таким образом, получается строка длинною в 3 символа. К слову, учитывая определение типа **char**, получается что длина строки фактически определяется как количество байт до нулевого символа, а не как количество текстовых символов (замечание относится к ОС с кодировкой **UTF8**);
- в 4-ой строке просто берём второй символ массива **pharase2** и печатаем его на консоли.

Работа с текстом

Повторимся, что любой текст в двойных кавычках представляет собой литерал строкового типа данных. Сами по себе литералы, к какому бы типу они не относились, представляют собой *неизменяемые* значения. Исходя из этого, C++ позволяет сохранять строки в **указателях на константную переменную** (см. четвёртая лекция, 24-ый слайд). Синтаксис следующий:

```
1 const char *phrase3 = "To be or not to be";
2 printf("%s\n", phrase3);
3 puts(phrase3);
```

- строка в двойных кавычках имеет время жизни, аналогичное локальным или глобальным переменным (в зависимости от контекста). Доступ к строке осуществляется через указатель **phrase3**;
- строки 2 и 3 примера делают одно и тоже — печатают строку в консоли и переходят в ней на следующую строчку.

Работа с текстом I

Пример про различие строки и массива элементов типа **char** в C++ (помним, что по умолчанию примеры демонстрируют работу с локальными переменными):

```
1 const size_t SZ = 15;
2 char symb_array[SZ];
3 symb_array[2] = 'w';
4 printf("symb_array: %s\n", symb_array);
5
6 char right_str[SZ] = { '\0' };
7 printf("right_str: %s\n", right_str);
```

По строкам:

2. создали массив на 15 элементов. Ничем не инициализировали => непонятно, что за значения появились в его элементах;
3. в третий элемент записали значение 'w'

Работа с текстом II

4. данный вызов печати строки на консоль — **очень опасен**. Функция будет искать в массиве **symb_array** символ конца строки, но его там может и не быть. В этом случае произойдёт выход за границу созданного массива со всеми неприятными последствиями;
6. здесь же объявлена правильная строка, которая является пустой — не содержит ни одного текстового символа. То есть при длине массивы в 15 элементов, длина строки равна нулю. Корректность строки обеспечивает *инициализация* массива начальными значениями. В данном случае, во все элементы массива гарантированно будет записан код **0**.

Работа с текстом

Строки можно также хранить с помощью динамических массивов. Для примера,

```
1  const size_t SZ = 16;
2  // Создаём динамический массив
3  char *str = new char[SZ];
4  // заполняем первые восемь элементов
5  // первыми восьмью буквами англ. алфавита
6  for (size_t i = 0; i < (SZ / 2); i++) {
7    str[i] = 'a' + i;
8  }
9  // не забываем установить символ конца строки
10 str[SZ/2] = '\0';
11
12 // Всё ок, можно безопасно печатать на консоль:
13 puts(str);
14
15 // не забываем вернуть динамическую память в ОС
16 delete[] str;
```

Работа с текстом

Предыдущий пример можено сделать удобнее, если использовать инициализацию при создании динамического массива

```
1  const size_t SZ = 16;
2  // Создаём динамический массив
3  char *str = new char[SZ] { '\0' };
4  // заполняем первые восемь элементов
5  // первыми восьмью буквами англ. алфавита
6  for (size_t i = 0; i < (SZ / 2); i++) {
7    str[i] = 'a' + i;
8  }
9  // не нужно явно сохранять нулевой символ, так
10 // как он уже присвоен всем элементам, кроме тех,
11 // что были изменены выше.
12 puts(str); // безопасный вызов
13
14 // не забываем вернуть динамическую память в ОС
15 delete[] str;
```

Работа с текстом

Сохранение символьных строк через статические или динамические массивы накладывают теже ограничения на удобство их использования.

Какие проблемы?

- нет простого копирования строк — к переменным-массивам не применима операция присваивания (за исключением инициализации);
- необходимо помнить про символ окончания строки \0 при переборе всех элементов массива;
- если строка сохранена в статическом массиве — его размер не изменить во время выполнения;
- при операциях со строками необходимо очень внимательно следить за границей используемых статических или динамических массивов.

Конечно, часть проблем решается с использованием функций из стандартной библиотеки языка C++. Нужно учесть, что большинство этих функций работают указателями на **char**,

Работа с текстом: длина строки

Прежде, чем переходить к набору функций стандартной библиотеки, рассмотрим собственные реализации функций для упрощения работы со строками. Одна из основных задач — определение длины строки. Она может быть решена с помощью следующей функции:

```
1 size_t str_length(char *str)
2 {
3   size_t counter = 0;
4   while (str[counter] != '\0') {
5     counter++;
6   }
7   return counter;
8 }
```

Подход аналогичен стандартной библиотеке: проходим по всем символам переданного массива (совершенно не важно, статического или динамического) и ищем *нулевой символ*.

Работа с текстом: длина строки

Использование функции:

```
1 char str[80] = "Not so long string";
2 printf("string <<%s>> has length %ul\n",
3        str, str_length(str)); // Покажет 18
4
5 char other[] = "other example";
6 size_t other_len = str_length(other);
7 printf("length of other is %ul\n",
8        other_len); // 13
```

Сам функция по вычислению длины строки ничего не знает про размер переданного ей массива. Правильность работы с ней относится к ответственности того, кто использует эту функцию: надо гарантировать, чтобы в переданном массиве хоть раз встретился символ конца строки. Иначе будут проблемы с работой программы: или ошибка доступа к памяти, или вычисление некорректной длины (и затем уже ошибка доступа к памяти).

Работа с текстом: объединение строк

Другой пример — добавление одной строки к другой. Объявление функии будет следующим:

```
1 void concatenate(char *dest, const char *source);
```

Под **dest** (от англ. destination — место назначения, приёмник) пониматеся строка, в которую будет осуществлено добавление. Под **source** — строка, из которой будут взяты все символы для добавления. Опять же, предполагается, что при вызове данной функции размер массива под первую строку будет достаточен для добавления всех символов второй строки.

Работа с текстом: объединение строк I

Теперь определим функцию

```
1  void concatenate(char *dest, const char *source)
2  {
3    size_t dest_end = str_length(dest);
4
5    size_t source_end = 0;
6    while (source[source_end] != '\0') {
7      dest[dest_end] = source[source_end];
8      dest_end++; source_end++;
9    }
10
11   dest[dest_end] = '\0';
12 }
```

Работа с текстом: объединение строк II

- в строке **3** с помощью вспомогательной функции находим количество символов в строке **dest**. Это количество совпадает с индексом элемента, в котором находится нулевой символ;
- в строке **5** заводим счётчик для прохода по всем символам строки **source** до тех пор, пока не будет найден нулевой символ;
- в строках **6 — 9** с помощью цикла **while** **поэлементно** копируем символы строки **source** в строку **dest**. Обратите внимание, первым замещается символ нулевой строки из **dest**;
- в строке **11** помещаем символ конца строки на новое место в **dest**.

Работа с текстом: объединение строк

Использование функции **concatenate**

```
1 const size_t SZ = 80;
2 char target[SZ] = "A little step for the human";
3 char addition[] = "a biggest step for the ↩
    humanity";
4
5 concatenate(target, " - ");
6 concatenate(target, addition);
7
8 printf("final string is {%s}\n", target);
```

Размер массива **target** задан с запасом для добавления двух строк (строчки 5,6).

Работа с текстом: ввод с консоли

```
1 char word[20];
2
3 printf("Enter the word: ");
4 scanf("%20s", word);
```

Три ключевые особенности ввода:

1. "**%20s**"у **scanf** означает ввод строки (до первого пробела или переноса), с ограничением максимальной длины в 19 символов (байт). Хотя размер введённой строки может быть и меньше
2. Символ окончания строки добавляется функцией **scanf** по умолчанию (если длина группы символов более 20 штук, то в **word** попадёт только 19 из них и двадцатым будет поставлен символ окончания строки)
3. Поскольку переменная-массив **word** фактически хранит адрес первого элемента, знак **&** в **scanf** указывать не нужно

Работа с текстом: ввод с консоли

Безопасный ввод строки текста

```
fgets(char *str, size_t max_chars,
      stream_pointer)
```

1. **str** - массив типа **char**, куда будут сохранена вводимая строка
2. **max_chars** - целое число, обозначающее **максимальное число символов**, которые будут сохранены в **str**
3. **stream_pointer** - указатель на структуру, ассоциированную с вводом текста. Для терминала это всегда переменная **stdin**, которая определена в **<cstdio>**

```
1 const size_t SZ = 140;
2 ...
3 char phrase[SZ];
4
5 printf("Enter a phrase: ");
6 fgets(phrase, 140, stdin);
```

Работа с текстом в C++ с помощью стандартной библиотеки

продолжение будет ко вторнику, 23 марта 2020