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 Microsoft 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 текстовом 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 **phrase1** 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 // То же 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 длиною 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 **phrase2** 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 массива 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 можно 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 те же 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 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 массивов.

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

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 %zu\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 %zu\n",
8        other_len); // 13
```

Сама 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) понимается 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 I

Первый способ ввести группу символов с консоли и сохранить её в строке — функция **scanf**.

```
1 char word[21];
2
3 printf("Enter the word: ");
4 scanf("%20s", word);
```

Ключевые особенности ввода:

1. "**%20s**" у **scanf** означает ввод строки (до первого пробела или переноса), с ограничением максимальной длины в 20 символов (байт). Хотя размер введённой строки может быть и меньше;
2. число «20» в данном примере является *шириной поля ввода данных*;

Работа с текстом: ввод с консоли II

3. символ окончания строки добавляется функцией **scanf** по умолчанию (если длина группы символов более 20 штук, то в **word** попадёт только 20 из них и двадцать первым будет поставлен символ окончания строки);
4. поскольку переменная-массив **word** фактически хранит адрес первого элемента, знак амперсанда **&** в **scanf** не нужен;
5. при задании ширины поля массив под текстовый фрагмент должен иметь размер: **ширина поля ввода** + 1 символ;
6. если введённое количество символов меньше ширины поля ввода, то в переменную **word** попадут все символы **до первого пробела или переноса строки**;

Работа с текстом: ввод с консоли III

7. символ перевода строки ('\n', добавляется нажатием клавиши Enter) в переменную не попадает (остаётся для дальнейшего ввода).

Зачем нужна ширина ввода?

Можно использовать просто спецификатор "**%s**". Но в этом случае функция **scanf** будет пытаться записать **все** символы до первого пробела в переменную. А далеко не факт, что размер массива будет достаточен для записи всех символов => запись в область памяти за границей массива => падение программы или повреждение других переменных программы.

Работа с текстом: ввод с консоли I

В тоже время **scanf** позволяет получить и всю строку до первого переноса.

```
1 char sentence[80];
2
3 printf("Enter the sentence: ");
4 scanf("%79[^\n]", sentence);
```

Что здесь происходит:

1. число **79** ограничивает максимально возможное количество текстовых символов, которые будут сохранены в **sentence**;
2. квадратные скобки внутри форматной строки означают группу символов, которые будут выбраны для записи в переменную **sentence**;

Работа с текстом: ввод с консоли II

3. символ «^» означает, что из консольного ввода для записи в строку будут считаны все символы до тех пор, пока не встретится один из группы, перечисленной после «крышки»
4. после символа «^» стоит '\n' — следовательно, как только встретится первый перенос строки (при условии, что перед ним меньше 79 символов), отбор символов для сохранения в переменную закончен.

Таким образом, данная форма позволяет вводить уже предложения — группу слов, разделённых пробелами.

Работа с текстом: ввод с консоли

Для примера, так можно получить строку до первой точки:

```
1 char sentence[80];
2
3 printf("Enter the sentence: ");
4 scanf("%79[^.]", sentence);
5
6 printf("You entered:\n<<%s>>\n", sentence);
```

Рекомендуется исследовать данный фрагмент: выполните код, попробуйте вводить предложения с переносом строк или без; с вводом точки или вводом больше 79 символов. И проверьте результат.

Работа с текстом: ввод с консоли

Другой способ ввести строку — воспользоваться функцией **fgets**

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

1. **str** - массив типа **char**, куда будет сохранена вводимая строка;
2. **max_chars** - целое число, обозначающее **максимальное число символов**, которые будут сохранены в **str** (включая и символ конца строки);
3. **stream_pointer** - указатель на структуру, ассоциированную с вводом текста. Для консоли это всегда переменная **stdin**, которая определена в <**cstdio**>;

Работа с текстом: ввод с консоли

Пример использования:

```
1 const size_t SZ = 140;
2
3 char phrase[SZ];
4
5 printf("Enter a phrase: ");
6 fgets(phrase, SZ, stdin);
7 printf("<<%s>>\n", phrase);
```

Неприятная особенность fgets

Если количество символов меньше максимального (139 в примере выше), то в переменную будет помещён и символ переноса строки '\n'.

Работа с текстом: ввод с консоли

Какую функцию предпочесть для ввода текста — зависит от задачи. Как правило, присутствие символа переноса строки в сохранённой группе символов не нужно. С **fgets** возможное исключение '\n' можно делать так:

```
1  // хотим получить 140 символов
2  const size_t SZ = 140;
3  // + 1 => под нулевой символ
4  char phrase[SZ + 1];
5
6  printf("Enter a phrase: ");
7  fgets(phrase, SZ + 1, stdin);
8
9  size_t len = str_length(phrase);
10 if (len < SZ) {
11   phrase[len - 1] = '\0';
12 }
13
14 printf("<<%s>>\n", phrase);
```

Работа с текстом: ввод с консоли I

Ввод всей строки способом с 26-го слайда на первый взгляд выигрывает — не нужны никакие проверки на длину получившейся строки и ручное удаление символа переноса. Однако, есть другая особенность:

```
1  const size_t SZ = 140;
2  char sentence[SZ + 1], phrase[SZ + 1];
3
4  printf("Enter a sentence: ");
5  scanf("%140[^\n]", sentence);
6  printf("Enter a phrase: ");
7  scanf("%140[^\n]", phrase);
8
9  printf("<<%s>>\n", sentence);
10 printf("{{%s}}\n", phrase);
```

Работа с текстом: ввод с консоли II

Если для первой строки будет введено меньше 139 символов, то в переменную **phrase** не будет введено ни одного символа. Это обусловлено тем, что символ переноса '\n' будет учтён во время второго вызова функции **scanf**.

Для исправления ситуации можно воспользоваться циклом **while** и функцией **getchar**

```
1  const size_t SZ = 140;
2  char sentence[SZ + 1], phrase[SZ + 1];
3
4  printf("Enter a sentence: ");
5  scanf("%140[^\n]", sentence);
6  while (getchar() != '\n');
7
8  printf("Enter a phrase: ");
9  scanf("%140[^\n]", phrase);
10 while (getchar() != '\n');
11
```

Работа с текстом: ввод с консоли III

```
12 printf("<<%s>>\n", sentence);
13 printf("{{%s}}\n", phrase);
```

Функция **getchar** позволяет получить следующий символ с консольного ввода. Циклы в 6-ой и 10-ой строках не имеют тела. Их работа заключается в том, чтобы убрать все символы из операции ввода перед следующим вызовом функции **scanf**.

Работа с текстом: ввод с консоли I

В предыдущих примерах должно напрягать число **140** внутри **scanf**. В программах все стараются избегать таких встроенных значений. Кроме того, если буфер под строку задан константой или наоборот, является динамическим — хочется иметь универсальный способ безопасного ввода строк. Для этого можно воспользоваться функцией **sprintf** из заголовочного файла <**cstdio**>. Она позволяет форматировать строку аналогично обычной **printf**, но результат записать в строку, переданную первым аргументом, а не вывести что-то на консоль.

Работа с текстом: ввод с консоли II

```
1 const size_t SZ = 20;
2 char buf[SZ] = { '\0' };
3
4 sprintf(buf, "real: %.3f", 5.89345);
5
6 printf("The result: <<%s>>\n", buf);
7 // Печатает: The result: <<real: 5.894>>
```

Задали массив под строку, воспользовались функцией **sprintf** — отформатировали число с плавающей точкой, вывели результат на экран.

Тогда пример с 34-35 слайдов можно реализовать так:

Работа с текстом: ввод с консоли III

```
1  char format_str[20] = { '\0' };
2  const size_t SZ = 140;
3  char sentence[SZ + 1], phrase[SZ + 1];
4
5  sprintf(format_str, "\%%zu[^\n]", SZ);
6
7  printf("Enter a sentence: ");
8  scanf(format_str, sentence);
9  while (getchar() != '\n');
10
11 printf("Enter a phrase: ");
12 scanf(format_str, phrase);
13 while (getchar() != '\n');
14
15 printf("<<%s>>\n", sentence);
16 printf("{{%s}}\n", phrase);
```

Работа с текстом: ввод с консоли IV

В 5-ой строке создаётся *форматная строка*, которая далее будет использоваться в вызовах **scanf**. Таким способом можно динамически создавать нужные форматные строки, например для последовательно ввода строк разной длины.

Работа с текстом в C++ с помощью стандартной библиотеки <cstring>

Далее используется библиотека из языка C, реализованная и в C++ для работы с базовыми строками:

```
1 #include <cstring>
```

Функции, предоставляемые данной библиотекой, широко используют указатели для различных строковых манипуляций.

Получение длины строки **в байтах** (аналог функции с 18-го слайда).

`size_t strlen(const char* str)`

```
1  char text[] = "A string";
2  // На экране покажется число 8
3  printf("Text length: %zu", strlen(text));
4
5  char text2[] = "Строка на русском";
6  /* А здесь — зависит от ОС и её локальных
7     настроек. В русскоязычных Windows 7, 8,
8     10 — скорее всего покажет длину равную 17.
9     В ОС, основанных на Linux, или Mac OS
10    наиболее вероятное значение — 32 */
11 printf("String length: %zu", strlen(text2));
```

Посимвольное сравнение строк

```
int strcmp(const char *first,
           const char *second)
```

Функция возвращает:

1. значение $<$ **0** - первый несовпадающий символ в строке **first** имеет код меньше, чем в строке **second**;
2. значение $==$ **0** - все символы в обоих строках совпадают;
3. значение $>$ **0** - первый несовпадающий символ в строке **first** имеет код больше, чем в строке **second**;

Сравнение идёт с начала строк и до тех пор, пока в одной из них не встретится символ окончания строки. Фактически, сравниваются целочисленные коды каждого символа (байта).

Сравнение строк

```
1 char str1[] = "Unite",
2      str2[] = "unite",
3      str3[] = "Unite";
4
5 printf("Compare str1 and str2: ")
6 printf("%d", strcmp(str1, str2));
7
8 printf("Compare str1 and str3: ");
9 printf("%d", strcmp(str1, str3));
```

Сравнение строк

```
1  const size_t SZ = 50;
2  char key_color[] = "orange";
3  char answer[SZ + 1], format[20] = {'\0'};
4
5  sprintf(format, "%%%zus", SZ);
6
7  do {
8    printf("\nGuest color (lower case): ");
9    scanf(format, answer);
10 } while ( strcmp(key_color, answer) != 0 );
11
12 puts("Right answer!");
```

Возможный вывод программы:

```
Guest color (lower case): red
Guest color (lower case): green
Guest color (lower case): orange
Right answer!
```

Посимвольное сравнение начальных фрагментов строк

```
int strncmp(const char *first,
            const char *second,
            size_t count)
```

Сравнение идёт с начала и до тех пор, пока в одной из строк не встретится символ окончания строки, **или** не закончится сравнение **count** символов каждой строки друг с другом.

Посимвольное сравнение начальных фрагментов строк

```
1  char str1[] = "12 reasons to be better",
2       str2[] = "12 explanations of failures",
3
4  if ( strncmp(str1, str2, 2) == 0 ) {
5    puts("Both strings starts with \"12\".");
6  }
7
8  // Как переносимым образом сравнивать
9  // фрагменты с текстом, отличным от ASCII
10 if ( strncmp(str1, str2,
11                 strlen("12 reasons")) != 0 ) {
12   puts("But not with \"12 reasons\".");
13 }
```

Копирование строки

```
char* strcpy(char *dest, const char *source)
```

- Функция копирует строку **source** в строку **dest**
- Функция возвращает указатель на строку **dest**. Как правило, возвращаемое значение игнорируется
- Копируются **все** символы из строки **source**, включая и завершающий *символ окончания строки*
- Следить за размером строк следует самостоятельно!

```
1 char source[] = "Demo of strcpy func";
2 char dest[70];
3 strcpy(dest, source);
4
5 printf("String in dest: <<%s>>\n", dest);
6 printf("It's length: %zu", strlen(dest));
```

Копирование фрагмента строки

```
char* strncpy(char        *dest,
              const char  *source,
              size_t      count)
```

- **count** - целое безнаковое число, количество символов которое копируется в **dest**;
- Если длина копируемой строки меньше, чем указано в аргументе **count**, то оставшиеся символы (байты) заполняются нулями (\0);
- Если длина копируемой строки больше, чем указано в аргументе **count**, то в строку **dest** не добавляется символ окончания '\0'.

Копирование фрагмента строки

```
1 char source[] = "Just a simple string";
2 const size_t LEN = strlen("Just a simp");
3
4 char *dest = new char[LEN + 1] {'\0'};
5 strncpy(dest, source, LEN);
6
7 printf("Fragment in dest: <<%s>>", dest);
8
9 delete[] dest;
```

Добавление одной строки в другую (склеивание строк)

```
char* strcat(char *dest, const char *source)
```

- Функция добавляет строку **source** в конец строки **dest**
- Функция возвращает указатель на строку **dest**. Как правило, возвращаемое значение игнорируется
- Завершающий *символ окончания строки* переносится в конец объединённой строки
- Следить за размером строк - самостоятельно

Добавление одной строки в другую

```
1 char message[140];
2
3 strcat(message, "These strings ");
4 strcat(message, "were combined ");
5 strcat(message, "by usage four calls ");
6 strcat(message, "of \"strcat\" function.");
7
8 puts(message);
```

Добавление начального фрагмента одной строки в другую

```
char* strncat(char       *dest,
              const char *source,
              size_t     count)
```

- Функция добавляет **count** символов из строки **source** в конец строки **dest**
- Функция возвращает указатель на строку **dest**. Как правило, возвращаемое значение игнорируется
- Завершающий *символ окончания строки* переносится в конец объединённой строки
- Если длина строки **source** меньше, чем аргумент **count**, то добавляются только символы до '\0'

Добавление начального фрагмента одной строки в другую

```
1  char part1[] = "Here have to be ",
2       part2[] = "a message was lost";
3
4  char str[50] = { '\0' };
5
6  strcat(str, part1);
7  strncat(str, part2, strlen("a message"));
8
9
10 printf("Final string: \"%s\"", str);
```

Поиск первого вхождения символа в строке

```
char* strchr(const char *str, int character);
```

- Функция ищет символ **character** в строке **str**
- Функция возвращает указатель на **первый** найденный символ
- Если совпадения не найдено, возвращается нулевой указатель **nullptr**
- Несмотря на то, что второй аргумент передаётся как значение типа **int**, фактически внутри функции оно приводится к типу **char**
- Из предыдущего пункта следует *правило хорошего тона*: **strchr** должна использоваться **только** для символов из кодировки ASCII

Поиск первого вхождения символа в строке

```
1  char text[] = "That's me in the corner\n"
2                "That's me in the spotlight\n"
3                "Losing my religion\n"
4                "Trying to keep up with you\n"
5                "And I don't know if I can do it\n"
6                "Oh no, I've said too much\n"
7                "I haven't said enough\n";
8  printf("Find all apostrophes in\n%s\n", text);
9  puts("----------------------------------");
10 const char apostr = '\'';
11
12 char *symb_ptr = strchr(text, apostr);
13 while (symb_ptr != nullptr) {
14   int place = p_space - text + 1;
15   printf("%c found at %d place\n", apostr, place);
16   // Поиск продолжается с первого символа,
17   // идущего после апострофа
18   symb_ptr = strchr(symb_ptr + 1, apostr);
19 }
```

Поиск подстроки (фрагмента)

```
char* strstr(const char *where,
             const char *what);
```

- **where** - строка, внутри которой ищется фрагмент **what**
- Если совпадение найдено, то есть фрагмент **what** присутствует в **where**, возвращается указатель на его первый символ. Указатель содержит адрес элемента из **исходной строки**
- Если совпадения не найдено (фрагмент **what** не присутствует в строке **where**), возвращается **NULL**

Пример: замена слова "What's"на 6 символов решётки (#)

```
1  char text[] = "What's to do? What's to study?"
2                " What's happen?";
3  const char fragm[] = "What's";
4  const size_t SZ = strlen(fragm);
5
6  char *sharps = new char[SZ + 1];
7  for (size_t i = 0; i < SZ; ++i) {
8    sharps[i] = '#';
9  }
10 sharps[SZ] = '\0';
11
12 printf("Before transform: \n%s\n\n", text);
13 char *p_str = strstr(text, fragm);
14 while (p_str != NULL) {
15   strncpy(p_str, sharps, SZ);
16   p_str = strstr(p_str + SZ + 1, fragm);
17 }
18 printf("After transform: \n%s\n\n", text);
19 delete[] sharps;
```

Работа с текстом. Стандартная библиотека

На самостоятельное изучение предлагается разобраться с файлом <**cctype**>.
В нём предоставляются функции для обработки отдельных символов (значений типа **char**).
Проверки: **isalpha**, **isdigit**, **isspace** и другие.
Преобразования: **tolower**, **toupper**.
Применимы только для символов из кодировки ASCII.
ссылка на одну из справок

Работа с текстом. Ввод строки произвольной длины

Среди предыдущих слайдов было несколько способов получения строк (как групп символов), но все они характеризовались тем, что было ограничено их максимальное число для сохранения в переменной. В то же время, при работе с текстом возникают задачи по прочтению строк заранее неизвестной длины. Далее разберём пример, как это возможно реализовать теми средствами, что были изучены к настоящему моменту.

Пример. Ввод строки произвольной длины

Итак, задача: ввести строку произвольной длины с консоли. Под строкой понимаем все символы, до первого переноса. Для её выполнения понадобятся следующие средства C++:

- динамические массивы типа **char** и сопутствующие им операторы **new[]** / **delete[]**;
- функция **getchar** из <**cstdio**>.

Для выделениея/освобождения динамической памяти воспользуемся подходом из четвёртой лекции. Там были приведены функции для работы с памятью: под одномерный массив типа **int**

- **allocate** — 4-ая лекция, слайд 41
- **deallocate** — 4-ая лекция, слайд 41
- **reallocate** — 4-ая лекция, слайд 47

Пример. Ввод строки произвольной длины

Не будем повторять исходный код указанных функций с предыдущего слайда. Для того, чтобы работать с указателями на **char**, достаточно просто в определении функций заменить тип указателей. Это остаётся на самостоятельную реализацию. Таким образом, предполагаем, что уже есть три функции со следующими сигнатурами:

```
1 void allocate(char *& ptr, size_t sz);
2 void deallocate(char *& ptr);
3 void reallocate(char *&ptr, size_t sz, size_t new_sz);
```

Ими воспользуемся для реализации поставленной задачи.

Пример. Ввод строки произвольной длины

Алгоритм ввода строки произвольной длины достаточно прямолинеен:

- задаём некоторый символный массив начального размера;
- считываем один символ с консоли (**getchar**);
- проверяем, не равен ли он символу переноса строки. Если не равен — добавляем в массив, иначе — завершаем чтение символов с консоли;
- проверяем, осталось ли место в массиве под следующий символ. Если не осталось — расширяем динамический массив;
- после окончания ввода, не забываем проставить *символ конца строки* в массив;
- *опционально*, можно сжать динамический массив до актуального размера строки.

Пример. Ввод строки произвольной длины

Нюанс расширения динамического массива

С точки зрения работы операционной системы по предоставлению динамической памяти не слишком хорошо каждый раз расширять массив на один символ. Будем использовать следующую схему: базовый размер массива будет сделан параметром реализуемой функции, а при необходимости расширения — размер массива будет увеличиваться **в два раза**.

Пример. Ввод строки произвольной длины

Теперь готовы представить прототип функции для считывания строки произвольной длины с консоли:

```
1 char* get_line(const size_t base_size,
2                const bool is_fit_to_len = false);
```

Функция принимает два параметра:

1. **base_size** — размер массива под строку, выделяемый перед началом считывания символов с консоли;
2. **is_fit_to_len** — параметр, который определяет, есть ли необходимость привести размер массива к «длине строки + 1 символ конца строки». Значение по умолчанию — **false**, что означает, что длина массива в большинстве случаев будет превышать длину строки.

Функция возвращает указатель на заполненную строку.

Пример. Ввод строки произвольной длины I

Возможная реализация:

```
1  char* get_line(const size_t base_size,
2                 const bool is_fit_to_len = false)
3  {
4    size_t cur_sz = base_size;
5    char *str = nullptr;
6    allocate(str, base_size);
7
8    char next_ch = getchar();
9    size_t cur_pos = 0;
10   while (next_ch != '\n') {
11     if (cur_pos == (cur_sz - 1)) {
12       reallocate(str, cur_sz, cur_sz * 2);
13       cur_sz *= 2;
14     }
15
16     str[cur_pos] = next_ch;
17     cur_pos++;
18
```

Пример. Ввод строки произвольной длины II

```
19      next_ch = getchar();
20    }
21
22    str[cur_pos] = '\0';
23
24    if (is_fit_to_len) {
25      reallocate(str, cur_sz, strlen(str) + 1);
26    }
27
28    return str;
29  }
```

Основные моменты:

- локальная переменная **cur_sz** сохраняет актуальный размер динамического массива **str**;
- локальная переменная **cur_pos** служит индексом позиции элемента, в который будет сохранено очередное значение символа с консоли;

Пример. Ввод строки произвольной длины III

- строки **11 — 14** отвечают за проверку возможности добавления следующего символа в массив без дополнительного выделения памяти. В условии стоит **cur_sz - 1** из-за того, что последний элемент текущего массива всегда зарезервирован под *символ конца строки*;
- строки **24 — 26** проверяют, нужно ли привести размер массива к «длине строки + 1 символ».

Пример. Ввод строки произвольной длины

И использование: ввод 10 строк произвольной длины

```
1  const size_t LINES = 10;
2  const size_t STR_SZ = 40;
3  char *array_of_strs[LINES];
4
5  for (size_t i = 0; i < LINES; i++) {
6    printf("Enter %zu line: ", i + 1);
7    array_of_strs[i] = get_line(STR_SZ);
8  }
9
10 printf("You entered:\n");
11 for (size_t i = 0; i < LINES; i++) {
12   printf("<<%s>> has %zu length\n",
13          array_of_strs[i], strlen(array_of_strs[i]));
14 }
15
16 for (size_t i = 0; i < LINES; i++) {
17   // Не забываем вернуть динамическую память в ОС
18   deallocate(array_of_strs[i]);
19 }
```