Строки

М. А. Ложников

25 марта 2019 г.

Это предварительная невыверенная версия. Читайте на свой страх и риск.

1 Класс std::string

Для работы со строками в C++ используется шаблон `std::basic_string` с единственным обязательным аргументом — типом данных символа. Этот шаблон определён в заголовочном файле `<string>`. Он представляет из себя оболочку над обычным динамическим массивом, то есть хранит символы в памяти непрерывно. Для удобства в `<string>` определены конкретизации шаблона часто использующимися типами символов.

Так класс `std::string` является конкретизацией базового шаблона `std::basic_string` шаблонным параметром `char`. То есть, он совпадает с `std::basic_string<char>`. Класс подходит для работы со строками в однобайтных кодировках, если нужно работать с Unicode, то следует использовать класс `std::wstring` (равный `std::basic_string<wchar_t>`). Далее везде будем рассматривать лишь класс `std::string`.

1.1 Операции, специфичные для контейнера

Класс `std::string` поддерживает все те же операции, что и класс `std::vector<char>`. В самом деле

```cpp
#include <iostream>
#include <string>
#include <algorithm>

int main() {
  std::string str;

  /* Метод push_back() кладёт элементы в конец строки. Ведёт себя также, как и
    в векторе. */
  str.push_back('a');
  str.push_back('p');
  str.push_back('p');
  str.push_back('l');
  str.push_back('e');

  /* Методы size() и length() позволяют узнать длину строки в символах. Методы работают
    одинаково. */
  std::cout << "String '" << str << "' has " << str.size() << " symbols." << std::endl;

  // Метод capacity() позволяет узнать размер выделенной памяти (в символах).
```

```
21   std::cout << "Allocated memory for " << str.capacity() << " symbols." << std::endl;
22
23   // Есть доступ по индексу как на чтение
24   for (std::size_t i = 0; i < str.size(); i++)
25     std::cout << "i = " << i << " sym = " << str[i] << std::endl;
26
27   // так и на запись
28   str[0] = 'A';
29
30   /* У класса std::string есть прямые/обратные константные/неконстантные итераторы.
31     Соответственно к строкам можно применять функции из библиотеки алгоритмов. */
32   // Например, найдём первый символ 'l'.
33   std::string::iterator it = std::find(str.begin(), str.end(), 'l');
34
35   /* Удалим этот символ. Функция вернёт итератор на символ, следующий за удалённым. */
36   it = str.erase(it);
37
38   /* Вставим обратно заглавный символ. Вместо одного символа можно передать
39     полуинтервал, который нужно вставить. Полуинтервал задаётся парой итераторов.
40     Метод возвращает итератор на первый вставленный элемент. */
41   str.insert(it, 'L');
42
43   /* Метод empty() позволяет узнать, пуста ли строка. */
44   std::cout << str.empty() << std::endl;
45
46   /* Метод clear() очищает строку. */
47   str.clear();
48
49   std::cout << str.empty() << std::endl;
50   return 0;
51 }
```

1.2 Ввод/вывод строк

В отличие от контейнеров `std::string` поддерживает операции ввода/вывода.

```
1  #include <iostream>
2  #include <string>
3  #include <sstream>
4  #include <cstdio> // Это для printf()
5
6  int main() {
7    /* Класс поддерживает неявную инициализация строкой в кавычках, то есть строкой
8      типа const char*. Если, например, функция принимает const std::string& или
9      просто std::string, то в неё можно передать строку в кавычках без явного
10     преобразования типа. */
11   std::string str = "some string";
12
13   /* Для класса std::string определены операции сравнения, сравнивающие строки в
14     лексикографическом порядке. В примере один из аргументов операции > имеет
```

```
    тип std::string, а второй к нему преобразуется неявным образом. */
  if (str > "abc")
    std::cout << "'" << str << "' is greater than 'abc'" << std::endl;

  // Строки можно выводить в поток
  std::cout << str << std::endl;

  // Можно читать из потока
  stringstream ss;
  ss << "word" << std::endl << "some line 1" << std::endl << "some line 2";

  // Можно читать отдельные слова
  ss >> str;

  /* Можно считывать целые строки до перевода но новую строку. Если указать третьим
    аргументом разделитель, то функция будет считывать до разделителя. При этом
    разделитель вытаскивается из потока, но не добавляется в строку. По умолчанию
    разделитель равен '\n'. Функция возвращает поток. */
  while (std::getline(ss, str))
    std::cout << "Line: '" << str << "'" << std::endl;

  /* Для тех, кому нравится работать со строками в стиле C можно получить указатель
    на массив символов. */
  str = "some string";
  const char* data = str.data();
  std::printf("%s\n", data);
}
```

1.3 Операции, специфичные для строк

Кроме того, `std::string` поддерживает операции, специфичные именно для строк. Поясним на примере.

```
#include <iostream>
#include <string>

int main() {
  std::string str = "some string";

  // Строки можно складывать. Операции + и += осуществляют конкатенцию строк.
  str += " and another string";
  str = str + " and the third string";

  std::cout << str << std::endl;

  /* Для класса std::string определены операции сравнения, сравнивающие строки в
    лексикографическом порядке. Доступны все операции сравнения <, >, <=, >=, ==, !=.
    Отметим, что все эти операции доступны и в векторе. В примере один из аргументов
    операции > имеет тип std::string, а второй к нему преобразуется неявным образом. */
  if (str > "abc")
```

```
18    std::cout << "'" << str << "' is greater than 'abc'" << std::endl;
19
20  /* Метод replace() заменяет подстроку новой строкой. Длины не обязаны совпадать.
21    У метода есть довольно много перегрузок, приведём одну из них. Первый аргумент
22    (first) задаёт позицию, второй (count) длину подстроки, которую нужно заменить.
23    Третьим аргументом передаётся подстрока, на которую нужно заменить. Метод
24    возвращает саму строку. */
25  // Заменим первое вхождение "string" на "STRING".
26  str.replace(5, 6, "STRING");
27
28  std::cout << str << std::endl;
29
30  /* Метод принимает два аргумента: начало промежутка (first) и количество символов
31    (count). Метод копирует полуинтервал [first, first + count) в новую строку.
32    Позиции задаются индексами. По умолчанию, first равен нулю, а count равен
33    статической константе std::string::npos, означающей, что нужно скопировать символы
34    до конца строки. Метод возвращает новую строку. */
35  // Выделим подстроку из 10 символов от 5-го символа.
36  std::string anotherStr = str.substr(5, 10);
37
38  std::cout << anotherStr << std::endl;
39
40  // Выделим подстроку от 5-го символа до конца.
41  std::string anotherStr2 = str.substr(5);
42
43  std::cout << anotherStr2 << std::endl;
44
45  /* Вставка символов на определённую позицию. Отличие от обычных контейнеров
46    заключается в том, что позиция задаётся индексом. Поддерживается вставка
47    отдельных символов, строк типа std::string и const char*. Метод возвращает саму
48    строку, в которую было осуществлена вставка. */
49  // Вставим 3 символа 'a' на позицию 5.
50  std.insert(5, 3, 'a');
51
52  // Вставим строку "abcde" на позицию 10.
53  std.insert(10, "abcde");
54
55  /* У метода точно такие же аргументы, как и у substr(). Метод удаляет полуинтервал
56    [first, first + count). Метод возвращает саму строку. */
57  // Удалить полуинтервал [10, 15).
58  str.erase(10, 15);
59  // Удалить все символы от 5-ого.
60  str.erase(5);
61
62  return 0;
63 }
```

Также есть метод `copy()`, позволяющий скопировать подстроку в заранее выделенный массив элементов типа `char`.

1.4 Преобразование из строки в число и обратно

Для преобразования числа в строку используется функция `std::to_string()`. Поддерживаются как целые числа, так и числа с плавающей точкой.

```
std::string str;

str = std::to_string(10);

std::cout << str << std::endl;

str = std::to_string(3.14);

std::cout << str << std::endl;
```

Для конвертации строки в число используется семейство функций `std::sto*()`. Например, для преобразования строки в целое типов `int` или `long` можно использовать `std::stoi()` или `std::stol()` соответственно. Для преобразования строки в числа типа `double` или `float` можно использовать функции `std::stod()` или `std::stof()` соответственно.

Все эти функции работают одинаково. Принимают строку первым аргументом и возвращают число. У каждой функции есть необязательный второй параметр типа `std::size_t*`, в который может быть записано число обработанных символов, то есть позиция первого необработанного символа. Функции для конвертации в целые числа имеют необязательный третий аргумент — основание системы исчисления, равный десяти по умолчанию. Функции игнорируют пробелы в начале строки. Функции могут выбросить исключение `std::invalid_argument`, если не было осуществлено преобразования или `std::out_of_range` если значение оказалось за пределами соответствующего типа данных. Все функции являются обёртками надо соответсвующими функциями из `<cstdlib>` (например, `std::strtol()`, `std::strtod()`, `std::strtof()`.

```
std::string str = "1 2 3 4 abcde";

// Самый простой вариант
int num = std::stoi(str);

std::size_t pos = 0;

while(1) {
  try {
    // Считаем следующее число
    num = std::stoi(str, &pos);

    // Обрежем строку
    str = str.substr(pos);

    std::cout << "Read " << num << ". The remaining string equals " << str << std::endl;
  } catch(std::invalid_argument&) {
    std::cout << "Can't convert any more." << std::endl;
    break; // Выходим из цикла
  }
}
```

```
22
23 // Для чисел с плавающей точкой всё аналогично
24 str = "3.14";
25 double = std::stod(str);
```

1.5 Методы поиска

Метод `find()` предназначен для нахождения первого вхождения символа или подстроки в строке. Функция принимает символ (`char`) или подстроку (`const std::string&` или `const char*`) для поиска, а также необязательный параметр — позицию, с которой начинать поиск, равную по умолчанию нулю. Функция возвращает индекс первого вхождения или константу `std::string::npos` если вхождения не обнаружено.

Метод `rfind()` ищет последнее вхождение (с конца строки). Позиция начала поиска по умолчанию равна `std::string::npos`.

```
1  /* Разобьём строку на части, используя в качестве разделителя слово "and". */
2  std::string str = "123 and abcde and 456 and defg";
3  std::size_t next = str.find("and"); // Ищем первое вхождение
4  const std::size_t len = 3; // Длина искомой строки "and".
5  std::size_t prev = 0;
6
7  while (next != std::string::npos) {
8    // Проверяем, что токен не пуст
9    if (next > prev) {
10     std::string token = str.substr(prev, next - prev);
11     std::cout << "Found token '" << token << "'!" << std::endl;
12   }
13
14   prev = next + len;
15   next = str.find("and", prev); // Ищем следующее вхождение
16 }
17
18 if (prev != str.size()) { // Печатаем последнее слово
19   std::string token = str.substr(prev);
20   std::cout << "Found token '" << token << "'!" << std::endl;
21 }
```

Кроме того, есть методы для поиска в строке первого элемента из набора (`find_first_of()`), не из набора (`find_first_not_of()`). Параметры и возвращаемое значение у этих методов те же, что и у метода `find()`. Есть также методы, выполняющие поиск с конца.

```
1  /* Разобьём строку на части по набору разделителей. */
2  std::string str = "123 and abcde and 456 and defg";
3  std::size_t next = str.find_first_of(" \t\n\r"); // Ищем первый разделитель
4  std::size_t prev = 0;
5
6  while (next != std::string::npos) {
7    // Проверяем, что токен не пуст
8    if (next > prev) {
```

```
    std::string token = str.substr(prev, next - prev);
    std::cout << "Found token '" << token << "'!" << std::endl;
  }

  prev = next + 1;
  next = str.find_first_of(" \t\n\r", prev); // Ищем следующий разделитель
}

if (prev != str.size()) { // Печатаем последнее слово
  std::string token = str.substr(prev);
  std::cout << "Found token '" << token << "'!" << std::endl;
}
```

2 Класс std::string_view (C++17)

У класса `std::string` есть одна проблема: он не позволяет получить подстроку не скопировав её в отдельный экземпляр класса `string`. Таким образом при извлечении подстроки происходит выделение памяти м копирование символов.

Во многих задачах требуется получить некоторую подстроку только для чтения. Для того, чтобы сделать это без копирования элементов, в C++17 был введён невладеющий контейнер шаблон `std::basic_string_view` с одним обязательным шаблонным параметром — типом данных символа. Шаблон определён в заголовочном файле `<string_view>`. Невладеющий означает, что он не хранит данных внутри себя, он хранит лишь указатель на начало промежутка и указатель на конец промежутка. Таким образом, `std::basic_string_view` может ссылаться на непрерывный промежуток элементов в памяти.

Для работами со строками, состоящими из символов типа `char` в заголовочном файле `<string_view>` определён класс `std::string_view`, являющийся конкретизацией шаблона `std::basic_string_view` типом `char` (то есть `std::basic_string_view<char>`).

```
std::string str = "string";
const char* arr = "array of const chars";

/* std::string_view можно создать из класса std::string или из массива char'ов.
  Объекты типа std::string_view можно неявно создавать из строк. */
std::string_view strView = str;

/* Сложность этого конструктора линейная, поскольку требуется подсчитать длину
  строки. */
std::string_view arrView = arr;

/* std::string_view поддерживает все те же методы, предназначенные для чтения
  элементов, что и std::string. */
std::cout << "View size equals " << strView.size()  << std::endl;
std::cout << "The third symbol equals " << << strView[2] << std::endl;
std::cout << strView.empty() << std::endl;

// Также можно получить указатель на const char
const char* data = strView.data();
```

Отметим, что `std::string_view` не позволяет менять символы в строке, на которую он ссылается. Кроме того, `std::string_view` может инвалидироваться в том случае, если освободится память, на которую он ссылается. Например, если добавить в строку типа `std::string`, на которую ссылается `std::string_view`, несколько символов, то это может привести к перевыделению памяти внутри контейнера `std::string` и следовательно к инвалидации `std::string_view`.

У класса `std::string_view` также есть итераторы, предоставляющие произвольный прямой/обратный доступ на чтение.

2.1 Извлечение подстрок

```
std::string str = "There were four of us --- George, and William Samuel Harris, "
                  "and myself, and Montmorency.";
std::string_view view = str;

/* Метод substr() принимает два аргумента: первый (pos) --- начало подстроки
  и второй (count) количество символов в подстроке. Оба аргумента необязательный,
  pos по умолчанию равен 0, а count статической константе std::string_view::npos,
  означающей, что нужно брать подстроку до конца. Метод возвращает новый string_view
  на требуемую подстроку. */
std::string_view george = view.substr(26, 6);

std::cout << george << std::endl;

std::string_view persons = view.substr(26);

std::cout << persons << std::endl;

/* В тех случаях, когда требуется изменить текущий string_view, можно воспользоваться
  методами, убирающими из начала или из конца N символов. Методы называются
  remove_prefix() и remove_suffix() соответственно. Их поведение не определено,
  если передать в них число, большее количества символов в string_view. */

str.remove_prefix(26);

std::cout << str << std::endl; // Выведет всё, начиная с "George".

str.remove_suffix(18);

std::cout << str << std::endl; // Выведет от "George" до "myself".
```

2.2 Методы поиска

В классе `std::string_view` есть все те же методы поиска, которые есть и в классе `string`. Эти методы принимают символ (типа `char`) или набор символов (либо `const char*` либо `std::string_view`, а следовательно и `string`) первым аргументом, кроме того, вторым необязательным аргументом они принимают позицию, с которой нужно начинать поиск. Методы возвращают позицию первого (или последнего) вхождения или константу `std::string_view::npos` если символ/подсткрока не были найдены.

Напишем функцию, извлекающую токены по заданному набору разделителей:

```cpp
std::string_view GetNextToken(std::string_view& str, std::string_view delimeters) {
  std::string_view retval; // Изначально view пуст.

  while (retval.size() == 0u) { // Избегаем подряд идущих разделителей
    std::size_t pos = str.find_first_of(delimeters);

    if (pos == std::string_view::npos)
      return retval;

    retval = str.substr(0, pos);

    str.remove_prefix(pos + 1);
  }

  return retval;
}
```

Обратите, что объекты `std::string_view` достаточно легковесные, их можно передавать в функцию без ссылки.

Разобьём строку на слова при помощи предложенной функции.

```cpp
std::string str = "There were four of us --- George, and William Samuel Harris, "
                  "and myself, and Montmorency.";
std::string_view view = str;

std::string_view token = GetNextToken(view, " -,.");

while (token.size() != 0) {
  std::cout << "Found token '" << token << "'!" << std::endl;
  token = GetNextToken(view, " -,.");
}
```

Список литературы

[1] https://en.cppreference.com/w/cpp/string/basic_string

[2] https://en.cppreference.com/w/cpp/string/basic_string_view

[3] https://en.cppreference.com/w/cpp/regex