Ravesli Ravesli

- Уроки по C++
- OpenGL
- SFML
- Qt5
- RegExp
- Ассемблер
- Купить .PDF

Введение в класс std::string_view в C++

Дмитрий Бушуев |

- Уроки C++

|

Обновл. 3 Окт 2020 |

12757

| 3

На этом уроке мы рассмотрим класс std::string_view, который является нововведением стандарта C++17.

Оглавление:

1. Проблемы строк
2. Введение в класс std::string_view
3. Функции, модифицирующие представление
4. std::string_view и обычные строки
5. Проблемы владения и доступа
6. Конвертация std::string_view в std::string
7. Конвертация std::string_view в строку C-style
8. Функция data()
9. Нюансы std::string_view

Проблемы строк

На уроке о **строках C-style** мы говорили об опасностях, которые возникают при их использовании. Конечно, строки C-style работают быстро, но при этом их использование не является таким уж простым и безопасным в сравнении с **std::string**.

Правда, стоит отметить, что и у std::string имеются свои недостатки, особенно когда речь заходит об использовании **константных** строк.

Рассмотрим следующий пример:

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

int main()
{
  char text[]{ "hello" };
  std::string str{ text };
  std::string more{ str };

  std::cout << text << ' ' << str << ' ' << more << '\n';

  return 0;
}
```

Результат выполнения программы:

```
hello hello hello
```

Внутри функции main() выполняется копирование строки `hello` 3 раза, в результате чего мы имеем 4 копии исходной строки:

- первая копия — это непосредственно сам строковый **литерал** `hello`, который создается на этапе компиляции и хранится в бинарном виде;
- еще одна копия создается при инициализации **массива** типа char;
- далее идут объекты `str` и `more` класса std::string, каждый из которых, в свою очередь, создает еще по одной копии строки.

Из-за того, что класс std::string спроектирован так, чтобы его объекты могли быть изменяемыми, каждому объекту класса std::string приходится хранить свою собственную копию строки. Благодаря этому, исходная строка может быть изменена без влияния на другие объекты std::string.

Это также справедливо и для константных строк (`const std::string`), несмотря на то, что подобные объекты не могут быть изменены.

Введение в класс std::string_view

В качестве следующего примера возьмем окно в вашем доме и автомобиль, стоящий на улице неподалеку. Вы можете посмотреть через окно и увидеть машину, но при этом вы не можете дотронуться до машины или передвинуть её. Ваше окно лишь обеспечивает вид на автомобиль, который является отдельным независимым от вас объектом.

В **стандарте C++17** вводится еще один способ использования строк — с помощью **класса std::string_view**, который находится в **заголовочном файле** string_view.

В отличие от объектов класса std::string, которые хранят свою собственную копию строки, класс std::string_view обеспечивает **представление** (англ. ***«view»***) для заданной строки, которая может быть

определена где-нибудь в другом месте.

Попробуем переписать код предыдущего примера, заменив каждое вхождение `std::string` на `std::string_view`:

```
#include <iostream>
#include <string_view>

int main()
{
  std::string_view text{ "hello" }; // представление для строки "hello", которое храни
  std::string_view str{ text }; // представление этой же строки - "hello"
  std::string_view more{ str }; // представление этой же строки - "hello"

  std::cout << text << ' ' << str << ' ' << more << '\n';

  return 0;
}
```

В результате мы получим точно такой же вывод на экран, как и в предыдущем примере, но при этом у нас не будут созданы лишние копии строки `hello`. Когда мы копируем объект класса std::string_view, то новый объект std::string_view будет «смотреть» на ту же самую строку, на которую «смотрел» исходный объект. Ко всему прочему, класс std::string_view не только быстр, но и обладает многими функциями, которые мы изучили при работе с классом std::string:

```
#include <iostream>
#include <string_view>

int main()
{
  std::string_view str{ "Trains are fast!" };

  std::cout << str.length() << '\n'; // 16
  std::cout << str.substr(0, str.find(' ')) << '\n'; // Trains
  std::cout << (str == "Trains are fast!") << '\n'; // 1

  // Начиная с C++20
  std::cout << str.starts_with("Boats") << '\n'; // 0
  std::cout << str.ends_with("fast!") << '\n'; // 1

  std::cout << str << '\n'; // Trains are fast!

  return 0;
}
```

Т.к. объект класса std::string_view не создает копии строки, то, изменив исходную строку, мы, тем самым, повлияем и на её представление в связанном с ней объектом std::string_view:

```
#include <iostream>
```

```
#include <string_view>

int main()
{
  char arr[]{ "Gold" };
  std::string_view str{ arr };

  std::cout << str << '\n'; // Gold

  // Изменяем 'd' на 'f' в arr
  arr[3] = 'f';

  std::cout << str << '\n'; // Golf

  return 0;
}
```

Изменяя `arr`, можно видеть, как изменяется и `str`. Это происходит из-за того, что исходная строка является общей для этих переменных. Стоит отметить, что при использовании объектов класса std::string_view лучше избегать модифицирования исходной строки, пока существуют связанные с ней объекты класса std::string_view, так как в противном случае, это может привести к путанице и ошибкам.

Совет: Используйте std::string_view вместо строк C-style. Для строк, которые не планируете изменять в дальнейшем, предпочтительнее использовать класс std::string_view вместо std::string.

Функции, модифицирующие представление

Вернемся к нашей аналогии с окном, только теперь рассмотрим окно с занавесками. Мы можем закрыть часть окна левой или правой занавеской, тем самым уменьшив то, что можно увидеть сквозь окно. Заметьте, мы не изменяем объекты, находящиеся снаружи окна, изменяется лишь сектор наблюдения из окна.

Аналогично и с классом std::string_view: в нем содержатся функции, позволяющие нам управлять представлением строки. Благодаря этому мы можем изменять представление строки без изменения исходной строки.

Для этого используются следующие функции:

- `remove_prefix()` — удаляет символы из левой части представления;
- `remove_suffix()` — удаляет символы из правой части представления.

Например:

```
#include <iostream>
#include <string_view>

int main()
{
  std::string_view str{ "Peach" };
```

```
7
8     std::cout << str << '\n';
9
10    // Игнорируем первый символ
11    str.remove_prefix(1);
12
13    std::cout << str << '\n';
14
15    // Игнорируем последние 2 символа
16    str.remove_suffix(2);
17
18    std::cout << str << '\n';
19
20    return 0;
21 }
```

Результат выполнения программы:

```
Peach
each
ea
```

В отличие от настоящих занавесок, с помощью которых мы закрыли часть окна, объекты класса std::string_view нельзя «открыть обратно». Изменив однажды область видимости, вы уже не сможете вернуться к первоначальным значениям (стоит отметить, что есть приемы, которые позволяют решить данную проблему, но вдаваться в них мы не будем).

std::string_view и обычные строки

В отличие от строк C-Style, объекты классов std::string и std::string_view не используют **нулевой символ** (**нуль-терминатор**) в качестве метки для обозначения конца строки. Данные объекты знают, где заканчивается строка, т.к. отслеживают её длину:

```
1  #include <iostream>
2  #include <iterator> // для функции std::size()
3  #include <string_view>
4
5  int main()
6  {
7    // Нет нуль-терминатора
8    char vowels[]{ 'a', 'e', 'i', 'o', 'u' };
9
10   // Массив vowels не является нуль-терминированным. Мы должны передавать длину вручну
11   // Поскольку vowels является массивом, то мы можем использовать функцию std::size(),
12   std::string_view str{ vowels, std::size(vowels) };
13
14   std::cout << str << '\n'; // это безопасно, так как std::cout знает, как выводить st
```

```
15
16    return 0;
17  }
```

Результат выполнения программы:

```
aeiou
```

Проблемы владения и доступа

Поскольку std::string_view является всего лишь представлением строки, его время жизни не зависит от времени жизни строки, которую он представляет. Если отображаемая строка выйдет за пределы области видимости, то std::string_view больше не сможет её отображать и при попытке доступа к ней мы получим неопределенные результаты:

```
1  #include <iostream>
2  #include <string>
3  #include <string_view>
4
5  std::string_view askForName()
6  {
7    std::cout << "What's your name?\n";
8
9    // Используем std::string, поскольку std::cin будет изменять строку
10   std::string str{};
11   std::cin >> str;
12
13   // Мы переключаемся на std::string_view только в демонстрационных целях.
14   // Если вы уже имеете std::string, то нет необходимости переключаться на std::string_
15   std::string_view view{ str };
16
17   std::cout << "Hello " << view << '\n';
18
19   return view;
20 } // str уничтожается и, таким образом, уничтожается и строка, созданная str
21
22 int main()
23 {
24   std::string_view view{ askForName() };
25
26   // view пытается обратиться к строке, которой уже не существует
27   std::cout << "Your name is " << view << '\n'; // неопределенное поведение
28
29   return 0;
30 }
```

Результат выполнения программы:

```
What's your name?
nascardriver
Hello nascardriver
Your name is ▯P@▯P@
```

Когда мы объявили переменную `str` и с помощью std::cin присвоили ей определенное значение, то данная переменная создала внутри себя строку, разместив её в динамической области памяти. После того, как переменная `str` вышла за пределы области видимости в конце функции askForName(), внутренняя строка вслед за этим прекратила свое существование. При этом объект класса std::string_view не знает, что строки больше не существует, и всё также позволяет нам к ней обратиться. Попытка доступа к такой строке через её представление в функции main() приводит к неопределенному поведению, в результате чего мы получаем **кракозябры**.

Такая же ситуация может произойти и тогда, когда мы создаем объект std::string_view из объекта std::string, а затем модифицируем первоначальный объект std::string. Изменение объекта std::string может привести к созданию в другом месте новой внутренней строки и последующему уничтожению старой. При этом std::string_view продолжит «смотреть» в то место, где была старая строка, но её там уже не будет.

***Предупреждение*:** Следите за тем, чтобы исходная строка, на которую ссылается объект std::string_view, не выходила за пределы области видимости и не изменялась до тех пор, пока используется ссылающийся на нее объект std::string_view.

Конвертация std::string_view в std::string

Объекты класса std::string_view не конвертируются неявным образом в объекты класса std::string, но конвертируются при явном преобразовании:

```
1  #include <iostream>
2  #include <string>
3  #include <string_view>
4
5  void print(std::string s)
6  {
7    std::cout << s << '\n';
8  }
9
10 int main()
11 {
12   std::string_view sv{ "balloon" };
13
14   sv.remove_suffix(3);
15
16   // print(sv); // ошибка компиляции: неявная конвертация запрещена
17
18   std::string str{ sv }; // явное преобразование
19
20   print(str); // ок
```

```
21
22    print(static_cast<std::string>(sv)); // ок
23
24    return 0;
25  }
```

Результат выполнения программы:

```
ball
ball
```

Конвертация std::string_view в строку C-style

Некоторые старые функции (такие как `strlen()`) работают только со строками C-style. Для того чтобы преобразовать объект класса std::string_view в строку C-style, мы сначала должны конвертировать его в объект класса std::string:

```
1  #include <cstring>
2  #include <iostream>
3  #include <string>
4  #include <string_view>
5
6  int main()
7  {
8    std::string_view sv{ "balloon" };
9
10   sv.remove_suffix(3);
11
12   // Создание объекта std::string из объекта std::string_view
13   std::string str{ sv };
14
15   // Получаем строку C-style с нуль-терминатором
16   auto szNullTerminated{ str.c_str() };
17
18   // Передаем строку с нуль-терминатором в функцию, которую мы хотим использовать
19   std::cout << str << " has " << std::strlen(szNullTerminated) << " letter(s)\n";
20
21   return 0;
22 }
```

Результат выполнения программы:

```
ball has 4 letter(s)
```

Однако стоит учитывать, что создание объекта класса std::string всякий раз, когда мы хотим преобразовать объект std::string_view в строку C-style, является дорогостоящей операцией, поэтому мы должны по возможности избегать подобных ситуаций.

Функция data()

Доступ к исходной строке объекта std::string_view можно получить при помощи **функции data()**, которая возвращает строку C-style. При этом обеспечивается быстрый доступ к представляемой строке (как к строке C-style). Но это следует использовать только тогда, когда объект std::string_view не был изменен (например, при помощи функций `remove_prefix()` или `remove_suffix()`) и связанная с ним строка имеет нуль-терминатор (так как это строка C-style).

В следующем примере функция std::strlen() ничего не знает о std::string_view, поэтому мы передаем ей функцию `str.data()`:

```
#include <cstring> // для функции std::strlen()
#include <iostream>
#include <string_view>

int main()
{
  std::string_view str{ "balloon" };

  std::cout << str << '\n';

  // Для простоты мы воспользуемся функцией std::strlen(). Вместо нее можно было бы ис
  // Здесь мы можем использовать функцию data(), так как мы не изменяли представление
  std::cout << std::strlen(str.data()) << '\n';

  return 0;
}
```

Результат выполнения программы:

```
balloon
7
```

Когда мы пытаемся обратиться к объекту класса std::string_view, который был изменен, функция data() может вернуть совсем не тот результат, который мы ожидали от нее получить. В следующем примере показано, что происходит, когда мы обращаемся к функции data() после изменения представления строки:

```
#include <cstring>
#include <iostream>
#include <string_view>

int main()
{
  std::string_view str{ "balloon" };

  // Удаляем символ "b"
  str.remove_prefix(1);

  // Удаляем часть "oon"
```

```
  str.remove_suffix(3);

  // Помните, что предыдущие 2 команды не изменяют исходную строку, они работают лишь
  std::cout << str << " has " << std::strlen(str.data()) << " letter(s)\n";
  std::cout << "str.data() is " << str.data() << '\n';
  std::cout << "str is " << str << '\n';

  return 0;
}
```

Результат выполнения программы:

```
all has 6 letter(s)
str.data() is alloon
str is all
```

Очевидно, что данный результат — это не то, что мы планировали увидеть, и он является следствием попытки функции data() получить доступ к данным представления std::string_view, которое было изменено. Информация о длине строки теряется при обращении к ней через функцию data(). std::strlen и std::cout продолжают считывать символы из исходной строки до тех пор, пока не встретят нуль-терминатор, который находится в конце строки `baloon`.

***Предупреждение*:** Используйте std::string_view::data() только в том случае, если представление std::string_view не было изменено и отображаемая строка содержит завершающий нулевой символ (нуль-терминатор). Использование функции std::string_view::data() со строкой без нуль-терминатора чревато возникновением ошибок.

Нюансы std::string_view

Будучи относительно недавним нововведением, класс std::string_view реализован не так уж и идеально, как хотелось бы:

```
std::string s{ "hello" };
std::string_view v{ "world" };

// Не работает
std::cout << (s + v) << '\n';
std::cout << (v + s) << '\n';

// Потенциально небезопасно или не то, что мы хотим получить,
// поскольку мы пытаемся использовать объект std::string_view в качестве строки C-styl
std::cout << (s + v.data()) << '\n';
std::cout << (v.data() + s) << '\n';

// Приемлемо, т.к. нам нужно создать новый объект std::string, но некрасиво и нерацион
std::cout << (s + std::string{ v }) << '\n';
std::cout << (std::string{ v } + s) << '\n';
std::cout << (s + static_cast<std::string>(v)) << '\n';
std::cout << (static_cast<std::string>(v) + s) << '\n';
```

Нет никаких причин для неработоспособности строк №5-6, но тем не менее они не работают. Вероятно, полная поддержка данного функционала будет реализована в следующих версиях стандарта C++.

Оценить статью:

(**93** оценок, среднее: **4,84** из 5)

Урок №79. Строки C-style

Урок №80. Указатели

Комментариев: 3

1.  *Сергій*:
   15 июля 2020 в 10:36

   Очень полезно что автор знакомит с разными стандартами, и плюсами\минусами новых стандартов.

   Ответить

2.  *Codder*:
   5 июня 2020 в 13:26

   "Нет никаких причин на то, почему бы строки №5-6 были бы неработоспособными, но, тем не менее, они не работают. Вероятно, полная поддержка данного функционала будет реализована в следующих версиях стандарта C++."

   Прочитать, чтобы понять — что std::string_view пока что фуфло. И лучше использовать обычный string, поскольку ни одного преимущества выявить не удалось.

   Ответить

   1.  *foo*:
      8 сентября 2020 в 21:13

      imho, все на производительность заточено. по сути std::string покрывает все потребности, но это неэффективно в некоторых случаях … очень _не_ эффективно. c++ с этим не может мириться (с такими накладными расходами по памяти и процессору) и вот … частное решение std::string_view, которое частично (т.е. если задача позволяет) эту проблему хоть как-то сглаживает.

Ответить

Добавить комментарий

Ваш E-mail не будет опубликован. Обязательные поля помечены *

Имя *

Email *

Комментарий

Сохранить моё Имя и E-mail. Видеть комментарии, отправленные на модерацию

Получать уведомления о новых комментариях по электронной почте. Вы можете подписаться без комментирования.

Отправить комментарий

TELEGRAM КАНАЛ

ПАБЛИК

ТОП СТАТЬИ

- Словарь программиста. Сленг, который должен знать каждый кодер
- Урок №1. Введение в программирование
- 70+ бесплатных ресурсов для изучения программирования
- Урок №1: Введение в создание игры «SameGame» на C++/MFC
- Урок №4. Установка IDE (Интегрированной Среды Разработки)