

Урок №200. Строковые классы `std::string` и `std::wstring`



Юрий | [Уроки по C++](#) | Обновл. 15 Сен 2021 | 32188 | 4

Стандартная библиотека C++ содержит много полезных классов, но одним из наиболее полезных является `std::string`. **`std::string`** (и **`std::wstring`**) — это строковый класс, который позволяет выполнять операции присваивания, сравнения и изменения строк. На следующих нескольких уроках мы подробно рассмотрим строковые классы Стандартной библиотеки C++.

Примечание: **Строки C-style** обычно называют «строками C-style», тогда как `std::string` (и `std::wstring`) обычно называют просто «строками».

Оглавление:

1. Зачем нужен std::string?
2. Класс std::string
3. Функционал std::string
 - Создание и удаление
 - Размер и ёмкость
 - Доступ к элементам
 - Изменение
 - Ввод/вывод
 - Сравнение строк
 - Подстроки и конкатенация
 - Поиск
 - Поддержка итераторов и распределителей

4. Заключение

Зачем нужен `std::string`?

Мы уже знаем, что строки C-style используют **массивы** типа `char` для хранения целой строки. Если вы попытаетесь что-либо сделать со строками C-style, то вы очень быстро обнаружите, что работать с ними трудно, запутаться легко, а проводить **отладку** сложно.

Строки C-style имеют много недостатков, в первую очередь связанных с тем, что вы должны самостоятельно управлять памятью. Например, если вы захотите поместить строку `Hello!` в буфер, то вам сначала нужно будет **динамически выделить** буфер правильной длины:

```
1 char *strHello = new char[7];
```

Не забудьте учесть дополнительный символ для нуль-терминатора! Затем вам нужно будет скопировать значение:

```
1 strcpy(strHello, "Hello!");
```

И здесь вам нельзя прогадать с длиной буфера, иначе произойдет **переполнение**! И, конечно, поскольку строка выделяется динамически, то вы должны её еще и правильно удалить:

```
1 delete[] strHello;
```

Не забудьте использовать форму оператора `delete`, которая работает с массивами, а не обычную форму оператора `delete`.

Кроме того, многие из интуитивно понятных операторов, которые предоставляет язык C++ для работы с числами, такие как `=`, `==`, `!=`, `<`, `>`, `>=` и `<=` попросту не работают со строками C-style. Иногда они могут работать без ошибок со стороны компилятора, но результат будет неверным. Например, сравнение двух строк C-style с

использованием оператора `==` на самом деле выполнит сравнение **указателей**, а не строк. Присваивание одной строки C-style другой строке C-style с использованием оператора `=` будет работать, но выполняться будет копирование указателя (**поверхностное копирование**), что не всегда то, что нам нужно. Такие вещи могут легко привести к ошибкам и сбоям в программе, а разбираться с ними не так уж и легко (относительно)!

Суть в том, что работая со строками C-style, вам нужно помнить множество придирчивых правил о том, что делать безопасно, а что — нет; запоминать много функций, таких как `strcat()` и `strcmp()`, чтобы использовать их вместо интуитивных операторов; а также самостоятельно выполнять управление памятью.

К счастью, язык C++ предоставляет гораздо лучший способ для работы со строками: классы `std::string` и `std::wstring`. Используя такие концепции C++, как **конструкторы**, **деструкторы** и **перегрузку операторов**, `std::string` позволяет создавать и манипулировать строками в интуитивно понятной форме и, что не менее важно, выполнять это безопасно! Никакого управления памятью, запоминания странных названий функций и значительно меньшая вероятность возникновения ошибок/сбоев.

Класс `std::string`

Весь функционал класса `std::string` находится в **заголовочном файле** `string`:

```
1 #include <string>
```

На самом деле в заголовочном файле есть 3 разных строковых класса. Первый — это **шаблон класса** с именем `basic_string<>`, который является **родительским классом**:

```
1 namespace std
2 {
3     template<class charT, class traits = char_traits<charT>, class Allocator = allocator<charT> >
4         class basic_string;
5 }
```

Вы не будете работать с этим классом напрямую, так что не беспокойтесь о том, что такое `traits` или `Allocator`. Значений по умолчанию, которые присваиваются этим объектам, будет достаточно почти во всех мыслимых и немыслимых случаях.

Дальше идут две разновидности `basic_string<>`:

```
1 namespace std
2 {
3     typedef basic_string<char> string;
4     typedef basic_string<wchar_t> wstring;
5 }
```

Это те два класса, которые вы будете использовать. **`std::string` используется для стандартных ASCII-строк (кодировка UTF-8), а `std::wstring` используется для Unicode-строк (кодировка UTF-16).** Встроенного класса для строк UTF-32 нет.

Хотя вы будете напрямую использовать `std::string` и `std::wstring`, весь функционал реализован в классе `basic_string<>`. `string` и `wstring` имеют доступ к этому функционалу напрямую. Следовательно, все функции, приведенные ниже, работают как со `string`, так и с `wstring`.

Функционал `std::string`

Создание и удаление:

- **конструктор** — создает или копирует строку;
- **деструктор** — уничтожает строку.

Размер и ёмкость:

- **capacity()** — возвращает количество символов, которое строка может хранить без дополнительного перевыделения памяти;
- **empty()** — возвращает логическое значение, указывающее, является ли строка пустой;
- **length(), size()** — возвращают количество символов в строке;
- **max_size()** — возвращает максимальный размер строки, который может быть выделен;
- **reserve()** — расширяет или уменьшает ёмкость строки.

Доступ к элементам:

- **[] , at()** — доступ к элементу по заданному индексу.

Изменение:

- **= , assign()** — присваивают новое значение строке;
- **+= , append(), push_back()** — добавляют символы к концу строки;
- **insert()** — вставляет символы в произвольный индекс строки;
- **clear()** — удаляет все символы строки;
- **erase()** — удаляет символы по произвольному индексу строки;
- **replace()** — заменяет символы произвольных индексов строки другими символами;
- **resize()** — расширяет или уменьшает строку (удаляет или добавляет символы в конце строки);
- **swap()** — меняет местами значения двух строк.

Ввод/вывод:

- `>>`, `getline()` — считывают значения из входного потока в строку;
- `<<` — записывает значение строки в выходной поток;
- `c_str()` — конвертирует строку в строку C-style с нуль-терминатором в конце;
- `copy()` — копирует содержимое строки (которое без нуль-терминатора) в массив типа `char`;
- `data()` — возвращает содержимое строки в виде массива типа `char`, который не заканчивается нуль-терминатором.

Сравнение строк:

- `==`, `!=` — сравнивают, являются ли две строки равными/неравными (возвращают значение типа `bool`);
- `<`, `<=`, `>`, `>=` — сравнивают, являются ли две строки меньше или больше друг друга (возвращают значение типа `bool`);
- `compare()` — сравнивает, являются ли две строки равными/неравными (возвращает `-1`, `0` или `1`).

Подстроки и конкатенация:

- `+` — соединяет две строки;
- `substr()` — возвращает подстроку.

Поиск:

- `find` — ищет индекс первого символа/подстроки;
- `find_first_of` — ищет индекс первого символа из набора символов;

- **find_first_not_of** — ищет индекс первого символа НЕ из набора символов;
- **find_last_of** — ищет индекс последнего символа из набора символов;
- **find_last_not_of** — ищет индекс последнего символа НЕ из набора символов;
- **rfind** — ищет индекс последнего символа/подстроки.

Поддержка **итераторов** и **распределителей (allocators)**:

- **begin(), end()** — возвращают «прямой» итератор, указывающий на первый и последний (элемент, который идет за последним) элементы строки;
- **get_allocator()** — возвращает распределитель;
- **rbegin(), rend()** — возвращают «обратный» итератор, указывающий на последний (т.е. «обратное» начало) и первый (элемент, который предшествует первому элементу строки — «обратный» конец) элементы строки. Отличие от **begin()** и **end()** в том, что движение итераторов происходит в обратную сторону.

Заключение

Вот и весь функционал `std::string`. Большинство из этих функций имеют несколько разновидностей для обработки разных типов входных данных, об этом мы еще поговорим.

Хотя функционал достаточно широк, все же есть несколько заметных упущений:

- поддержка **регулярных выражений**;
- конструкторы для создания строк из чисел;
- прописные/строчные функции;

- токенизация/разбиение строк на массивы;
- простые функции для получения левой или правой части строки;
- обрезка пробелов;
- конвертация из UTF-8 в UTF-16 и наоборот.

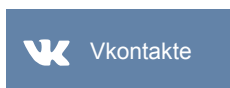
Всё вышеперечисленное вам придется реализовать самостоятельно, либо конвертировать вашу строку в строку C-style (используя функцию `c_str()`) и тогда уже использовать функционал, который предлагает язык C++.

Примечание: Хотя мы используем `string` в наших примерах, всё это также применимо и к `wstring`.

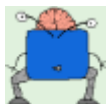
Оценить статью:

★★★★★ (117 оценок, среднее: **4,92** из 5)

Поделиться в социальных сетях:



Комментариев: 4



Rock

17 ноября 2020 в 13:39

Скучная глава ибо тут чисто техническая информация которая полезна как справочник если подзабыл. Но в голове держать это тяжело да и не нужно скорее всего (10-15% полезного, остальное интуитивно понятно и не требует заучивания).

Ответить



Илья

11 июля 2019 в 15:16

Это ж юбилей, 200 уроков и скоро, конец... интересное чувство....

Урок №201. Создание, уничтожение и конвертация std::string



Юрий | [Уроки по C++](#) | Обновл. 15 Сен 2021 | 18781 | 2

На этом уроке мы рассмотрим, как создаются объекты `std::string` в языке C++, а также создание строк из чисел (и наоборот).

Оглавление:

1. [Создание std::string](#)
2. [Создание std::string из чисел](#)

3. Конвертация std::string в числа

Создание std::string

Строковые классы имеют ряд **конструкторов** и **деструктор**, которые можно использовать для создания строк. Мы рассмотрим каждый из них.

string::string()

→ Конструктор по умолчанию, который создает пустую строку.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sSomething;
7     std::cout << sSomething;
8
9     return 0;
10 }
```

Результат:



string::string(const string& strString)

→ Конструктор копирования, который создает новую строку путем копирования `strString`.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sSomething("What a string!");
7     std::string sOutput(sSomething);
8     std::cout << sOutput;
9
10    return 0;
11 }
```

Результат:

What a string!

`string::string(const string& strString, size_type unIndex)`

`string::string(const string& strString, size_type unIndex, size_type unLength)`

- Конструкторы, которые создают новые строки, которые состоят из строки `strString` (начиная с индекса `unIndex`) и количества символов, указанных в `unLength`.
- Если компилятор встречает `NULL`, то копирование строки завершается, даже если `unLength` не был достигнут.
- Если `unLength` не был указан, то все символы, начиная с `unIndex`, будут использованы.
- Если `unIndex` больше, чем размер строки, то **выбрасывается исключение** `out_of_range`.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sSomething("What a string!");
7     std::string sOutput(sSomething, 3);
8     std::cout << sOutput<< std::endl;
9     std::string sOutput2(sSomething, 5, 6);
10    std::cout << sOutput2 << std::endl;
11
12    return 0;
13 }
```

Результат:

```
t a string
a stri
```

string::string(const char *szCString)

→ Конструктор, который создает новую строку из передаваемой **строки C-style** `szCString` вплоть до нуль-терминатора (но его не включает). Если размер результата превышает максимальную длину строки, то генерируется исключение `length_error`.

Предупреждение: `szCString` не должен быть `NULL`.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     const char *szSomething("What a string!");
7     std::string sOutput(szSomething);
8     std::cout << sOutput << std::endl;
9
10    return 0;
11 }
```

Результат:

What a string!

string::string(const char *szCString, size_type unLength)

- Конструктор, который создает новую строку из строки C-style `szCString` с количеством символов, указанных в `unLength`.
- Если размер результата превышает максимальную длину строки, то генерируется исключение `length_error`.

Предупреждение: Только для этого конструктора значение `NULL` не обрабатывается как объект, указывающий на завершение строки `szCString`! Это означает, что компилятор дойдет до конца строки (если это позволяет `unLength`), даже если встретит `NULL`.

Например:

```
1 #include <iostream>
2 #include <string>
```

```

3
4 int main()
5 {
6     const char *szSomething("What a string!");
7     std::string sOutput(szSomething, 7);
8     std::cout << sOutput << std::endl;
9
10    return 0;
11 }

```

Результат:

What a

string::string(size_type nNum, char chChar)

- Конструктор, который создает новую строку, инициализированную символом `chChar` и требуемым количеством вхождений этого символа (указывается в `nNum`).
- Если размер результата превышает максимальную длину строки, то генерируется исключение `length_error`.

Например:

```

1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sOutput(5, 'G');
7     std::cout << sOutput << std::endl;
8
9     return 0;
10 }

```


Результат:

GGGGG

template string::string(InputIterator itBeg, InputIterator itEnd)

- Конструктор, который создает новую строку, инициализированную символами диапазона `[itBeg, itEnd)`.
- Если размер результата превышает максимальную длину строки, то генерируется исключение `length_error`.

Здесь нет примера, так как вероятность того, что вы будете использовать этот конструктор, ничтожно мала.

string::~~string()

- Деструктор, который уничтожает строку и освобождает память.

Примера нет, так как деструктор вызывается неявно.

Создание std::string из чисел

Одно заметное упущение в классе `std::string` — это отсутствие возможности создавать строки из чисел. Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sFive(5);
7
8     return 0;
9 }
```

Здесь мы получим ошибку неудачной конвертации значения типа `int` в `std::basic_string`. Самый простой способ конвертировать числа в строки — это **задействовать класс `std::ostringstream`**, который находится в **заголовочном файле `sstream`**. `std::ostringstream` уже настроен для приема разных входных данных: символов, чисел, строк и т.д. А с помощью **`std::istringstream`** можно выполнять обратную конвертацию — выводить строки (либо через оператор вывода `>>`, либо через функцию `str()`).

Например, создадим `std::string` из разных входных данных:

```
1  #include <iostream>
2  #include <sstream>
3  #include <string>
4
5  template <typename T>
6  inline std::string ToString(T tX)
7  {
8      std::ostringstream oStream;
9      oStream << tX;
10     return oStream.str();
11 }
12
13 int main()
14 {
15     std::string sFive(ToString(5));
16     std::string sSevenPointEight(ToString(7.8));
17     std::string sB(ToString('B'));
18     std::cout << sFive << std::endl;
19     std::cout << sSevenPointEight << std::endl;
20     std::cout << sB << std::endl;
21 }
```

Результат:

5

7.8

В

Обратите внимание, здесь отсутствует проверка на ошибки. Может случиться так, что конвертация `tX` в `std::string` будет неудачной. В таком случае, хорошим вариантом было бы подключить генерацию исключения.

Конвертация `std::string` в числа

Аналогично вышеприведенному решению:

```
1 #include <iostream>
2 #include <sstream>
3 #include <string>
4
5 template <typename T>
6 inline bool FromString(const std::string& sString, T &tX)
7 {
8     std::istringstream iStream(sString);
9     return (iStream >> tX) ? true : false; // извлекаем значение в tX, возвращаем true (если удачно) или false (если
10 }
11
12 int main()
13 {
14     double dX;
15     if (FromString("4.5", dX))
16         std::cout << dX << std::endl;
17     if (FromString("TOM", dX))
18         std::cout << dX << std::endl;
19 }
```

Результат:

4.5

Обратите внимание, наша вторая конвертация потерпела неудачу, и мы получили `false`.

Оценить статью:

★★★★☆ (86 оценок, среднее: **4,91** из 5)

Поделиться в социальных сетях:



← Урок №200. Строковые классы `std::string` и `std::wstring`

Урок №202. Длина и ёмкость `std::string` →

Урок №202. Длина и ёмкость `std::string`



Юрий | [Уроки по C++](#) | Обновл. 15 Сен 2021 | 55791 | 6

При создании строки не помешало бы указать её длину и ёмкость (или хотя бы знать эти параметры).

Оглавление:

1. [Длина `std::string`](#)
2. [Ёмкость `std::string`](#)

Длина std::string

Длина строки — это количество символов, которые она содержит. Есть две идентичные функции для определения длины строки:

→ **size_type string::length() const**

→ **size_type string::size() const**

Обе эти функции возвращают текущее количество символов, которые содержит строка, исключая **нуль-терминатор**. Например:

```
1 #include <iostream>
2
3 int main()
4 {
5     std::string sSomething("012345");
6     std::cout << sSomething.length() << std::endl;
7
8     return 0;
9 }
```

Результат:

6

Хотя также можно использовать функцию `length()` для определения того, содержит ли строка какие-либо символы или нет, эффективнее использовать функцию `empty()`:

→ **bool string::empty() const** — эта функция возвращает `true`, если в строке нет символов, и `false` — в противном случае.

Например:

```
1 #include <iostream>
2
3 int main()
4 {
5     std::string sString1("Not Empty");
6     std::cout << (sString1.empty() ? "true" : "false") << std::endl;
7     std::string sString2; // пустая строка
8     std::cout << (sString2.empty() ? "true" : "false") << std::endl;
9
10    return 0;
11 }
```

Результат:

false

true

Есть еще одна функция, связанная с длиной строки, которую вы, вероятно, никогда не будете использовать, но мы все равно её рассмотрим:

→ **size_type string::max_size() const** — эта функция возвращает максимальное количество символов, которое может хранить строка. Это значение может варьироваться в зависимости от операционной системы и архитектуры операционной системы.

Например:

```
1 #include <iostream>
2
3 int main()
4 {
5     std::string sString("MyString");
```

```
6 std::cout << sString.max_size() << std::endl;
7 }
```

Результат:

```
2147483647
```

Ёмкость `std::string`

Ёмкость строки — это максимальный объем памяти, выделенный строке для хранения содержимого. Это значение измеряется в символах строки, исключая нуль-терминатор. Например, строка с ёмкостью 8 может содержать 8 символов.

→ **`size_type string::capacity() const`** — эта функция возвращает количество символов, которое может хранить строка без дополнительного перераспределения/перевыделения памяти.

Например:

```
1 #include <iostream>
2
3 int main()
4 {
5     std::string sString("0123456789");
6     std::cout << "Length: " << sString.length() << std::endl;
7     std::cout << "Capacity: " << sString.capacity() << std::endl;
8
9     return 0;
10 }
```

Результат:

Length: 10

Capacity: 15

Примечание: Запускать эту и следующие программы следует в полноценных IDE, а не в веб-компиляторах.

Обратите внимание, ёмкость строки больше её длины! Хотя длина нашей строки равна 10, памяти для неё выделено аж на 15 символов! Почему так?

Здесь важно понимать, что, если пользователь захочет поместить в строку больше символов, чем она может вместить, строка будет перераспределена и, соответственно, ёмкость будет больше. Например, если строка имеет длину и ёмкость равную 10, то добавление новых символов в строку приведет к её перераспределению. Делая ёмкость строки больше её длины, мы предоставляем пользователю некоторое буферное пространство для расширения строки (добавление новых символов).

Но в перераспределении есть также несколько нюансов:

- Во-первых, это сравнительно ресурсозатратно. Сначала должна быть выделена новая память. Затем каждый символ строки копируется в новую память. Если строка большая, то тратится много времени. Наконец, старая память должна быть удалена/освобождена. Если вы делаете много перераспределений, то этот процесс может значительно снизить производительность вашей программы.
- Во-вторых, всякий раз, когда строка перераспределяется, её содержимое получает новый адрес памяти. Это означает, что все текущие **ссылки**, **указатели** и **итераторы** строки становятся недействительными!

Обратите внимание, не всегда строки создаются с ёмкостью, превышающей её длину. Рассмотрим следующую программу:

```
1 #include <iostream>
2
3 int main()
```

```

4 {
5     std::string sString("0123456789abcde");
6     std::cout << "Length: " << sString.length() << std::endl;
7     std::cout << "Capacity: " << sString.capacity() << std::endl;
8
9     return 0;
10 }

```

Результат:

Length: 15

Capacity: 15

Примечание: Результаты могут отличаться в зависимости от компилятора.

Теперь давайте добавим еще один символ в конец строки и посмотрим на изменение её ёмкости:

```

1  #include <iostream>
2
3  int main()
4  {
5      std::string sString("0123456789abcde");
6      std::cout << "Length: " << sString.length() << std::endl;
7      std::cout << "Capacity: " << sString.capacity() << std::endl;
8
9      // Добавляем новый символ
10     sString += "f";
11     std::cout << "Length: " << sString.length() << std::endl;
12     std::cout << "Capacity: " << sString.capacity() << std::endl;
13
14     return 0;
15 }

```

Результат:

```
Length: 15  
Capacity: 15  
Length: 16  
Capacity: 31
```

Есть еще одна функция (а точнее 2 варианта этой функции) для работы с ёмкостью строки:

- **void string::reserve(size_type unSize)** — при вызове этой функции мы устанавливаем ёмкость строки, равную, как минимум, `unSize` (она может быть и больше). Обратите внимание, для выполнения этой функции может потребоваться перераспределение.
- **void string::reserve()** — если вызывается эта функция или вышеприведенная функция с `unSize` меньше текущей ёмкости, то компилятор попытается срезать (уменьшить) ёмкость строки до размера её длины. Это необязательный запрос.

Например:

```
1  #include <iostream>  
2  
3  int main()  
4  {  
5      std::string sString("0123456789");  
6      std::cout << "Length: " << sString.length() << std::endl;  
7      std::cout << "Capacity: " << sString.capacity() << std::endl;  
8  
9      sString.reserve(300);  
10     std::cout << "Length: " << sString.length() << std::endl;  
11     std::cout << "Capacity: " << sString.capacity() << std::endl;  
12  
13     sString.reserve();
```

```
14     std::cout << "Length: " << sString.length() << std::endl;
15     std::cout << "Capacity: " << sString.capacity() << std::endl;
16
17     return 0;
18 }
```

Результат:

```
Length: 10
Capacity: 15
Length: 10
Capacity: 303
Length: 10
Capacity: 303
```

Здесь мы можем наблюдать две интересные вещи. Во-первых, хотя мы запросили ёмкость равную 300, мы фактически получили 303. Ёмкость всегда будет не меньше, чем мы запрашиваем (но может быть и больше). Затем мы запрашиваем изменение ёмкости в соответствии со строкой. Этот запрос был проигнорирован, так как, очевидно, что ёмкость не изменилась.

Если вы заранее знаете, что вам нужна строка побольше, так как вы будете выполнять с ней множество операций, которые потенциально могут увеличить её длину или ёмкость, то вы можете избежать перераспределений, сразу установив строке её окончательную емкость:

```
1  #include <iostream>
2  #include <string>
3  #include <cstdlib> // для rand() и srand()
4  #include <ctime> // для time()
5
6  int main()
7  {
8      srand(time(0)); // генерация случайного числа
```

```
9
10 std::string sString; // длина 0
11 sString.reserve(80); // резервируем 80 символов
12
13 // Заполняем строку случайными строчными символами
14 for (int nCount = 0; nCount < 80; ++nCount)
15     sString += 'a' + rand() % 26;
16
17 std::cout << sString;
18 }
```

Результат этой программы будет меняться при каждом её новом запуске:

```
tregsxxmselsqlfoahsvsxfmfwurcmmjclfcqqgzkzohztirriibtoibucswaudyirkvjbwxfysoqzcc
```

Вместо того, чтобы перераспределять `sString` несколько раз, мы устанавливаем её ёмкость один раз, а затем просто заполняем данными.

Оценить статью:

★★★★★ (104 оценок, среднее: **4,94** из 5)

Поделиться в социальных сетях:



← Урок №201. Создание, уничтожение и конвертация std::string

Урок №203. Доступ к символам std::string. Конвертация std::string в строки C-style →

Комментариев: 6



Maks

2 мая 2021 в 23:38

Вместо `std::string::reserve()` для уменьшения ёмкости строки до размера её длины, лучше использовать `std::string::shrink_to_fit` (C++11)

Ответить



Влад

5 января 2021 в 12:55

Подскажите, допустим мне надо просто перебрать все элементы строки в цикле:

Урок №203. Доступ к символам `std::string`. Конвертация `std::string` в строки C-style



Юрий | [Уроки по C++](#) | Обновл. 15 Сен 2021 | 17050 | 2

На этом уроке мы рассмотрим способы доступа к символам `std::string` и способы конвертации `std::string` в **строки C-style**.

Оглавление:

1. [Доступ к символам `std::string`](#)

2. Конвертация std::string в строки C-style

Доступ к символам std::string

Есть два практически идентичных способа доступа к символам std::string. Наиболее простой и быстрый — использовать **перегруженный оператор индексации []**.

char& string::operator[](size_type nIndex)

const char& string::operator[](size_type nIndex) const

- Обе эти функции возвращают символ под индексом `nIndex`.
- Передача неверного индекса приведет к неопределенным результатам.
- Использование **функции `length()`** в качестве индекса допустимо только для константных строк и возвращает значение, сгенерированное **конструктором по умолчанию `std::string`**. Это не рекомендуется делать.
- Поскольку `char&` — это тип возврата, то вы можете использовать его для изменения символов строки.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sSomething("abcdefg");
7     std::cout << sSomething[4] << std::endl;
8     sSomething[4] = 'A';
```



```
9 std::cout << sSomething << std::endl;
10 }
```

Результат:

```
e
abcdAfg
```

Другой способ доступа к символам `std::string` медленнее, чем вышеприведенный вариант, так как использует **исключения** для проверки корректности `nIndex`. Если вы не уверены в корректности передаваемого `nIndex`, то вы должны использовать именно этот способ (тот, что описан ниже) для доступа к символам строки.

`char& string::at(size_type nIndex)`
`const char& string::at(size_type nIndex) const`

- Обе эти функции возвращают символ под индексом `nIndex`.
- Передача неверного индекса приведет к генерации исключения `out_of_range`.
- Поскольку `char&` — это тип возврата, то вы можете использовать его для изменения символов строки.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sSomething("abcdefg");
7     std::cout << sSomething.at(4) << std::endl;
8     sSomething.at(4) = 'A';
9     std::cout << sSomething << std::endl;
```

```
10 }
```

Результат:

```
е  
abcdAfg
```

Конвертация `std::string` в строки C-style

Многие функции (включая все функции языка C++) ожидают форматирования строк как строк C-style, а не как `std::string`. По этой причине `std::string` предоставляет 3 разных способа конвертации `std::string` в строки C-style.

`const char* string::c_str() const`

- Возвращает содержимое `std::string` в виде константной строки C-style.
- Добавляется нуль-терминатор.
- Строка C-style принадлежит `std::string` и не должна быть удалена.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sSomething("abcdefg");
7     std::cout << strlen(sSomething.c_str());
8 }
```

Результат:

const char* string::data() const

- Возвращает содержимое std::string в виде константной строки C-style.
- Не добавляется нуль-терминатор.
- Строка C-style принадлежит std::string и не должна быть удалена.

Например:

```
1  #include <iostream>
2  #include <string>
3
4  int main()
5  {
6      std::string sSomething("abcdefg");
7      const char *szString = "abcdefg";
8      // Функция memcmp() сравнивает две вышеприведенные строки C-style и возвращает 0, если они равны
9      if (memcmp(sSomething.data(), szString, sSomething.length()) == 0)
10         std::cout << "The strings are equal";
11     else
12         std::cout << "The strings are not equal";
13 }
```

Результат:

```
The strings are equal
```

size_type string::copy(char *szBuf, size_type nLength) const

size_type string::copy(char *szBuf, size_type nLength, size_type nIndex) const

- Отличие второго варианта этой функции от первого состоит в том, что копирование не более `nLength` символов передаваемой строки в `szBuf` начинается с символа под индексом `nIndex`. В первой же функции копирование всегда начинается с символа под индексом `[0]`.
- Количество скопированных символов возвращается.
- Caller отвечает за то, чтобы не произошло **переполнения** строки `szBuf`.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sSomething("lorem ipsum dolor sit amet");
7
8     char szBuf[20];
9     int nLength = sSomething.copy(szBuf, 5, 6);
10    szBuf[nLength] = '\0'; // завершаем строку в буфере
11
12    std::cout << szBuf << std::endl;
13 }
```

Результат:

`ipsum`

Если вы не гонитесь за максимальной эффективностью, то `c_str()` — это самый простой и безопасный способ конвертации `std::string` в строки C-style.

Оценить статью:

★★★★★ (73 оценок, среднее: 4,85 из 5)

Поделиться в социальных сетях:



← Урок №202. Длина и ёмкость `std::string`

Урок №204. Присваивание и перестановка значений с `std::string` →

Комментариев: 2



rrrrr

21 января 2019 в 14:41

в крайнем примере не помешает

Урок №204. Присваивание и перестановка значений с `std::string`



Юрий | [Уроки по C++](#) | Обновл. 15 Сен 2021 | 13456 | 2

На этом уроке мы рассмотрим операции присваивания других значений для `std::string` и перестановку значений двух строк.

Оглавление:

1. [Присваивание для `std::string` других значений](#)
2. [Перестановка значений двух строк](#)

Присваивание для `std::string` других значений

Самый простой способ присвоить `std::string` другое значение — использовать **перегруженный оператор присваивания** `=`. Или, в качестве альтернативы, **метод** `assign()`.

`string& string::operator=(const string& str)`

`string& string::assign(const string& str)`

`string& string::operator=(const char* str)`

`string& string::assign(const char* str)`

`string& string::operator=(char c)`

- Эти функции позволяют присваивать `std::string` значения разных типов.
- Они возвращают **скрытый указатель `*this`**, что позволяет «связывать» объекты.
- Обратите внимание, функции `assign()`, которая бы принимала один символ, нет.

Например:

```
1  #include <iostream>
2  #include <string>
3
4  int main()
5  {
6      using namespace std;
7
8      string sString;
9
10     // Присваиваем std::string другую строку
11     sString = string("One");
```

```
12 cout << sString << endl;
13
14 const string sTwo("Two");
15 sString.assign(sTwo);
16 cout << sString << endl;
17
18 // Присваиваем std::string строку C-style
19 sString = "Three";
20 cout << sString << endl;
21
22 sString.assign("Four");
23 cout << sString << endl;
24
25 // Присваиваем std::string значение типа char
26 sString = '5';
27 cout << sString << endl;
28
29 // Связываем объекты
30 string sOther;
31 sString = sOther = "Six";
32 cout << sString << " " << sOther << endl;
33
34 return 0;
35 }
```

Результат:

One
Two
Three
Four

5

Six Six

Метод `assign()` также имеет несколько разновидностей.

`string& string::assign(const string& str, size_type index, size_type len)`

- Эта функция присваивает `std::string` подстроку `str` длиной `len`, начиная с `index`-а.
- **Генерирует исключение** `out_of_range`, если `index` недопустим.
- Возвращает скрытый указатель `*this`, что позволяет «связывать» объекты.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     const std::string sSomething("abcdefgh");
7     std::string sDest;
8
9     sDest.assign(sSomething, 3, 5); // присваиваем sDest подстроку sSomething длиной 5, начиная с индекса 3
10    std::cout << sDest << std::endl;
11
12    return 0;
13 }
```

Результат:

defgh

string& string::assign(const char* chars, size_type len)

- Эта функция присваивает std::string **строку C-style** длиной `len`.
- Генерирует исключение `length_error`, если результат превышает максимальное количество символов.
- Возвращает скрытый указатель `*this`, что позволяет «связывать» объекты.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sDest;
7
8     sDest.assign("abcdefgh", 5);
9     std::cout << sDest << std::endl;
10
11     return 0;
12 }
```

Результат:

abcde

Эта функция потенциально опасна, поэтому использовать её не рекомендуется.

string& string::assign(size_type len, char c)

- Эта функция присваивает std::string определенное количество вхождений символа `c`. Количество вхождений указывается в `len`.

- Генерирует исключение `length_error`, если результат превышает максимальное количество символов.
- Возвращает скрытый указатель `*this`, что позволяет «связывать» объекты.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sDest;
7
8     sDest.assign(5, 'h');
9     std::cout << sDest << std::endl;
10
11     return 0;
12 }
```

Результат:

```
hhhhh
```

Перестановка значений двух строк

Если у вас есть две строки, значения которых вы хотите поменять местами, используйте **функцию `swap()`**.

`void string::swap(string &str)`

`void swap(string &str1, string &str2)`

- Обе функции меняют местами значения двух строк. Первый вариант функции `swap()` меняет местами значения `*this` и `str`, а второй — `str1` и `str2`.
- Используйте данные функции вместо операции присваивания, если нужно поменять местами значения двух строк.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 using namespace std;
5
6 int main()
7 {
8     string sStr1("green");
9     string sStr2("white");
10
11     cout << sStr1 << " " << sStr2 << endl;
12     swap(sStr1, sStr2);
13     cout << sStr1 << " " << sStr2 << endl;
14     sStr1.swap(sStr2);
15     cout << sStr1 << " " << sStr2 << endl;
16
17     return 0;
18 }
```

Результат:

```
green white
white green
green white
```

На следующем уроке мы рассмотрим добавление значений к `std::string`.

Оценить статью:

★★★★★ (84 оценок, среднее: **4,90** из 5)

Поделиться в социальных сетях:



← Урок №203. Доступ к символам `std::string`. Конвертация `std::string` в строки C-style

Урок №205. Добавление к `std::string` →

Комментариев: 2

Steindvart

Урок №205. Добавление к `std::string`



Юрий | [Уроки по C++](#) | Обновл. 15 Сен 2021 | 21497 | 5

Чтобы добавить одну **строку** к другой строке, можно использовать **перегруженный** оператор `+=`, функцию `append()` или функцию `push_back()`.

`string& string::operator+=(const string& str)`

`string& string::append(const string& str)`

- Обе функции добавляют к `std::string` строку `str`.
- Возвращают **скрытый указатель `*this`**, что позволяет «связывать» объекты.

→ **Генерируют исключение** `length_error`, если результат превышает максимально допустимое количество СИМВОЛОВ.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sString("one");
7
8     sString += std::string(" two");
9
10    std::string sThree(" three");
11    sString.append(sThree);
12
13    std::cout << sString << std::endl;
14 }
```

Результат:

```
one two three
```

Существует также разновидность функции `append()`, которая может добавлять подстроку.

`string& string::append(const string& str, size_type index, size_type num)`

→ Эта функция добавляет к `std::string` строку `str` с количеством символов, которые указываются в `num`, начиная с `index`-а.

→ Возвращает скрытый указатель `*this`, что позволяет «связывать» объекты.

- Генерирует исключение `out_of_range`, если `index` некорректен.
- Генерирует исключение `length_error`, если результат превышает максимально допустимое количество символов.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sString("one ");
7
8     const std::string sTemp("twothreefour");
9     sString.append(sTemp, 8, 4); // добавляем к std::string подстроку sTemp длиной 4, начиная с символа под индексом
10    std::cout << sString << std::endl;
11 }
```

Результат:

one four

Оператор `+=` и функция `append()` также имеют версии, которые работают со **строками C-style**.

`string& string::operator+=(const char* str)`

`string& string::append(const char* str)`

- Обе функции добавляют к `std::string` строку C-style `str`.
- Возвращают скрытый указатель `*this`, что позволяет «связывать» объекты.

→ Генерируют исключение `length_error`, если результат превышает максимально допустимое количество символов.

→ `str` не должен быть `NULL`.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sString("one");
7
8     sString += " two";
9     sString.append(" three");
10    std::cout << sString << std::endl;
11 }
```

Результат:

```
one two three
```

И есть еще одна разновидность функции `append()`, которая работает со строками C-style.

`string& string::append(const char* str, size_type len)`

→ Добавляет к `std::string` количество символов (которые указаны в `len`) строки C-style `str`.

→ Возвращает скрытый указатель `*this`, что позволяет «связывать» объекты.

- Генерирует исключение `length_error`, если результат превышает максимально допустимое количество символов.
- Игнорирует специальные символы (включая `"`).

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sString("two ");
7
8     sString.append("fivesix", 4);
9     std::cout << sString << std::endl;
10 }
```

Результат:

```
two five
```

Эта функция опасна, поэтому использовать её не рекомендуется. Существуют также функции, которые добавляют отдельные (единичные) символы.

`string& string::operator+=(char c)`
`void string::push_back(char c)`

- Обе функции добавляют к `std::string` символ `c`.
- Оператор `+=` возвращает скрытый указатель `*this`, что позволяет «связывать» объекты.

- Обе функции генерируют исключение `length_error`, если результат превышает максимально допустимое количество символов.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sString("two");
7
8     sString += ' ';
9     sString.push_back('3');
10    std::cout << sString << std::endl;
11 }
```

Результат:

```
two 3
```

`string& string::append(size_type num, char c)`

- Эта функция добавляет к `std::string` количество вхождений (которые указываются в `num`) символа `c`.
- Возвращает скрытый указатель `*this`, что позволяет «связывать» объекты.
- Генерирует исключение `length_error`, если результат превышает максимально допустимое количество символов.

Например:

```
1 #include <iostream>
```

```
2 #include <string>
3
4 int main()
5 {
6     std::string sString("eee");
7
8     sString.append(5, 'f');
9     std::cout << sString << std::endl;
10 }
```

Результат:

eeefffff


Есть еще одна (последняя) вариация функции `append()`, использование которой вы не поймете, если не знакомы с **итераторами**.

`string& string::append(InputIterator start, InputIterator end)`

- Эта функция добавляет к `std::string` все символы из диапазона `(start, end)`.
- Возвращает скрытый указатель `*this`, что позволяет «связывать» объекты.
- Генерирует исключение `length_error`, если результат превышает максимально допустимое количество символов.

На следующем уроке мы рассмотрим вставку символов в `std::string`.

Оценить статью:

 (75 оценок, среднее: **4,88** из 5)

Поделиться в социальных сетях:



← Урок №204. Присваивание и перестановка значений с `std::string`

Урок №206. Вставка символов и строк в `std::string` →

Комментариев: 5



Анастасия

7 ноября 2019 в 17:38

скукотища

Ответить

kmish

Урок №206. Вставка символов и строк в `std::string`



Юрий | [Уроки по C++](#) | Обновл. 15 Сен 2021 | 31335 | 2

Вставлять символы/строки в `std::string` можно с помощью функции `insert()`.

`string& string::insert(size_type index, const string& str)`

`string& string::insert(size_type index, const char* str)`

- Обе функции вставляют символы/строки, начиная с определенного `index` `std::string`.
- Возвращают **скрытый указатель `*this`**, что позволяет «связывать» объекты.
- **Генерируют исключение** `out_of_range`, если `index` некорректен.

→ Генерируют исключение `length_error`, если результат превышает максимально допустимое количество символов.

→ Во второй версии функции `insert()` `str` не должен быть `NULL`.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sString("bbb");
7     std::cout << sString << std::endl;
8
9     sString.insert(2, std::string("mmm"));
10    std::cout << sString << std::endl;
11
12    sString.insert(5, "aaa");
13    std::cout << sString << std::endl;
14 }
```

Результат:

bbb

bbmmmb

bbmmaaab

А вот версия функции `insert()`, которая позволяет вставить с определенного `index` `std::string` подстроку.

`string& string::insert(size_type index, const string& str, size_type startindex, size_type num)`

- Эта функция вставляет с определенного `index` `std::string` указанное количество символов (`num`) строки `str`, начиная со `startindex`-а.
- Возвращает скрытый указатель `*this`, что позволяет «связывать» объекты.
- Генерирует исключение `out_of_range`, если `index` или `startindex` некорректны.
- Генерирует исключение `length_error`, если результат превышает максимально допустимое количество символов.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sString("bbb");
7
8     const std::string sInsert("012345");
9     sString.insert(1, sInsert, 2, 4); // вставляем подстроку sInsert длиной 4, начиная с символа под индексом 2, в с
10    std::cout << sString << std::endl;
11 }
```

Результат:

b2345bb

А вот версия функции `insert()`, с помощью которой в `std::string` можно вставить часть **строки C-style**.

`string& string::insert(size_type index, const char* str, size_type len)`

- Эта функция вставляет с определенного `index` `std::string` указанное количество символов (`len`) строки C-style `str`.
- Возвращает скрытый указатель `*this`, что позволяет «связывать» объекты.
- Генерирует исключение `out_of_range`, если `index` некорректен.
- Генерирует исключение `length_error`, если результат превышает максимально допустимое количество символов.
- Игнорирует специальные символы (такие как `"`).

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sString("bbb");
7
8     sString.insert(2, "acdef", 4);
9     std::cout << sString << std::endl;
10 }
```

Результат:

bbacdeb

А вот версия функции `insert()`, которая вставляет в `std::string` один и тот же символ несколько раз.

`string& string::insert(size_type index, size_type num, char c)`

- Эта функция вставляет с определенного `index` `std::string` указанное количество вхождений (`num`) символа `c` .
- Возвращает скрытый указатель `*this`, что позволяет «связывать» объекты.
- Генерирует исключение `out_of_range`, если `index` некорректен.
- Генерирует исключение `length_error`, если результат превышает максимально допустимое количество символов.

Например:

```
1 #include <iostream>
2 #include <string>
3
4 int main()
5 {
6     std::string sString("bbb");
7
8     sString.insert(2, 3, 'a');
9     std::cout << sString << std::endl;
10 }
```

Результат:

bbaaab

И, наконец, функция `insert()` имеет три разные версии, которые работают с **итераторами**.

`void insert(iterator it, size_type num, char c)`

`iterator string::insert(iterator it, char c)`

`void string::insert(iterator it, InputIterator begin, InputIterator end)`

- Первая версия функции вставляет в `std::string` указанное количество вхождений (`num`) символа `c` перед итератором `it`.
- Вторая версия функции вставляет в `std::string` одиночный символ `c` перед итератором `it` и возвращает итератор в позицию вставленного символа.
- Третья версия функции вставляет в `std::string` все символы диапазона (`begin, end`) перед итератором `it`.
- Все функции генерируют исключение `length_error`, если результат превышает максимально допустимое количество символов.

Оценить статью:

★★★★★ (85 оценок, среднее: **4,95** из 5)

Поделиться в социальных сетях:



← Урок №205. Добавление к std::string

Урок №207. Потоки ввода и вывода →

Комментариев: 2

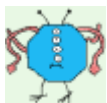


DarkMatter

23 августа 2021 в 10:19

Самая сухая глава из всех, к сожалению, которая заменяется беглым взглядом в документацию

Ответить



kmish

26 марта 2019 в 13:17

Пример функций

```
1 void insert(iterator it, size_type num, char c)
```