Arjan van Leeuwen
@avl7771

Opera

Dealing with Strings in C++

The Story of RSS and the StringHash

“When I check for new feed items, Opera *almost* crashes”

An RSS feed

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="2.0">
  <channel>
    <title>Example RSS Feed</title>
    <item>
      <title>Example Item</title>
      <description>A summary.</description>
      <link>http://www.example.com/foo</link>
      <pubDate>Mon, 23 Sep 2013 03:00:05 GMT
      </pubDate>
    </item>
  </channel>
</rss>
```


How long can it get?

Strings can sometimes get much longer than you expect, especially if it's some kind of user input being processed (which strings often are). We tend to forget strings have a variable length, and that the length of the string has an important effect on any operation done on it. Longer strings lead to all kinds of problems: slowness, unexpected results when

Many operations O(N)

Operations

clear	clears the contents (public member function)
insert	inserts characters (public member function)
erase	removes characters (public member function)
push_back	appends a character to the end (public member function)
pop_back (C++11)	removes the last character (public member function)
append	appends characters to the end (public member function)
operator+=	appends characters to the end (public member function)
compare	compares two strings (public member function)
replace	replaces specified portion of a string (public member function)
substr	returns a substring (public member function)
copy	copies characters (public member function)
resize	changes the number of characters stored (public member function)
swap	swaps the contents (public member function)

Search

find	find characters in the string (public member function)
rfind	find the last occurrence of a substring (public member function)
find_first_of	find first occurrence of characters (public member function)
find_first_not_of	find first absence of characters (public member function)
find_last_of	find last occurrence of characters (public member function)
find_last_not_of	find last absence of characters (public member function)

Do you actually need to use a string?

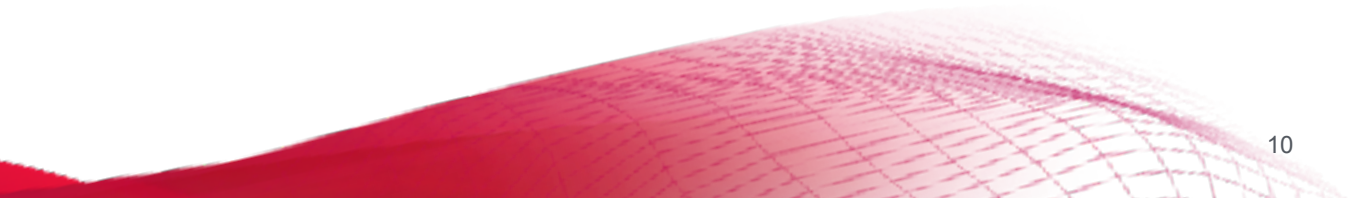
Mutable vs. immutable strings

Mutable / Immutable

Language	String type	Mutable	Notes
C	`None / char[]`	Mutable	Change in length hard
C++	`std::string`	Mutable	
Objective-C	`NSString`	Immutable	`NSMutableString` available
Python	`String`	Immutable	
JavaScript	`String`	Immutable	

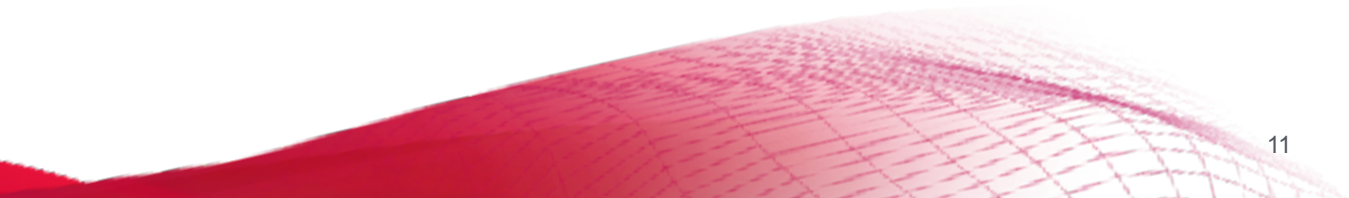
Mutable vs. immutable strings

- Changes that change length are expensive
- Many strings never need to be changed
- Immutable views of strings and substrings can replace mutable strings in many common cases
  - e.g. tokenization, splitting, trimming, …



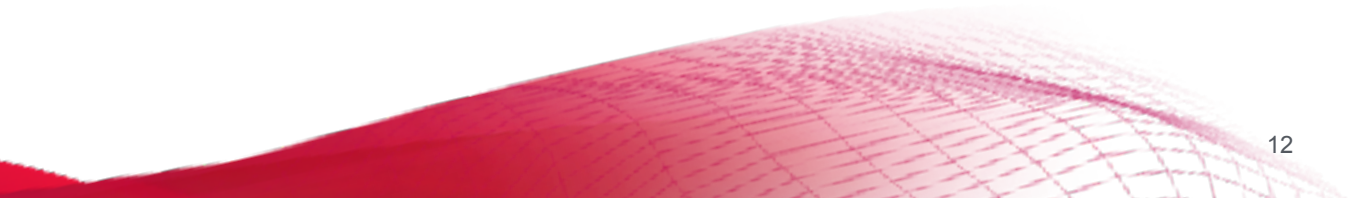
Problems with `char` arrays

- Is that a string or an array of 8-bit integers?
- `strlen()` is O(N)
- `NULL`-terminated
- Unsafe (as any C array)



The case for `std::string`

- No memory ownership issues
- Clear use case
- String length known
- Safe string utility functions
- Operators increase readability



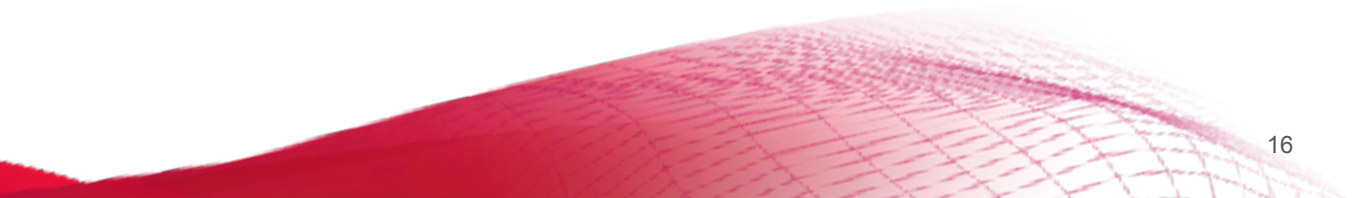
You Wouldn't Copy An Array

ASSIGNING STRINGS IS COPYING

COPYING IS AGAINST THE LAW!

Problems with `std::string`

- Always allocates memory
- Makes copying invisible with assignment and copy-constructor
- String literals are always copied
- Any substring creates another copy
- Mutable type



Does `std::string` make people lazy?

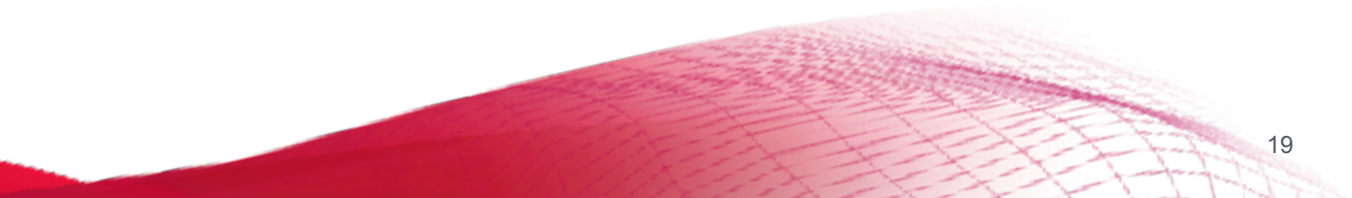
Or: Don't tell anybody, but I actually liked `char` arrays

Enter `std::string_view`

- ‘view on a string’: a non-owning string type
- String length known, safe helper functions
- Can be used with `std::string` and `char` arrays
- Copying to `std::string` is possible but explicit
- Real string literals!
- View can be a substring

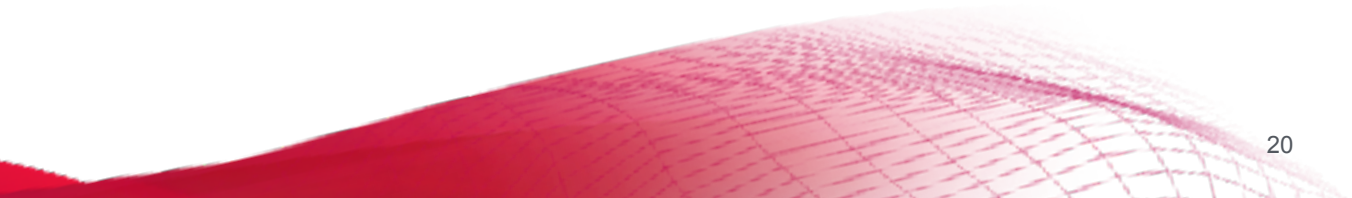
String types

	Type	Alloc	Copy	View
No STL	`char[]`	Heap, stack	`strdup`, `str{n/l}cpy`	`const char*`
C++03	`std::string`	Heap	`std::string`	`const std::string&`
C++17	`std::string_view`	Non-owning	`std::string`	`std::string_view`



`std::string_view` everywhere!

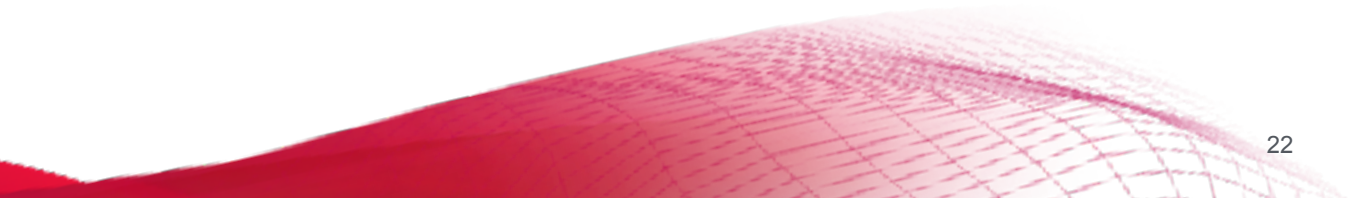
- use a `std::string_view` wherever you can!
- When you're working with `std::strings`, take care of ownership to prevent copy
  - e.g. use `std::move` when content not needed



Strings at compile-time

Growing strings

- Be careful with modifying operations such as append()
- Avoid creating a string out of many parts, better to create at once (remember immutability)



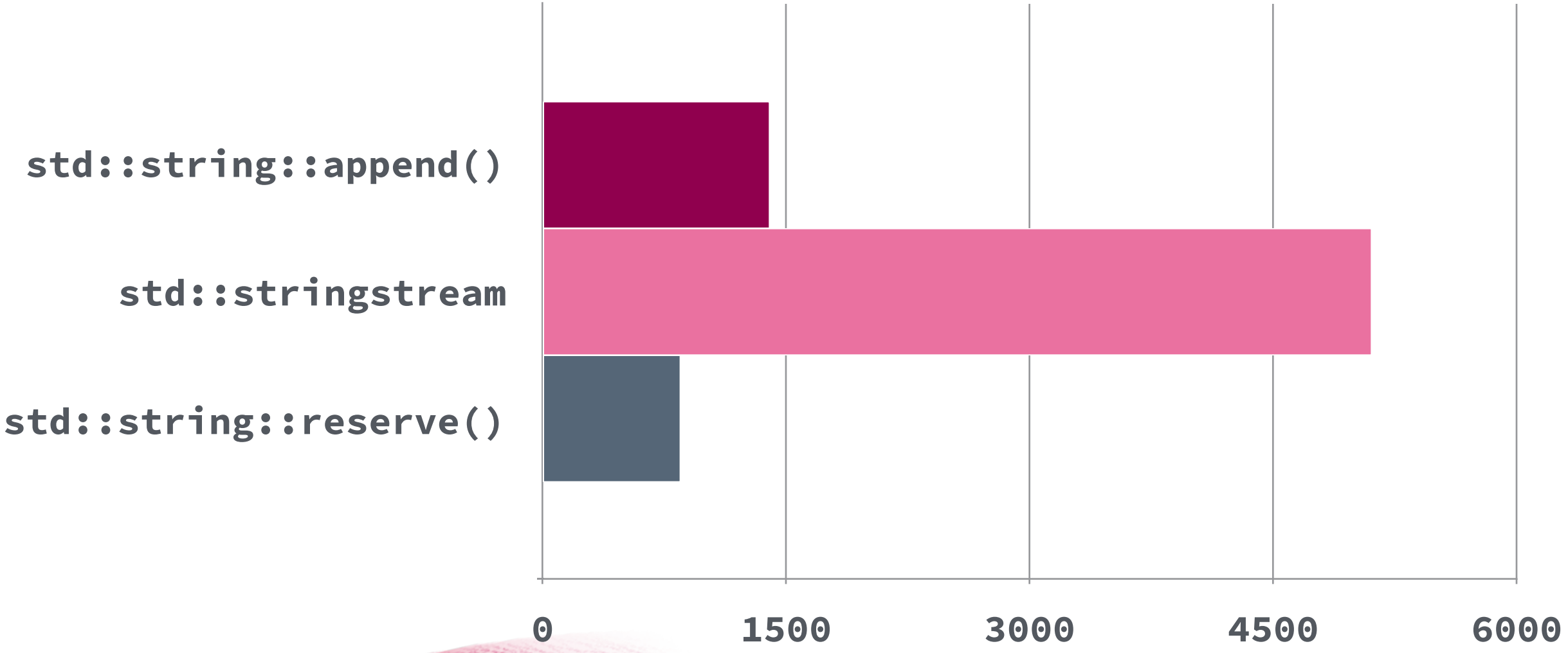
Growing strings

```
std::string CopyString(
    const char* to_copy, size_t length) {
  std::string copied;

  for (size_t i = 0; i < length; i += BLOCKSIZE)
    copied.append(to_copy + i,
        std::min(BLOCKSIZE, length - i));

  return copied;
}
```


Growing strings

```
std::string CopyString(
    const char* to_copy, size_t length) {
  std::stringstream copied;

  for (size_t i = 0; i < length; i += BLOCKSIZE)
    copied.write(to_copy + i,
        std::min(BLOCKSIZE, length - i));

  return copied.str();
}
```


Growing strings



Converting numbers

When you need to switch between two worlds

Converting to string

```
std::string Convert(int i) {
  std::stringstream stream;
  stream << i;
  return stream.str();
}
```

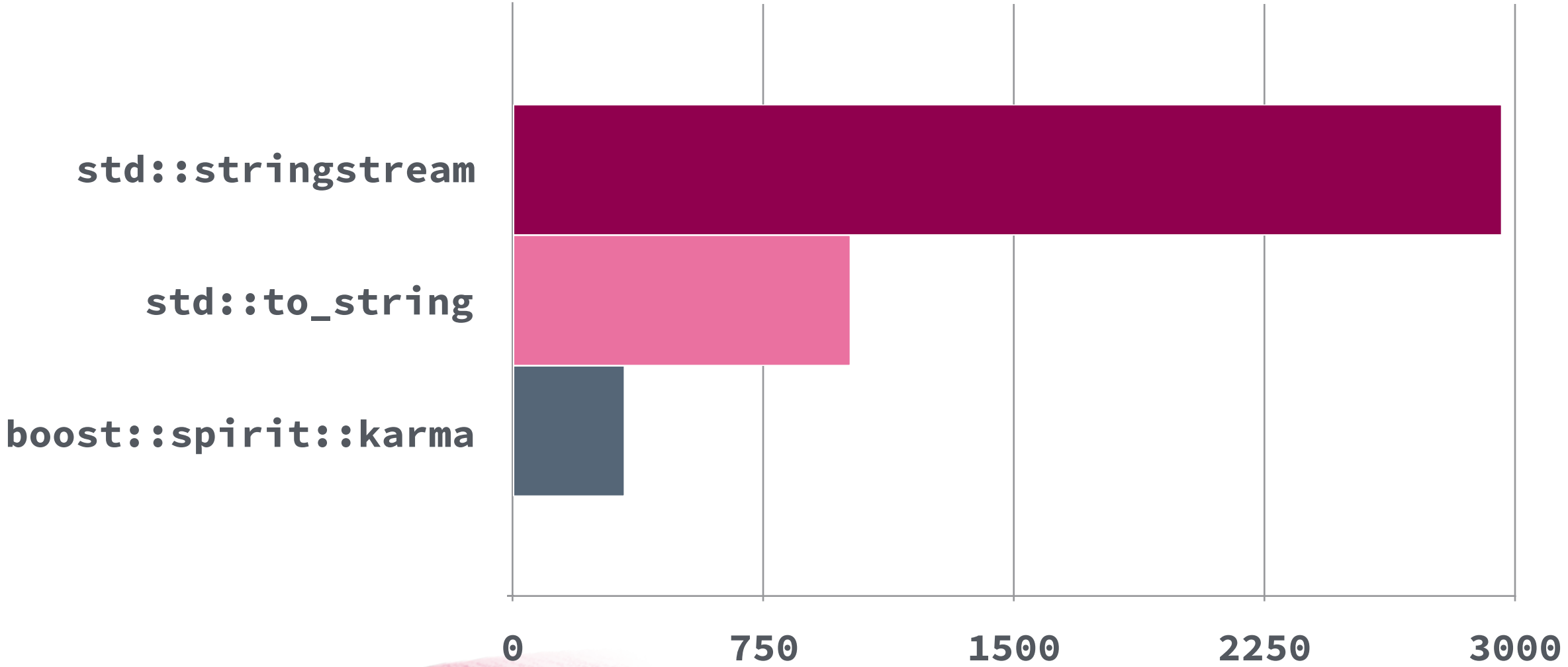

Converting to string

```
std::string Convert(int i) {
  return std::to_string(i);
}
```


Converting to string

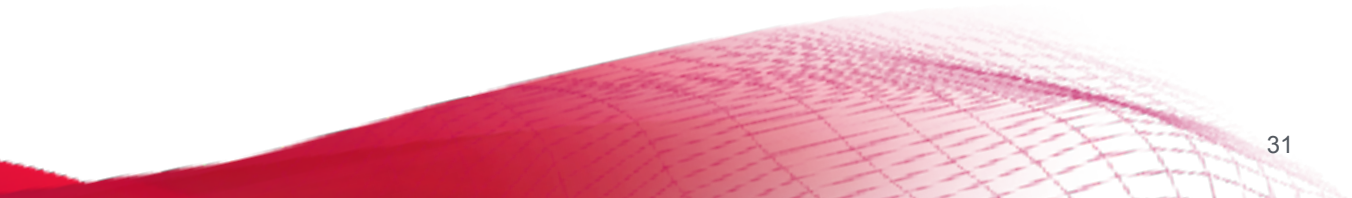
```
std::string Convert(int i) {
  namespace karma = boost::spirit::karma;
  std::string converted;
  std::back_insert_iterator<std::string>
    sink(converted);

  karma::generate(sink, karma::int_, i);
  return converted;
}
```


Converting to string



Converting to integer

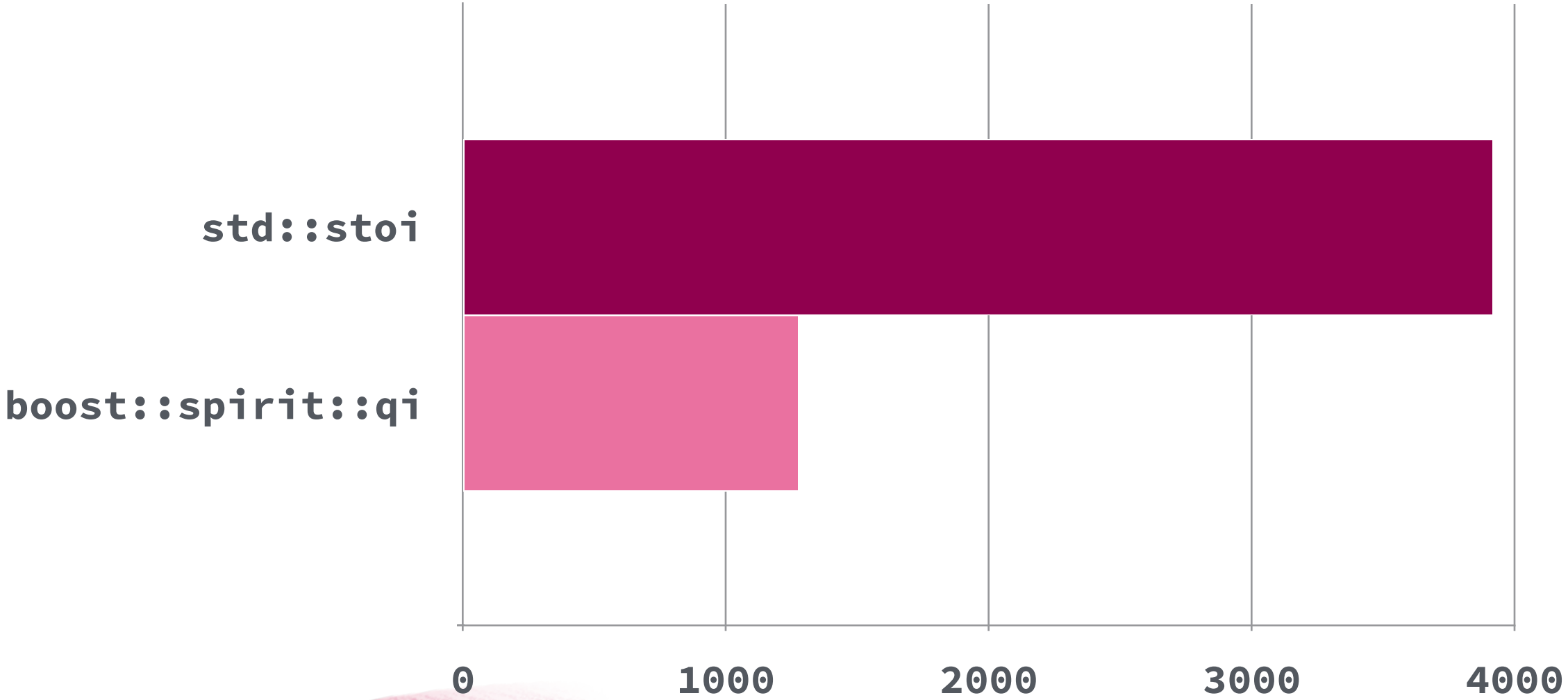
```
int Convert(const std::string& str) {
  return std::stoi(str);
}
```

Converting to integer

```
int Convert(const std::string& str) {
  namespace qi = boost::spirit::qi;
  int converted;

  qi::parse(str.begin(), str.end(),
      qi::int_, converted);
  return converted;
}
```

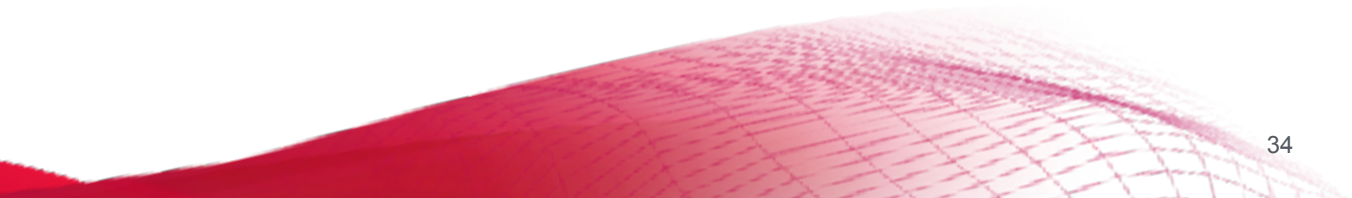

Converting to integer



Other options

`std::from_chars` *and* `std::to_chars` *are coming!*

low-level, locale-independent functions for conversions between integers and strings and between floating-point numbers and strings.



Regular Expressions

why the hotel really didn't have a pool

Avoid Regular Expressions

if you can

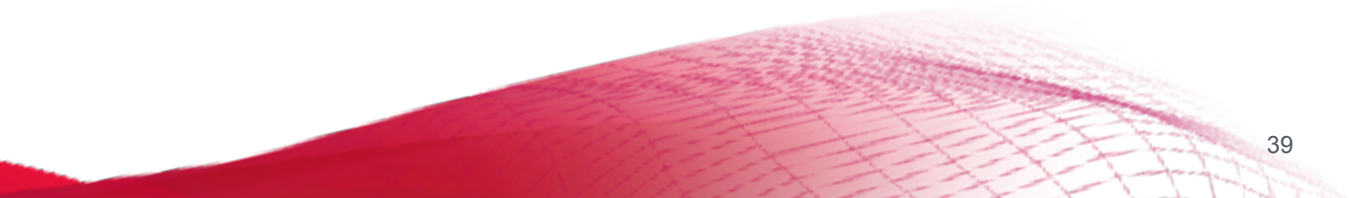
Character encoding

Basics of character encoding

- Language uses words or sentences made of characters
  - Latin ‘á’, Chinese ‘請’ are examples of characters
- Code points are numbers assigned to characters
- Encoding specifies how to represent code points in bytes

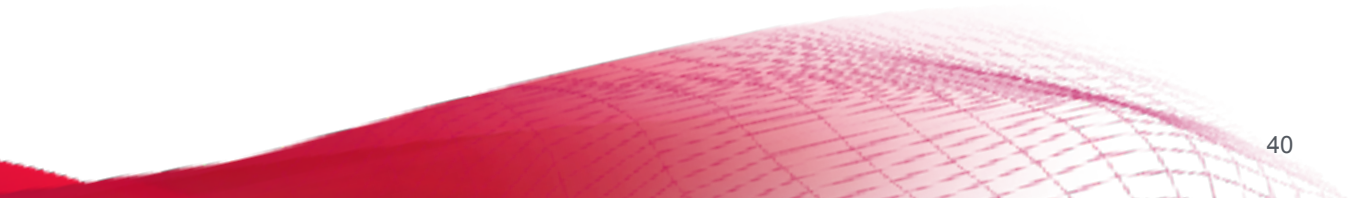
The Unicode standard

- Unicode specifies standard code points for many characters
  - Regularly updated to add emoji
- Unicode specifies several encodings that map all code points to byte sequences, called the Unicode Transformation Format (UTF)



The Unicode standard

- Unicode specifies standard code points for many characters
  - Regularly updated to add emoji 💩
- Unicode specifies several encodings that map all code points to byte sequences, called the Unicode Transformation Format (UTF)

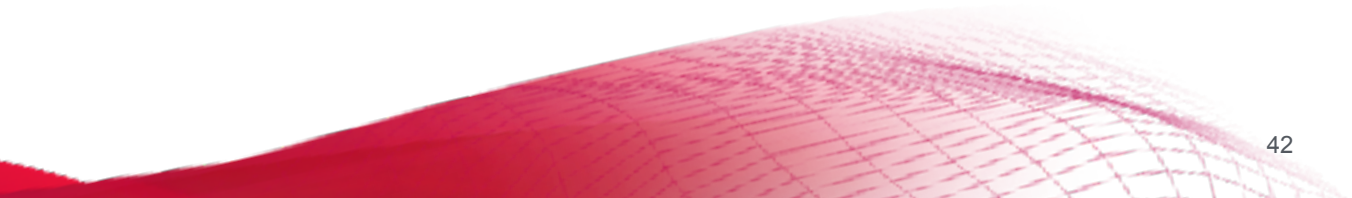


UTF encodings

	Size	Width per code point	Remarks
UTF-8	8 bit	Variable	Backwards compatible with ASCII
UTF-16	16 bit	Variable	Used in Windows system libraries, endianness matters
UTF-32	32 bit	Fixed	Used on Linux/BSD system libraries, endianness matters

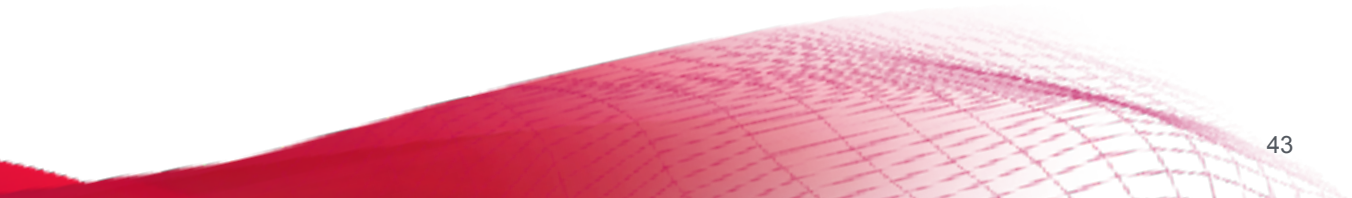
Why you should use UTF-8

- Backwards compatibility with ASCII is useful (no need to write separate functions)
- Uses less memory for ASCII-based data formats
- Uses more memory per code point for characters that actually represent more content
- No endianness concerns means compatibility



When you don’t use UTF-8

- Interacting continuously with APIs that require other encodings (e.g. UTF-16 in the Windows API)
- Can still use UTF-8 transcoding when saving files or doing network traffic
- Avoid transcoding as much as possible



UTF encodings in C++

	Primitive type	String type	String view
UTF-8	`char`	`std::string`	`std::string_view`
UTF-16	`char16_t`	`std::u16string`	`std::u16string_view`
UTF-32	`char32_t`	`std::u32string`	`std::u32string_view`

UTF encodings in C++

- Transcoding between different UTF variants is possible with the standard library but messy (look up `<codecvt>` for details)
- String literals can specify encoding and use non-ascii characters using code point specifiers:

```
u8"A 4 digit code point in UTF-8: \u2018."
u”An 8 digit code point in UTF-16: \U00002018."
U”An 8 digit code point in UTF-32: \U00002018."
```


Dictionaries and indexes

Arjan van Leeuwen
@avl7771

Opera

Dealing with Strings in C++