A Container Pretty-Printer for C++

Louis “Kerrek SB” Delacroix et al.[*]

Abstract

We present a header-only library for C++ which enables formatted output for arbitrary containers in the usual style like `std::cout << x << std::endl;`. The only requirements on the type `T` of `x` is that it have a member type `T::const_iterator` and methods `T::begin()` and `T::end()`.

Introduction

Templated containers are fundamental in C++ and occur ubiquitously. Yet producing formatted output of containers requires either a loop ranging over the container’s elements, or an overload of `operator<<` for the desired container. One frequently finds oneself writing the same pattern of code time and time again. It would be handy if printing a container would “just work”. This is what the `prettyprint` library achieves.

Expectations. This library was designed to meet the following generic expectations for formatted output of a container:

- The empty container should print as “`[]`”.
- A container with one element should print as “`[3]`”.
- A container with n elements should print as “`[4, -1, 11, 8]`”.
- The container’s elements should be printed by recursively invoking formatted output.

Formatted container output in this library has three parameters: The opening delimiter, the closing delimiter, and the separator. All three parameters should be customizable, but sensible defaults should be provided so that formatted output works without any action on the user’s part. We chose to select the following defaults:

- The separator is always “`, `”.
- Pairs and tuples appear in round parentheses: “`(a, b)`”.
- Sets and (set-likes) appear in curly braces: “`{3, -7, 18}`”.
- All other containers appear in square brackets: “`[2, 2, 0, 1]`”.

These defaults are inspired by the prevailing typographic tradition of mathematical notation, and we hope that they are both intuitive and æsthetically pleasing.

Note that key-value containers are not treated specially, since their value type is simply `std::pair<key_type, mapped_type>`.

[*]See the “Acknowledgements” for a full list of contributors,

Acknowledgements

Credits go to Marcelo Cantos for the initial approach, to Sven Groot for an improved, self-contained solution which became the foundation for this code, and to StackOverflow's Xeo for the tuple-printing code. This library would not have been possible without the support of these people.

Installation and Usage

Simply make the header file `prettyprint.hpp` available and include it in your code. Containers can be printed immediately:

```
#include <vector>
#include <map>
#include <set>
#include <iostream>
#include "prettyprint.hpp"

int main()
{
  std::vector<int>        v;
  std::set<double>        s;
  std::map<size_t, void*> m;
  int                     a[10];

  /* populate containers */

  std::cout << "Vector:  " << v << std::endl
            << "Set:     " << s << std::endl
            << "Map:     " << m << std::endl
            << "C-Array: " << pretty_print::array_wrapper(a) << std::endl;
}
```

This will output something like the following:

```
Vector:  [1, 2, 3]
Set:     {4.2, -1.3, NaN}
Map:     [(15, CD01EF23), (9, 19283ABC), (25, D2A92158)]
C-Array: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

C-style arrays. In the usage example above we introduced a helper function called `pretty_print::array_wrapper` which can be used to print C-style arrays in the same fashion as other containers. It comes in two overloads, one for compile-time sized arrays and one for dynamically sized arrays:

```
size_t n = get_number(), m = get_number();
int a[] = {1, 2, 4, 8}; // stack array of static size 4
int * b = new int[n];   // heap array of dynamic size n
int c[m];               // stack array of dynamic size m (VLA)
```

```
std::cout << pretty_print::array_wrapper(a)    << std::endl  // 1
          << pretty_print::array_wrapper(b, n) << std::endl  // 2
          << pretty_print::array_wrapper(c, m) << std::endl  // 2
;
```

In case (1) the array size is deduced at compile time; in cases (2) and (3) the array size has to be passed explicitly to the wrapper function.

Advanced Features: Customizing Delimiters

Note. All interna of the library live in the namespace `pretty_print`.

The only advanced feature of the library at this point is the ability to customize the delimiters used during printing. There are two ways to do this:

- Create a (partial or explicit) specialisation of a helper class for a specific container type and use the usual output syntax.
- Create a helper class holding custom delimiters and invoking a modified, slightly more verbose output function.

The two approaches meet orthogonal needs: If you have a container that is used repeatedly and which always must appear with custom delimiters, use the first approach and specialize the `delimiters` struct for your container type. On the other hand, if you have a set of delimiters that you only want to use in certain situations (but with any sort of container), create a helper class to hold your delimiters and invoke output with the `custom_delims` function like this:

```
std::cout << pretty_print::custom_delims<MyDelims>(x) << std::endl;
```

Method 1: Specialize.

The default delimiter class is:

```
template <typename T, typename TChar> struct pretty_print::delimiters;
```

All delimiter classes are templated on a character type `TChar`, which should match the desired output stream parameter (i.e. `char` for `ostream` and `wchar_t` for `wostream`), and on the container type `T`. The class has a static member constant `values` of type `delimiters_values<T, TChar>`, which itself is defined like this:

```
template<typename TChar> struct delimiters_values
{
    typedef TChar char_type;
    const TChar * prefix;
    const TChar * delimiter;
    const TChar * postfix;
};
```

By providing an explicit or partial specialization of `delimiters`, the delimiter parameters can be changed globally for a specific container type.

Example 1. We specialize for `std::vector<double>` to print "`((1.2; -3.4; 5.6))`".

```
template <> const pretty_print::delimiters_values<char>
pretty_print::delimiters<std::vector<double>, char>::values = { "((", "; ", "))" };
```

Example 2. We specialize for all `std::list<T>` to print "`a--b--c`".

```
template <typename T> const pretty_print::delimiters_values<char>
pretty_print::delimiters<std::list<T>, char>::values = { "", "--", "" };
```

Method 2: Custom delimiter class.

Instead of providing specializations for the default delimiter class, we can also create our own delimiter class. It must provide a static member constant `values` just like the default class. For example:

```
struct HalfOpen
{
  static const pretty_print::delimiters_values<wchar_t> values;
};
const pretty_print::delimiters_values<wchar_t>
HalfOpen::values = { L"[", L", L", ")" };

std::wcout << pretty_print::custom_delims<HalfOpen>(x) << std::endl;
```

Implementation

[To be written.]