Chapter 7. Boost.Format

Boost.Format offers a replacement for the function `std::printf()`. `std::printf()` originates from the C standard and allows formatted data output. However, it is neither type safe nor extensible. Boost.Format provides a type-safe and extensible alternative.

Boost.Format provides a class called `boost::format`, which is defined in `boost/format.hpp`. Similar to `std::printf()`, a string containing special characters to control formatting is passed to the constructor of `boost::format`. The data that replaces these special characters in the output is linked via the operator `operator%`.

Example 7.1. Formatted output with `boost::format`

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%1%.%2%.%3%"} % 12 % 5 % 2014 << '\n';
}
```

The Boost.Format format string uses numbers placed between two percent signs as placeholders for the actual data, which will be linked in using `operator%`. Example 7.1 creates a date string in the form `12.5.2014` using the numbers 12, 5, and 2014 as the data. To make the month appear in front of the day, which is common in the United States, the placeholders can be swapped. Example 7.2 makes this change, displaying `5/12/2014`

Example 7.2. Numbered placeholders with `boost::format`

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%2%/%1%/%3%"} % 12 % 5 % 2014 << '\n';
}
```

To format data with manipulators, Boost.Format provides a function called `boost::io::group()`.

Example 7.3. Using manipulators with `boost::io::group()`

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%1% %2% %1%"} %
    boost::io::group(std::showpos, 1) % 2 << '\n';
}
```

Example 7.3 uses the manipulator `std::showpos()` on the value that will be associated with "%1%". Therefore, this example will display `+1 2 +1` as output. Because the manipulator `std::showpos()` has been linked to the first data value using `boost::io::group()`, the plus

sign is automatically added whenever this value is displayed. In this case, the format placeholder “%1%” is used twice.

If the plus sign should only be shown for the first output of 1, the format placeholder needs to be customized.

Example 7.4. Placeholders with special characters

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%|1$+| %2% %1%"} % 1 % 2 << '\n';
}
```

Example 7.4 does this. In this example, the first instance of the placeholder “%1%” is replaced with “%|1$+|”. Customization of a format does not just add two additional pipe signs. The reference to the data is also placed between the pipe signs and uses “1$” instead of “1%”. This is required to modify the output to be `+1 2 1`. You can find details about the format specifications in the Boost documentation.

Placeholder references to data must be specified either for all placeholders or for none. Example 7.5 only provides references for one of three placeholders, which generates an error at run time.

Example 7.5. `boost::io::format_error` in case of an error

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  try
  {
    std::cout << boost::format{"%|+| %2% %1%"} % 1 % 2 << '\n';
  }
  catch (boost::io::format_error &ex)
  {
    std::cout << ex.what() << '\n';
  }
}
```

Example 7.5 throws an exception of type `boost::io::format_error`. Strictly speaking, Boost.Format throws `boost::io::bad_format_string`. However, because the different exception classes are all derived from `boost::io::format_error`, it is usually easier to catch exceptions of this type.

Example 7.6 shows how to write the program without using references in the format string.

Example 7.6. Placeholders without numbers

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
```

```
  std::cout << boost::format{"%|+| %|| %||"} % 1 % 2 % 1 << '\n';
}
```

The pipe signs for the second and third placeholder can safely be omitted in this case because they do not specify any format. The resulting syntax then closely resembles `std::printf()` (see Example 7.7).

Example 7.7. `boost::format` with the syntax used from `std::printf()`

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%+d %d %d"} % 1 % 2 % 1 << '\n';
}
```

While the format may look like that used by `std::printf()`, Boost.Format provides the advantage of type safety. The letter “d” within the format string does not indicate the output of a number. Instead, it applies the manipulator `std::dec()` to the internal stream object used by `boost::format`. This makes it possible to specify format strings that would make no sense for `std::printf()` and would result in a crash.

Example 7.8. `boost::format` with seemingly invalid placeholders

```
#include <boost/format.hpp>
#include <iostream>

int main()
{
  std::cout << boost::format{"%+s %s %s"} % 1 % 2 % 1 << '\n';
}
```

`std::printf()` allows the letter “s” only for strings of type `const char*`. With `std::printf()`, the combination of “%s” and a numeric value would fail. However, Example 7.8 works perfectly. Boost.Format does not require a string. Instead, it applies the appropriate manipulators to configure the internal stream.

Boost.Format is both type safe and extensible. Objects of any type can be used with Boost.Format as long as the operator `operator<<` is overloaded for `std::ostream`.

Example 7.9. `boost::format` with user-defined type

```
#include <boost/format.hpp>
#include <string>
#include <iostream>

struct animal
{
  std::string name;
  int legs;
};

std::ostream &operator<<(std::ostream &os, const animal &a)
{
  return os << a.name << ',' << a.legs;
}
```

```
int main()
{
  animal a{"cat", 4};
  std::cout << boost::format{"%1%"} % a << '\n';
}
```

Example 7.9 uses `boost::format` to write an object of the user-defined type `animal` to standard output. This is possible because the stream operator is overloaded for `animal`.