Tips for formatting when printing to console from C++

Next time you struggle with console output formatting, refer to this article or the related cheat sheet.

By Stephan Avenwedde (Correspondent)

November 25, 2021 | 0 Comments | 7 min read

41 readers like this.

Image by: *Ray Smith*

When I started writing, I did it primarily for the purpose of documenting for myself. When it comes to programming, I'm incredibly forgetful, so I began to write down useful code snippets, special characteristics, and common mistakes in the programming languages I use. This article perfectly fits the original idea as it covers common use cases of formatting when printing to console from C++.

[Download the C++ std::cout cheat sheet]

As usual, this article comes with a lot of examples. Unless otherwise stated, all types and classes shown in the code snippets are part of the `std` namespace. So while you are reading this code, you have to put `using namespace std;` in front of types and classes. Of course, the example code is also available on GitHub.

Programming and development

- **Red Hat Developers Blog**
- **Programming cheat sheets**
- **Try for free: Red Hat Learning Subscription**
- **eBook: An introduction to programming with Bash**
- **Bash Shell Scripting Cheat Sheet**
- **eBook: Modernizing Enterprise Java**

Object-oriented stream

If you've ever programmed in C++, you've certainly already used cout. The *cout* object of type ostream comes into scope when you include <iostream>. This article focuses on *cout*, which lets you print to the console but the general formatting described here is valid for all stream objects of type ostream. An *ostream* object is an instance of basic_ostream with the template parameter of type char. The header <iosfwd>, which is part of the include hierarchy of <iostream>, contains forward declarations for common types.

The class *basic_ostream* inherits from basic_ios and this type, in turn, inherits from ios_base. On cppreference.com you find a class diagram that shows the relationship between the different classes.

The class ios_base is the base class for all I/O stream classes. The class basic_ios is a template class that has a specialization for common character types called **ios**. So when you read about *ios* in the context of standard I/O, it is the char-type specialization of basic_ios.

Formatting streams

In general, there are three ways of formatting ostream-based streams:

1. Using the format flags provided by ios_base.
2. Stream modifying functions defined in the header <iomanip> and <ios>.
3. By invoking a specific overload of the <<-operator.

All methods have their pros and cons, and it usually depends on the situation when which method is used. The following examples show a mixture of all methods.

Right justified

By default, *cout* occupies as much space as the data to print requires. To allow that right-justified output to take effect, you have to define the maximum width that a line is allowed to occupy. I use the format flags to reach the goal.

The flag for right-justified output and the width adjustment only applies to the subsequent line:

```
cout.setf(ios::right, ios::adjustfield);
cout.width(50);
cout << "This text is right justified" << endl;
cout << "This text is left justified again" << endl;
```

In the above code, I configure the right-justified output using setf. I recommend you apply the bitmask *ios::adjustfield* to *setf*, which causes all flags the bitmask specifies to be reset before the actual *ios::right* flag gets set to prevent colliding combinations.

Fill white space

When using right-justified output, the empty space is filled with blanks by default. You can change it by specifying the fill character using setfill:

```
cout << right << setfill('.') << setw(30) << 500 << " pcs" <<
cout << right << setfill('.') << setw(30) << 3000 << " pcs" <<
cout << right << setfill('.') << setw(30) << 24500 << " pcs" <
```

The code produces the following output:

```
...........................500 pcs
..........................3000 pcs
.........................24500 pcs
```

Combine

Imagine your C++ program keeps track of your pantry inventory. From time to time, you want to print a list of the current stock. To do so, you could use the following formatting.

The following code is a combination of left- and right-justified output using dots as fill characters to get a nice looking list:

```
cout << left << setfill('.') << setw(20) << "Flour" << right <
cout << left << setfill('.') << setw(20) << "Honey" << right <
cout << left << setfill('.') << setw(20) << "Noodles" << righ
cout << left << setfill('.') << setw(20) << "Beer" << right <<
```

Output:

```
Flour...............................0.70 kg
Honey..................................2 Glasses
```

```
Noodles..............................800 g
Beer.................................20 Bottles
```

Printing values

Of course, stream-based output also offers a multitude of possibilities to output all kinds of variable types.

Boolean

The boolalpha switch lets you convert the binary interpretation of a bool to a string:

```
cout << "Boolean output without using boolalpha: " << true <<
cout << "Boolean output using boolalpha: " << boolalpha << tru
```

The lines above produce the following output:

```
Boolean output without using boolalpha: 1 / 0
Boolean output using boolalpha: true / false
```

Addresses

If the value of an integer should be treated as an address, it is sufficient to cast it to *void** in order to invoke the correct overload. Here is an example:

```
unsigned long someAddress = 0x0000ABCD;
cout << "Treat as unsigned long: " << someAddress << endl;
cout << "Treat as address: " << (void*)someAddress << endl;
```

The code produces the following output:

```
Treat as unsigned long: 43981
Treat as address: 0000ABCD
```

The code prints the address with the correct length. A 32-bit executable produced the above output.

Integers

Printing integers is straightforward. For demonstration purpose I specify the base of the number using setf and setiosflags. Applying the stream modifiers hex/oct would have the same effect:

```
int myInt = 123;

cout << "Decimal: " << myInt << endl;

cout.setf(ios::hex, ios::basefield);
cout << "Hexadecimal: " << myInt << endl;

cout << "Octal: " << resetiosflags(ios::basefield) << setiost
```

Note: There is no indicator for the used base by default, but you can add one using showbase.

```
Decimal: 123
Hexadecimal: 7b
Octal: 173
```

Padding with zeros

```
0000003
0000035
0000357
0003579
```

You can get an output like the above by specifying the width and the fill character:

```
cout << setfill('0') << setw(7) << 3 << endl;
cout << setfill('0') << setw(7) << 35 << endl;
cout << setfill('0') << setw(7) << 357 << endl;
cout << setfill('0') << setw(7) << 3579 << endl;
```

Floating-point values

If I want to print floating-point values, I can choose between the *fixed*- and *scientific*-format. Additionally, I can specify the precision.

```
double myFloat = 1234.123456789012345;
int defaultPrecision = cout.precision(); // == 2

cout << "Default precision: " << myFloat << endl;
cout.precision(4);
cout << "Modified precision: " << myFloat << endl;
cout.setf(ios::scientific, ios::floatfield);
cout << "Modified precision & scientific format: " << myFloat
/* back to default */
cout.precision(defaultPrecision);
cout.setf(ios::fixed, ios::floatfield);
cout << "Default precision & fixed format:  " << myFloat << er
```

The code above produces the following output:

```
Default precision: 1234.12
Modified precision: 1234.1235
Modified precision & scientific format: 1.2341e+03
Default precision & fixed format:  1234.12
```

Time and Money

With put_money, you can print currency units in the correct, locale-dependent formatting. This requires that your console can output UTF-8 charset. Note that the variable *specialOffering* stores the monetary value in cents:

```
long double specialOffering = 9995;

cout.imbue(locale("en_US.UTF-8"));
```

```
cout << showbase << put_money(specialOffering) << endl;
cout.imbue(locale("de_DE.UTF-8"));
cout << showbase << put_money(specialOffering) << endl;
cout.imbue(locale("ru_RU.UTF-8"));
cout  << showbase << put_money(specialOffering) << endl;
```

The imbue-method of *ios* lets you specify a locale. With the command `locale -a`, you can get a list of all available locale identifiers on your system.

```
$99.95
99,950€
99,950₽
```

(For whatever reason, it prints euro and ruble with three decimal places on my system, which looks strange for me, but maybe this is the official formatting.)

The same principle applies to time output. The function put_time lets you print the time in the corresponding locale format. Additionally, you can specify which parts of a time object get printed.

```
time_t now = time(nullptr);
tm localtm = *localtime(&now);


cout.imbue(locale("en_US.UTF-8"));
cout << "en_US : " << put_time(&localtm, "%c") << endl;
cout.imbue(locale("de_DE.UTF-8"));
cout << "de_DE : " << put_time(&localtm, "%c") << endl;
cout.imbue(locale("ru_RU.UTF-8"));
cout << "ru_RU : " << put_time(&localtm, "%c") << endl;
```

The format specifier *%c* causes to print a standard date and time string:

```
en_US : Tue 02 Nov 2021 07:36:36 AM CET
de_DE : Di 02 Nov 2021 07:36:36 CET
ru_RU : Вт 02 ноя 2021 07:36:36
```

Creating custom stream modifiers

You can also create your own stream. The following code inserts a predefined string when applied to an *ostream* object:

```
ostream& myManipulator(ostream& os) {
    string myStr = ">>>Here I am<<<";
    os << myStr;
    return os;
}
```

Another example: If you have something important to say, like most people on the internet, you could use the following code to insert exclamation marks after your message depending on the level of importance. The level of importance gets passed as an argument:

```
struct T_Importance {
     int levelOfSignificance;
};

T_Importance importance(int lvl){
    T_Importance x = {.levelOfSignificance = lvl };
    return x;
}

ostream& operator<<(ostream& __os, T_Importance t){

    for(int i = 0; i < t.levelOfSignificance; ++i){
        __os.put('!');
    }
    return __os;
}
```

Both modifiers can now be simply passed to *cout*:

```
cout << "My custom manipulator: " << myManipulator << endl;
cout << "I have something important to say" << importance(5) <
```

Producing the following output:

```
My custom manipulator: >>>Here I am<<<

I have something important to say!!!!!
```

Conclusion

Next time you struggle with console output formatting, I hope you remember this article or the related cheat sheet.

In C++ applications, *cout* is the new neighbor of printf. While using *printf* is still valid, I would probably always prefer using *cout*. Especially the combination with the modifying function defined in *<ios>* results in nice, readable code.

Tags: PROGRAMMING CHEAT SHEETS

Stephan Avenwedde
Stephan is a technology enthusiast who appreciates open source for the deep insight of how things work. Stephan works as a full time support engineer in the mostly proprietary area of industrial automation software. If possible, he works on his Python-based open source projects, writing articles, or driving motorbike.

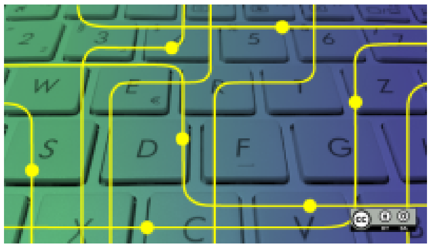
More about me

Comments are closed.

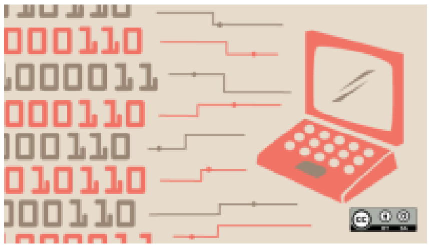
These comments are closed.

Related Content

[C vs. Go: Comparing programming languages]



Learn Tcl/Tk and Wish with this simple game



BASIC vs. FORTRAN 77: Comparing programming blasts from the past

This work is licensed under a Creative Commons Attribution-Share Alike 4.0 International License.

ABOUT THIS SITE

The opinions expressed on this website are those of each author, not of the author's employer or of Red Hat.

Opensource.com aspires to publish all content under a **Creative Commons license** but may not be able to do so in all cases. You are responsible for ensuring that you have the necessary permission to reuse any work on this site. Red Hat and the Red Hat logo are trademarks of Red Hat, Inc., registered in the United States and other countries.

A note on advertising: Opensource.com does not sell advertising on the site or in any of its newsletters.

Copyright ©2024 Red Hat, Inc.

Privacy Policy

Terms of use

Cookie preferences