TSIO

A type safe implementation of *sprintf* and more.

Table of Contents

1 Introduction.

Many C++ programmers who where previously C programmers miss the convenience and conciseness of the printf family of functions.

The new generation of c++ programmers don’t know what they are missing.

Take this trivial example:

```
std::cout << std::setw(10) << std::dec << 99 << '\n';
```

With printf this would be:

```
printf("%10d\n", 99);
```

Even for this trivial example, the printf code is smaller and arguably more readable.

A more realistic example would be:

std::cout << std::setw(10) << std::setprecision(3) << std::fixed << std::showpos << 123.45 << '\n';

versus:

```
printf("%+10.3f\n", 123.45);
```

Then why not just use the functions from the *std::printf* family?
Because they are fragile and error prone. When the argument does not correspond exactly with what the format expects, the behavior is undefined and often results in wrong output or core dumps.

Also, when the number of arguments is less than what the format expects, all kinds of havoc can be caused.

The TSIO packages provides a type safe version of *printf*, *sprintf* and *fprintf* that write into output streams or *std::string*. It also offers many useful extensions.

Lastly, the TSIO package allows to write the above example as:

```
std::cout << fmt(“%+10.3f”) << 123.45 << ‘\n’;
```

For more information on the standard printf family of functions, you can for example go to http://en.cppreference.com/w/cpp/io/c/fprintf.

2 Printf compatible functions.

The TSIO namespace contains the following printf compatible functions.

- `template <typename... Arguments>`
  `int sprintf(std::string& dest, const char* format, const Argumens&... arguments)`

  sprintf formats the *arguments* according to the *format* and outputs the result into the string *dest*, replacing the current contents of *dest*.

- `template <typename... Arguments>`
  `int asprintf(std::string& dest, const char* format, const Argumens&... arguments)`

  asprintf formats the *arguments* according to the *format* and outputs the result into the string *dest*, appending to the current contens of *dest*.

- `template <typename... Arguments>`
  `int fprintf(std::ostream& dest, const char* format, const Argumens&... arguments)`

  fprintf formats the *arguments* according to the *format* and outputs the result into the output stream *dest*.

- `template <typename... Arguments>`
  `int oprintf(const char* format, const Argumens&... arguments)`

  fprintf formats the *arguments* according to the *format* and outputs the result into the output stream *std::cout*.

- `template <typename... Arguments>`
  `int eprintf(const char* format, const Argumens&... arguments)`

  fprintf formats the *arguments* according to the *format* and outputs the result into the output stream *std::cerr*.

- `template <typename... Arguments>`
  `std::string fstring(const char* format, const Argumens&... arguments)`

  fprintf formats the *arguments* according to the *format* and *returns the result as a* std::string.

All the *printf* compatible functions use a format string as defined for *std::printf*.

All the conversion specifiers, flags and other parameters are faithfully implemented except wide characters.

Length modifiers are ignored, because the are not required. This means that the for example format `“%d”` can be used with all integral types (*char*, *short*, *int*, *long*, *long long* and the corresponding unsigned variants).

As with s*td::printf*, positional and sequential formats cannot be mixed. There is however an exception for tuple formatting. The formats embedded in a element format can be positional even if the element format itself is not. There is an example of this further down.

Many useful extensions are provided as described in the next chapter.

3 Extensions.

3.1 The *C* and *S* format specifiers.

The *C* and *S* format specifiers will output a string, just like *c* or *s* commands, but will replace unprintable characters by either a dot (.) or, in the alternative form (*#C* and *#S*), by an escape sequence.

Example:

```
     fstring("’%C %C’", '\x12', 'a')
```

produces

```
     ‘. a’
```

`and`

```
     fstring("’%#C %#C %#C’", '\a', 'a', '\x5')
```

produces

```
     ‘\\a a \\005’
```

Likewise

```
     fstring("’%S’", "12\a\b\f34")
```

produces

```
     ‘12...34’
```

and

```
     fstring("%#S", "12\a\b\f34")
```

produces

```
     ‘12\\a\\b\\f34’
```

3.2 The *T* format.

The *T* format allows to jump to a tab stop or to a given column in the output.

If no # flag is given, the format *%nnT* jumps to the next tab stop, where *nn* specifies the distance between tab stops.

If a # flag is given, the format *%#nnT* jumps to column *nn* in the output. If the output is already beyond that column, then a new line is generated followed with a jump to the given column.

The *T* format emits at least one space or new line.

3.3 The generic *s* format.

The *s* format specifier can be used for many data types, not just *const char**. It will choose an appropriate format as follows:

- For boolean arguments it will output the string literal *true* or *false*.
- For char arguments, it will format like the *c* specifier.
- For other integral arguments it will format like the *i* specifier for signed values and like *u* for unsigned values.
- For floating point arguments it will format like the *g* specifier.
- For pointer arguments (except *const char** of course) the *s* specifier is not implemented.
- The *s* format specifier accepts *std::string* as argument.
- The *s* format treats C-style char arrays and std char arrays as zero terminated character strings.

Example:

```
std::string str("This is a std::string");
fstring("'%s %s %s %s %s'", 123, 234.567, true, false, str)
```

produces

```
'123 234.567 true false This is a std::string'
```

and

```
fstring("'%06s %5.2s %10.3s'", 123, 234.567, 98.765)
```

produces

```
'000123 2.3e+02       98.8'
```

3.4 Fill characters.

The *TSIO* package supports two types of fill characters: numeric and alphabetic fill characters.

Numeric fill characters are used in formatting numeric values. They are inserted between the sign or the radix indicator and the value.

The numeric fill character is specified by adding *‘x* to the format flags. The zeorofill flag (*0*) sets the numeric fill character to ‘0’.

Example:

```
fstring("%'*7d', %#'*7x", -23, 0xab)
```

produces:

```
-****23', 0x***ab
```

The alphabetic fill characters are used in formatting both numeric and alphabetic data. Theu are inserted outside the value (including sign and radix indicator).

The alphabetic fill character is specified by adding “x to the format flags. Don’t forget to escape the quote character in the format string.

Example:

```
fstring("%\"*7d', %#\"*7x", -23, 0xab)
```

produces:

```
****-23', ***0xab
```

and

```
fstring("%\"*20s", "abc")
```

produces:

```
*****************abc
```

3.5 Centering text.

Text can be centered by adding the ^ flag to the format. The alphabetic fill character will be used to fill the areas left and right of the text.

The centering works on all values, numeric and alphabetic.

Example:

```
 fstring("'%^20s'", "1234")
```

Produces:

```
    '        1234        '
```

3.6 Containers, tuples, pairs and arrays as arguments.

In the context of this text, a range is any class for which free standing *begin* and *end* functions exist. This obviously includes all *std* containers.

Additionally *std::tuple* and *std::pair* and C-style arrays are also considered ranges.

implicit range format.

When a range is passed as a parameter to a simple format, all elements of the ranges will be printed with the same format.

Note that this will not work for C-style arrays.

Not also that this will work for a *const char** pointer. The format will be applied to every *char* preceding the terminating 0.

Example:

```
    std::vector<int> v = {10, 200, 3000 };
    fstring("vector { %10d }", v)
```

produces:

```
    vector {         10        200       3000 }
```

explicit range format.

The range format allows to format the elements of a range, giving a prefix and a suffix for each element. Each element of the range will be formatted using the same format.

The range format is specified with the *%[* format specifier. The syntax of this specifier is

%[xxxxx%fffyyyy%] where *xxxxx* is the prefix, *yyyy* is the suffix and *fff* is the format to be used for each element.

An example will make this clear:

```
    std::vector<int> v = { 9, 8, 7, 6 };
    fstring("%[v=%d, %]", v)
    //         ^^ prefix
    //           ^^ format to be executed for each element
    //              ^^ suffix
```

produces:

```
    v=9, v=8, v=7, v=6,
```

The extra comma at the end of the output is of course annoying, but this can be cured with the *#* flagin the *%]* format. This flag suppresses the suffix on the last element.

Example:

```
    fstring("{ %[v=%d, %#] }", v)
```

produces:

```
    { v=9, v=8, v=7, v=6 }
```

Since *begin* and *end* are declared for arrays, the following also works.

Example:

```
    double fa[] = { 1.2, 2.3, 3.4, 4.55555 };
    fstring("{ %[v=%.2f, %#] }", fa)
```

produces:

```
    { v=1.20, v=2.30, v=3.40, v=4.56 }
```

Since all elements of a *tuple* or a *pair* are formatted using the same fortmat, care must be taken to make sure that the format applies to all elements. The simplest way to achieve this is to use the *%s* format.

Example:

```
    auto t = std::make_tuple(1, 2.3, "four");
    fstring("{ %[v=%s, %#] }", t)
```

produces

```
    { v=1, v=2.30000, v=four }
```

The range format can also be used to format a single element form a container.

The syntax for this usage is *%nn[ffff %]* where *nn* is the index (starting at 1) of the element to be formatted.

Example:

```
    std::vector<int> v = {10, 200, 3000};
    fstring("%2[ %d %]", v)
```

produces

```
    200
```

Slices

The range format can also be used to print a slice from a range.

The format syntax is then *%iii.ccc[... %]*. where *iii* is the start index in the range and *ccc* is the number of elements to format. If *iii* is omitted, then it defaults to 0. If *ccc* is omitted but the dot is not then then the count defaults to 'all remaining elements' If both *ccc* and the dot are omitted but *ccc* is not then only one element is formatted.

By default indexes start at 1. To make indexes start at 0, the # flag can be used.

Since the printf format does not allow a 0 to be given in the width position, The *iii* value must be omitted if it is 0.

Overview:

`%[ fff %]`	prints the whole range. (*fff* is the format to be used).
`%.[ fff %]`	prints the whole range.
`%.3[ fff %]`	prints the the 3 first elements in the range.
`%2.2[ fff %]`	prints the second and third element in the range.
`%#1.2[ fff %]`	prints the second and third element in the range.

Element format.

ranges can also be formatted with different formats for each element.

The *element format* specifier does exactly that. The syntax for this specifier is *%<* followed with the formats of the elements, one per element followed with %>.

Example:

```
auto t = std::make_tuple(1, 2.3, "four");
fstring("%<%5d %5.2f %10s%>", t)
```

produces:

```
    1  2.30       four
```

Although in general, positional formats and sequential formats cannot be mixed, the embedded formats can be positional, even when the element format itself is not.

Example:

```
auto t = std::make_tuple(1, 2.3, "four");

fstring("%<{ %2$6.2f, %1$5d, %3$5s and again %2$6.2f%> }", t)
```

produces

```
{   2.30,     1,  four and again   2.30 }
```

3.7 Nesting formats.

The range format (*%[… %]*) the element format (*%< … %>*) and the repeating format (*%{ … %}*) can be nested with some limitations:

When all repeating formats embedded in a range or element format are unrolled, the result must still be a valid range or element format.

Dynamic formats (using ***, *%nn$* or **$*) cannot be nested in range or element formats.

The *%n* format cannot be nested in range format or element formats.

Repeating formats can be nested into other repeating formats without limitation.

A nice axample of nested formats is the formatting of a *std::map:*

```
    std::map<int, const char*> m = { {1, "one"}, {3, "three"}, {2,
"two"} };

    fstring("%[%<{ key: %3d, value: %5s }%>, %#]", m)

         // ^^^ start of map formatting

         //    ^^ start of element formatting

         //      ^^^^^^^^^^^^^^^^^^^^^^^^ formats for element

         //                              ^^ end of element

         //                                  ^^^ end of map
```

produces:

```
    { key:   1, value:   one }, { key:   2, value:   two }, { key:
3, value: three }
```

3.8 Repeating formats.

The *tsio* package provides repeating formats. A repeating format is a format that is executed "abcabcabcabcabcfor multiple consecutive arguments.

A repeating format starts with *%n{* where n is an integer value that specifies the repeat count and ends with *%}*. All the text and format specifications between these markers are repeated. Enough parameters must be passed to satisfy all these format specifiers.

Example:

```
fstring("%10{-=%}")
```

produces:

```
-=-=-=-=-=-=-=-=-=-=
```

and

```
fstring("repeat: %3{%4d%}", 1, 22, 333)
```

produces:

```
repeat:    1  22 333
```

Repeating format can be nested without any practical limit.

It is possible to specify the repeat count at execution time using the * notation.

Example:

```
fstring("%*{abc%}", 5)
```

produces:

```
abcabcabcabcabc
```

3.9 The *%N* format.

Inside a range format, the *%N* command can be used to generate the value of the current index in the container. By default indexes start counting at one, but if the *alternative (#)* flag is given then they start at zero,

Example:

```
    auto t = std::make_tuple(1, 2.3, "four");
    fstring("%[%5N: %s%]", t)
```

produces

```
        1: 1    2: 2.3    3: four
```

and

```
    std::vector<int> v = {10, 200, 3000};
    fstring("%[%5N: %s%]", v)
```

produces

```
        1: 10    2: 200    3: 3000
```

and with the *alternative* flag:

```
    fstring("%[%#5N: %s%]", v)
```

produces

```
        0: 10    1: 200    2: 3000
```

Likewise the *%N* format can also be used inside a repeated format to generate the repeat count.

Example:

```
    fstring("%5{%5N%}")
```

produces

```
        1    2    3    4    5
```

and

```
    fstring("%5{%#5N%}")
```

produces

```
        0    1    2    3    4
```

3.10 Customized formats.

The tsio package allows to define your own formatters for your own classes and seamlessly integrating them in the format strings.

To use customized formats, you need to do the following:

- Create a formatter class for the class you want to format.
- Create a getFormatter function that will be called by tsio to find your formatter class.
- Use the *customized format specifier* in your format string.

The formatter class.

You need to create a formatter class that can format the members of your class. This formatter class can be any class, as long as it has a member function with the signature:

```
std::tuple<bool, std::string> format(SingleFormat fmt)
```

This function takes *SingleFortmat* parameter that describes a format. It has methods to retrieve the flags, width, precision and specifier from the format and to modify these values. These are described further.

It returns a *tuple* odf a bool and a string. If the bool is false, the format was invalid for your formatter. If the bool is true, then the string contains the result after formatting.

The getFormatter function.

You have to provide a *getFormatter* function with the following signature:

```
formatter_class getFormatter(const your_class& t)
```

It usualy simply returns a new instance of the formatter class.

The customized format.

In a format string, the customized format is specified as *%(fff %)*, where fff are the formats that you want to forward to your formatter. The *fff* formats can be any syntactically correct format, but it is up to you to interpret that format. Existing formats (like *%d* or *%s*...) can be used, but they loose their predefined meaning.

There are a few formats that retain their predefined meaning and will be resolved by tsio i.s.o sending them to the formatter. These formats are *%T*, *%N*, *%%* and *%{*. The range and element formats shall not be used in a customized format.

How it fits together.

When tsio encounters a *customized format*, it calls *getFormatter* to get a copy of your formatter. For every format inside the *customized format*, it calls the *format* function described above and includes the returned string in the output.

How your formatter creates the result of a format is entirely up to you, but tsio gives you some help:

- Your formatter function can call sprintf recursively to create the result string.
- The SingleFormat class has a *asprintf* function that formats a value according to the contents of the *SingleFormat*. As explained above, the *SingleFormat* can be modified to suit your needs.

A simple example can be found in the file *examples/showTime.cpp*.

***The SingleFormat* class.**

The *SingleFormat* class has the following API:

Functions to get format flags:

```
    bool getNumericFill() const;
    bool getAlfaFill() const;
    bool getPlusIfPositive() const;
    bool getSpaceIfPositive() const;
    bool getLeftJustify() const;
    bool getCenterJustify() const;
    bool getAlternative() const;
```

Functions to check whether width or precision were specified:

```
    bool widthGiven() const;
    bool precisionGiven() const;
```

Functions to get format elements:

```
    unsigned getWidth() const;
    unsigned getPrecision() const;
    char getSpecifier() const;
    char getFillCharacter() const;
```

Functions to modify the format:

```
void setWidthGiven(bool v = true);

void setPrecisionGiven(bool v = true);

void setWidth(unsigned v);

void setPrecision(unsigned v);

void setSpecifier(char v);

void setFillCharacter(char v);

void setAlternative(bool v);
```

Function to format a value according to the *SingleFormat*.

```
template <typename T>

void asprintf(std::string& dest, const T& t);
```

4 The fmt io manipulator.

The *tsio::fmt* io manipulator allows to set *std::osttream* flags according to a printf like format.

Although the format uses the same syntax as *tsio::sprintf*, it does not implement all its features. Only the features are implemented that have an equivalent in the *ostream* flags.

For example, the *space if positive* flag from printf has no equivalence in *ostream'*

Some formats, like the *g* format reproduce different results.

An example will make its usage clear:

```
std::string str;
short s = 123;
int i = 42;
float f = float(234.567);
double d = 345.678;
const char* ch = "Don't panic";

std::cout << fmt("%-15s") << ch << ',' << fmt("%12i") << s
    << ',' << fmt("d") << i << ',' << fmt("12.2f") << f
    << ',' << fmt("%7.1E") << d << ',' << fmt("a") << d
    << fmt() << '\n';
```

produces:

```
"Don't panic    ,         123,42,
 234.57,3.5E+02,0x1.59ad916872b02p+8"
```

The *fmt* io manipulator clears all *ostream* flags when it starts. It then sets the flag as required by the format. After the next value is printed, the flags keep the newly set values except *width* that is reset to default.

The call to *fmt()* without any format resets all flags to their default value.

5 Cached formats.

The *TSIO* package provides the *CFormat* class that caches a format string in a binary form that allows multiple fast invocations of the sameformat.

The *CFormat* class is instantiated with the format string as parameter. Then it can replace the format string in all output functions.

The execution of an output function with a cached format is approximately 25% faster.

Example:

```
    CFormat cformat("%5d");

    for (int i = 0; i < 10; ++i) {
        oprintf(cformat, i);
    }

    oprintf("\n");
```

produces

```
        0    1    2    3    4    5    6    7    8    9
```

6 Error reporting.

The *TSIO* functions report all errors that are encountered during formatting on *std::cerr*.

A call to any function except *fstring* returns a negative value if an error occurred. Otherwise the number of characters generated during formatting is returned. This is compatible with the standard functions.

The error message are precise and the location of the error is indicated.

Example:

```
    fstring("%d %d", 1)
```

produces

```
    TSIO error: Extraneous format.
             at "%d %d"
                   ^
```

7 Appendix. Summary of formats.

Formats.

%	literal %.
a	hexadecimal floating point.
A	hexadecimal floating point uppercase.
b	binary integral.
B	binary integral uppercase.
c	character.
C	character possibly escaped.
d	decimal integral.
e	scientific floating point.
E	scientific floating point uppercase.
f	fixed floating point.
F	fixed floating (same as **f**).
g	floating point default.
G	floating point default uppercase.
i	decimal integer (same as **d**),
n	return numbers of characters written.
N	index of range format or repeat format.
o	octal integral
p	pointer.
s	character string or generic format.
S	character string, possibly escaped.
u	unsigned integral.
x	hexadecimal integral.
X	hexadecimal integral uppercase.
{	start repeating format.
}	end repeating format.
[	start range format.

]	end range format.
<	start element format.
>	end element format.

Parameters.

*****	width or precision from argument.
%n$	positional argument.
*m$	width or precision from positional argument.

Flags.

-	left justify
^	center justify.
#	alternative.
0	zerofill
space	add leading space if positive.
+	always add sign.
'x	set *x* as numeric fill character.
"x	set x as alphabetic fill character.

Length modifiers.

h, **j**, **l**, **t**, **z**	All length modifiers are unneeded and thus ignored.