字符串格式化漫谈

on April 29, 2019 under cxx

38 minute read

前言

春风化雨，万物复苏，编程始于 `Hello world` ， C 语言的 Hello world 可如下，同样 C++ 亦可如下:

```
#include <stdio.h>
int main() {
  printf("Hello World!");
  return 0;
}
```

printf 函数类型为:

```
int printf(const char *fmt,...);
```

按照此声明我们可以格式化输出:

```
#include <stdio.h>
int main() {
  const char *name="Tony Stark";
  printf("Hello %s\n",name);
  return 0;
}
```

使用 C 编译器 Clang/MSVC/GCC 将其编译运行，在终端或者命令行中会输出如下结果：

> $ Hello Tony Stark

printf 函数家族声明如下：

```
// https://en.cppreference.com/w/cpp/header/cstdio
int printf( const char* format, ... );
int fprintf(std::FILE* stream, const char* format, ... );
int sprintf( char* buffer, const char* format, ... );
int snprintf( char* buffer, std::size_t buf_size, const char* format, ... );
int vprintf( const char* format, va_list vlist );
int vfprintf( std::FILE* stream, const char* format, va_list vlist );
int vsprintf( char* buffer, const char* format, va_list vlist );
int vsnprintf( char* buffer, std::size_t buf_size, const char* format, va_list vlist );
```

如果我们需要格式化字符串时，则可以使用 `sprintf` 或者是 `snprintf` ，那么问题来了， `snprintf` 是如何格式化的？

格式化内幕

要了解 snprintf 的细节，我们需要去找一个 libc 了解一番，这里建议是 musl (https://github.com/bminor/musl)，musl 只支持 Linux，没有像 Glibc 那么多的遗留代码，代码比较整洁。而 Visual C++ 的 ucrt 源码基本使用 C++ 模板编写，比较复杂，不容易借此理清 snprintf 的细节。翻阅 musl snprintf.c 源码，我们发现 snprintf 将会调用 `vsnprintf` ，

```c
// https://github.com/bminor/musl/blob/master/src/stdio/snprintf.c
#include <stdio.h>
#include <stdarg.h>

int snprintf(char *restrict s, size_t n, const char *restrict fmt, ...)
{
        int ret;
        va_list ap;
        va_start(ap, fmt);
        ret = vsnprintf(s, n, fmt, ap);
        va_end(ap);
        return ret;
}
// https://github.com/bminor/musl/blob/master/src/stdio/vsnprintf.c
int vsnprintf(char *restrict s, size_t n, const char *restrict fmt, va_list ap)
{
        unsigned char buf[1];
        char dummy[1];
        struct cookie c = { .s = n ? s : dummy, .n = n ? n-1 : 0 };
        FILE f = {
                .lbf = EOF,
                .write = sn_write,
                .lock = -1,
                .buf = buf,
                .cookie = &c,
        };

        if (n > INT_MAX) {
                errno = EOVERFLOW;
                return -1;
        }

        *c.s = 0;
```

```
    return vfprintf(&f, fmt, ap);
}
```

在 musl 之中，vsnprintf 创建一个 FILE 结构，然后最终使用 vfprintf 格式化字符串。这里使用了 `va_list` `va_start` `va_end` 宏，这组宏将变参函数的参数从函数栈中取出来，从而实现了变参函数的功能，我们可以参考 Visual C++ 在 `vadefs.h` (https://gist.github.com/fcharlie/e2b6a2d578d7b484d0338886ce0db768) 中的定义。

而 va_list 本质上是从函数参数栈中获得特定位置的值。

`vfprintf` 函数的源码在： musl: src/stdio/vfprintf.c (https://github.com/bminor/musl/blob/master/src/stdio/vfprintf.c)。我们可以发现格式化输出实际上是解析 `format` 字符串，在解析到占位符时，使用 `va_arg` 提取 `va_list` 中的变量（或者转变为特定格式字符串后）替换占位符，从而实现格式化输出的目的（文件或者缓冲区）。格式化占位符以 `%` 开头，支持格式化的类型可以参考 http://pubs.opengroup.org/onlinepubs/9699919799/functions/fprintf.html (http://pubs.opengroup.org/onlinepubs/9699919799/functions/fprintf.html)

了解到格式化输出的原理之后，我们可以很容易的实现一个格式化字符串或者格式化输出函数，在 nginx 源码中，就有一个 类似实现： `ngx_vslprintf` (https://github.com/nginx/nginx/blob/27b3d3dcca5fcc82350a823881f3d06161327b59/src/core/ngx_string.c#L163) ，这个代码比较简单比较容易移植。

C-Style 格式化的缺陷

上述格式化使用变参函数，使用 va_list 获得参数值，这种使用 va_list 的函数在 C++ 中是不被推荐的 F.55: Don't use va_arg arguments (https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md#F-varargs)。当用户编码时，可能是程序员疏忽或者故意，依赖 va_list 的代码很容易由于类型不匹配，参数个数不匹配而造成栈溢出。导致安全漏洞或者是程序崩溃。以下代码可能会导致程序崩溃，并且编译器也不会警告：

```
#include <string_view>
#include <cstdio>
#include <cstdarg>

int dump(const char *fmt,...){
  int ret;
  va_list ap;
  va_start(ap,fmt);
  vfprintf(stderr,fmt,ap);
  va_end(ap);
  return ret;
}

int main(){
  std::string_view name="Tony Stark";
  dump("hello %s\n",name);
  return 0;
}
```

编译器的构建信息如下:

```
使用内建 specs。
COLLECT_GCC=/opt/gcc/bin/g++
COLLECT_LTO_WRAPPER=/opt/gcc/libexec/gcc/x86_64-linux-gnu/9.0.1/lto-wrapper
目标：x86_64-linux-gnu
配置为：../configure --with-pkgversion=Baslat.Inc --prefix=/opt/gcc --enable-shared --enable-linker-build-id --witho
ut-included-gettext --enable-threads=posix --enable-checking=release --enable-languages=c,c++ --disable-multilib --
enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default-libstdcxx-abi=new --enable-gnu-unique-object --di
sable-vtable-verify --enable-libmpx --enable-plugin --with-system-zlib --with-abi=m64 --build=x86_64-linux-gnu --ho
st=x86_64-linux-gnu --target=x86_64-linux-gnu
线程模型：posix
gcc 版本 9.0.1 20190426 (prerelease) (Baslat.Inc)
```

运行编译器命令：

> /opt/gcc/bin/g++ -fsanitize=address -fno-omit-frame-pointer fuck.cc -std=c++17

运行程序：

> ./a.out

地址消毒剂报告如下：

```
AddressSanitizer:DEADLYSIGNAL
=================================================================
==16509==ERROR: AddressSanitizer: SEGV on unknown address 0x00000000000a (pc 0x7fc137bc3af2 bp 0x7ffffbf14ea0 sp 0x7ffffbf145c8 T0)
==16509==The signal is caused by a READ memory access.
==16509==Hint: address points to the zero page.
    #0 0x7fc137bc3af1  (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x109af1)
    #1 0x7fc137b0e61c  (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x5461c)
    #2 0x7fc137b0efb4 in __interceptor_vfprintf (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x54fb4)
    #3 0x400cf0 in dump(char const*, ...) (/tmp/a.out+0x400cf0)
    #4 0x400e06 in main (/tmp/a.out+0x400e06)
    #5 0x7fc136dabb96 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x21b96)
    #6 0x400b09 in _start (/tmp/a.out+0x400b09)

AddressSanitizer can not provide additional info.
SUMMARY: AddressSanitizer: SEGV (/usr/lib/x86_64-linux-gnu/libasan.so.5+0x109af1)
==16509==ABORTING
```

出现问题其实很容易理解，由于 std::string_view 的结构大致如下：

```
template<class charT, class traits = char_traits<charT>>
class basic_string_view {
public:
      // types
      typedef traits traits_type;
      typedef charT value_type;
      typedef charT* pointer;
      typedef const charT* const_pointer;
      typedef charT& reference;
      typedef const charT& const_reference;
      typedef implementation-defined const_iterator;
      typedef const_iterator iterator;
      typedef reverse_iterator<const_iterator> const_reverse_iterator;
      typedef const_reverse_iterator reverse_iterator;
      typedef size_t size_type;
      typedef ptrdiff_t difference_type;
      static constexpr size_type npos = size_type(-1);
      // 7.3, basic_string_view constructors and assignment operators
      constexpr basic_string_view() noexcept;
      constexpr basic_string_view(const basic_string_view&) noexcept = default;
      basic_string_view& operator=(const basic_string_view&) noexcept = default;
      template<class Allocator>
      constexpr basic_string_view(const charT* str);
      constexpr basic_string_view(const charT* str, size_type len);
      // 7.4, basic_string_view iterator support
      constexpr const_iterator begin() const noexcept;
      constexpr const_iterator end() const noexcept;
      constexpr const_iterator cbegin() const noexcept;
      constexpr const_iterator cend() const noexcept;
      const_reverse_iterator rbegin() const noexcept;
      const_reverse_iterator rend() const noexcept;
      const_reverse_iterator crbegin() const noexcept;
      const_reverse_iterator crend() const noexcept;
```

```
// 7.5, basic_string_view capacity
constexpr size_type size() const noexcept;
constexpr size_type length() const noexcept;
constexpr size_type max_size() const noexcept;
constexpr bool empty() const noexcept;
// 7.6, basic_string_view element access
constexpr const_reference operator[](size_type pos) const;
constexpr const_reference at(size_type pos) const;
constexpr const_reference front() const;
constexpr const_reference back() const;
constexpr const_pointer data() const noexcept;
// 7.7, basic_string_view modifiers
constexpr void remove_prefix(size_type n);
constexpr void remove_suffix(size_type n);
constexpr void swap(basic_string_view& s) noexcept;
size_type copy(charT* s, size_type n, size_type pos = 0) const;
constexpr basic_string_view substr(size_type pos = 0, size_type n = npos) const;
constexpr int compare(basic_string_view s) const noexcept;
constexpr int compare(size_type pos1, size_type n1, basic_string_view s) const;
constexpr int compare(size_type pos1, size_type n1,
                      basic_string_view s, size_type pos2, size_type n2) const;
constexpr int compare(const charT* s) const;
constexpr int compare(size_type pos1, size_type n1, const charT* s) const;
constexpr int compare(size_type pos1, size_type n1,
                      const charT* s, size_type n2) const;
constexpr size_type find(basic_string_view s, size_type pos = 0) const noexcept;
constexpr size_type find(charT c, size_type pos = 0) const noexcept;
constexpr size_type find(const charT* s, size_type pos, size_type n) const;
constexpr size_type find(const charT* s, size_type pos = 0) const;
constexpr size_type rfind(basic_string_view s, size_type pos = npos) const noexcept;
constexpr size_type rfind(charT c, size_type pos = npos) const noexcept;
constexpr size_type rfind(const charT* s, size_type pos, size_type n) const;
constexpr size_type rfind(const charT* s, size_type pos = npos) const;
constexpr size_type find_first_of(basic_string_view s, size_type pos = 0) const noexcept;
```

```
      constexpr size_type find_first_of(charT c, size_type pos = 0) const noexcept;
      constexpr size_type find_first_of(const charT* s, size_type pos, size_type n) const;
      constexpr size_type find_first_of(const charT* s, size_type pos = 0) const;
      constexpr size_type find_last_of(basic_string_view s, size_type pos = npos) const noexcept;
      constexpr size_type find_last_of(charT c, size_type pos = npos) const noexcept;
      constexpr size_type find_last_of(const charT* s, size_type pos, size_type n) const;
      constexpr size_type find_last_of(const charT* s, size_type pos = npos) const;
      constexpr size_type find_first_not_of(basic_string_view s, size_type pos = 0) const noexcept;
      constexpr size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;
      constexpr size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;
      constexpr size_type find_first_not_of(const charT* s, size_type pos = 0) const;
      constexpr size_type find_last_not_of(basic_string_view s, size_type pos = npos) const noexcept;
      constexpr size_type find_last_not_of(charT c, size_type pos = npos) const noexcept;
      constexpr size_type find_last_not_of(const charT* s, size_type pos, size_type n) const;
      constexpr size_type find_last_not_of(const charT* s, size_type pos = npos) const;
      constexpr bool starts_with(basic_string_view s) const noexcept; // C++2a
      constexpr bool starts_with(charT c) const noexcept;             // C++2a
      constexpr bool starts_with(const charT* s) const;               // C++2a
      constexpr bool ends_with(basic_string_view s) const noexcept;   // C++2a
      constexpr bool ends_with(charT c) const noexcept;               // C++2a
      constexpr bool ends_with(const charT* s) const;                 // C++2a
private:
      const_pointer data_;  // exposition only
      size_type     size_;  // exposition only
    };
```

由于 string_view 值被错误的转变为 `char *` ，而 C-Style 的字符串是 `null-terminated string` ，在处理 `%s` 的时候无法正常解析到终止字符，出现溢出然后导致程序崩溃，同样，如果类型宽度不一致，比如 format 中需要 `%lld` ，而输入为 `int` 同样容易出现问题，但这种问题可能更多的预期结果不一致。。

在这个例子中，如果将 `std::string_view` 改成 `std::string` , clang 8.0.1 则会报告 std::string 不是 POD 类型的错误：

```
fuck2.cc:16:21: error: cannot pass object of non-trivial type 'std::string' (aka 'basic_string<char>') through vari
adic
      function; call will abort at runtime [-Wnon-pod-varargs]
  dump("hello %s\n",name);
                    ^
1 error generated.
```

安全格式化解决方案

编译器的参数匹配检查

为了减少上面错误的发生，开发者增加了很多解决方案，比如，对于 C 或者 C++ 而言，可以使用特定的 `Attributes` 限制函数的属性，当格式不匹配时，编译器会发出警告。

```
#ifdef __GNUC__
void log_unlocked(int level, const char *fmt, ...)
    __attribute__((__format__(__printf__, 2, 3)));
#else
void log_unlocked(int level, const char *fmt, ...);
#endif
```

但这种方案受限于编译器，不是一个普遍方案。Clang (https://clang.llvm.org/docs/AttributeReference.html#format) 保持了对 GCC 的兼容，可以使用上述 `__attribute__` 。而 MSVC 则可以使用 SAL：`_Printf_format_string_` (https://docs.microsoft.com/en-us/previous-versions/visualstudio/visual-studio-2010/ms235402(v=vs.100))。

变参函数模板的辅助

如果要在 C++ 当中使用类似的格式化函数方案，可以变参函数模板包装，cppwinrt 的作者 Kenny Kerr 就有一篇文章告诉人们改怎么做： Windows with C++ - Using Printf with Modern C++ (https://msdn.microsoft.com/en-us/magazine/dn913181.aspx)。

```
template <typename T> T Argument(T value) noexcept { return value; }
template <typename T>
T const *Argument(std::basic_string<T> const &value) noexcept {
  return value.c_str();
}

template <typename... Args>
int StringPrint(char *const buffer, size_t const bufferCount,
                char const *const format, Args const &... args) noexcept {
  int const result = snprintf(buffer, bufferCount, format, Argument(args)...);
  return result;
}

template <typename... Args>
std::string StrFormat(const char *format, Args... args) {
  std::string buffer;
  size_t const size = StringPrint(nullptr, 0, format, args...);
  buffer.resize(size);
  StringPrint(&buffer[0], buffer.size() + 1, format, args...);
  return buffer;
}
```

在格式化时将 `std::string` 转变为了 `const char *` ，这种方案的缺陷有两处，第一 `std::string` 是可以存在 `\0` 这样的字符的，但是在这里却被截断，第二， `std::string` 需要再次计算长度。

现代 C++ 格式化库

既然 C-Style 的格式化方案缺陷那么多，那么 C++ 开发者们也会不遗余力的去造轮子，实现自己的目标的，早期比如我刚学 C++ 那会，都是 `iostream` 家族，字符串格式化可以使用 `stringstream` ， `iostream` 这类方案一直被视为糟糕的设计，一来继承复杂，二来效率低。特别在 C++11 变参模板等特性出来后，饱受鄙视，因此也不是建议的格式化方案。现在 C++20 都快发布了，也涌现了一些更好的方案，好的格式化库有积极入标准的 fmtlib (https://github.com/fmtlib/fmt)。还有 Facebook 的 C++ 标准库补充 folly (https://github.com/facebook/folly) 也有一个 `format` 实现。在 Google 里面，很多项目使用了 C++，他们也积累了一些 C++ 组件，后来开源了 Abseil (https://github.com/abseil/abseil-cpp)，Abseil 中也有字符串格式化函数 `absl::StrFormat` ， `absl::StrAppendFormat` ，当然还有一些实现比较慢，代码不太干净的，这里也就不多说了。

积极入标准的 fmtlib

fmtlib 目前是冲着进入 C++ 标准去的，它支持两种风格，一个是类似 python 的风格：

```
fmt::format("The answer is {}.", 42);
fmt::print("I'd rather be {1} than {0}.", "right", "happy");
```

这种风格还可以通过重载 `format_to` 函数， 输出特定的对象。

fmtlib 还支持 printf 的格式化风格，使用变参模板展开，是格式化安全的。

```
std::string message = fmt::sprintf("The answer is %d", 42);
```

当类型不匹配时 fmtlib 会抛出异常，这是一种运行时行为。另外 fmtlib 还支持 `wchar_t` ，这在 `Windows` 系统中比较重要。在格式化浮点类型时，可能会回退到 `snprintf` 。

Facebook folly format

Folly format (https://github.com/facebook/folly/blob/master/folly/docs/Format.md) 的风格类似于 python 的格式化风格，与 fmtlib 的第一种一致。

```
using folly::format;
using folly::sformat;
using folly::vformat;
using folly::svformat;

// Objects produced by format() can be streamed without creating
// an intermediary string; {} yields the next argument using default
// formatting.
std::cout << format("The answers are {} and {}", 23, 42);
// => "The answers are 23 and 42"

// If you just want the string, though, you're covered.
std::string result = sformat("The answers are {} and {}", 23, 42);
// => "The answers are 23 and 42"

// Arguments can be referenced out of order, even multiple times
std::cout << format("The answers are {1}, {0}, and {1} again", 23, 42);
// => "The answers are 42, 23, and 42 again"

// It's perfectly fine to not reference all arguments
std::cout << format("The only answer is {1}", 23, 42);
// => "The only answer is 42"

// Values can be extracted from indexable containers
// (random-access sequences and integral-keyed maps), and also from
// string-keyed maps
std::vector<int> v {23, 42};
std::map<std::string, std::string> m { {"what", "answer"} };
std::cout << format("The only {1[what]} is {0[1]}", v, m);
// => "The only answer is 42"

// If you only have one container argument, vformat makes the syntax simpler
std::map<std::string, std::string> m { {"what", "answer"}, {"value", "42"} };
```

```
std::cout << vformat("The only {what} is {value}", m);
// => "The only answer is 42"
// same as
std::cout << format("The only {0[what]} is {0[value]}", m);
// => "The only answer is 42"
// And if you just want the string,
std::string result = svformat("The only {what} is {value}", m);
// => "The only answer is 42"
std::string result = sformat("The only {0[what]} is {0[value]}", m);
// => "The only answer is 42"

// {} works for vformat too
std::vector<int> v {42, 23};
std::cout << vformat("{} {}", v);
// => "42 23"

// format and vformat work with pairs and tuples
std::tuple<int, std::string, int> t {42, "hello", 23};
std::cout << vformat("{0} {2} {1}", t);
// => "42 23 hello"

// Format supports width, alignment, arbitrary fill, and various
// format specifiers, with meanings similar to printf
// "X<10": fill with 'X', left-align ('<'), width 10
std::cout << format("{:X<10} {}", "hello", "world");
// => "helloXXXXX world"

// Field width may be a runtime value rather than part of the format string
int x = 6;
std::cout << format("{:-^*}", x, "hi");
// => "--hi--"

// Explicit arguments work with dynamic field width, as long as indexes are
// given for both the value and the field width.
```

```
std::cout << format("{2:+^*0}",
9, "unused", 456); // => "+++456+++"

// Format supports printf-style format specifiers
std::cout << format("{0:05d} decimal = {0:04x} hex", 42);
// => "00042 decimal = 002a hex"

// Formatter objects may be written to a string using folly::to or
// folly::toAppend (see folly/Conv.h), or by calling their appendTo(),
// str(), and fbstr() methods
std::string s = format("The only answer is {}", 42).str();
std::cout << s;
// => "The only answer is 42"

// Decimal precision usage
std::cout << format("Only 2 decimals is {:.2f}", 23.34134534535);
// => "Only 2 decimals is 23.34"
```

但 Folly 的构建是个重量级的活动，所以像我这样的开发者一般是不会采用 folly 的。虽然 folly 侧重与 Linux ， 但目前可以构建为 Windows x64 目标，使用 vcpkg 亦可安装。

Google Abseil StrFormat

在 GNK 项目中，我曾使用 fmtlib，但 clang-tidy 老是警告没有捕获异常，添加异常捕获，代码繁琐不整洁，我在考察 Abseil 之后，发现使用体验要优于 fmtlib，就将其切换到 Abseil 了。
Abseil StrFormat 只支持 C-Style 的格式化风格，格式化支持的类型可以查看如下注释：

```
// In specific, the `FormatSpec` supports the following type specifiers:
//   * `c` for characters
//   * `s` for strings
//   * `d` or `i` for integers
//   * `o` for unsigned integer conversions into octal
//   * `x` or `X` for unsigned integer conversions into hex
//   * `u` for unsigned integers
//   * `f` or `F` for floating point values into decimal notation
//   * `e` or `E` for floating point values into exponential notation
//   * `a` or `A` for floating point values into hex exponential notation
//   * `g` or `G` for floating point values into decimal or exponential
//     notation based on their precision
//   * `p` for pointer address values
//   * `n` for the special case of writing out the number of characters
//     written to this point. The resulting value must be captured within an
//     `absl::FormatCountCapture` type.
//
// NOTE: `o`, `x\X` and `u` will convert signed values to their unsigned
// counterpart before formatting.
//
// Examples:
//     "%c", 'a'                -> "a"
//     "%c", 32                 -> " "
//     "%s", "C"                -> "C"
//     "%s", std::string("C++") -> "C++"
//     "%d", -10                -> "-10"
//     "%o", 10                 -> "12"
//     "%x", 16                 -> "10"
//     "%f", 123456789          -> "123456789.000000"
//     "%e", .01                -> "1.00000e-2"
//     "%a", -3.0               -> "-0x1.8p+1"
//     "%g", .01                -> "1e-2"
//     "%p", *int               -> "0x7ffdeb6ad2a4"
```

```
//
//     int n = 0;
//     std::string s = absl::StrFormat(
//         "%s%d%n", "hello", 123, absl::FormatCountCapture(&n));
//     EXPECT_EQ(8, n);
//
// The `FormatSpec` intrinsically supports all of these fundamental C++ types:
//
// *   Characters: `char`, `signed char`, `unsigned char`
// *   Integers: `int`, `short`, `unsigned short`, `unsigned`, `long`,
//         `unsigned long`, `long long`, `unsigned long long`
// *   Floating-point: `float`, `double`, `long double`
//
// However, in the `str_format` library, a format conversion specifies a broader
// C++ conceptual category instead of an exact type. For example, `%s` binds to
// any string-like argument, so `std::string`, `absl::string_view`, and
// `const char*` are all accepted. Likewise, `%d` accepts any integer-like
// argument, etc.
```

Abseil 支持如下函数:

- StrFormat
- StrAppendFormat
- StreamFormat
- PrintF
- FPrintF
- SNPrintF

我们在实现日志库时，可以使用 `StrFormat` 格式化日志级别，时间等信息，然后使用 `StrAppendFormat` 格式化日志内容，这比 fmtlib 要方便的多。Abseil StrFormat 使用编译期检查取代运行时异常，这是让我选择的主要原因。配合 `absl::string_view` 在 GNK 一个 C++14 项目中，C++17 的使用体验非常好，字符串内存分配也减少了很多。
如果在 Windows 环境 `wchar_t` 编码环境使用 Abseil 可能效果还不如 fmtlib。由于实现了编译期类型检查，代码还是比较复杂，如果要将 Abseil StrFormat 剥离出来还比较麻烦。

字符串去格式化

如果我们使用字符串连接取代字符串格式化，字符串格式化问题则会少很多。在 Abseil 之中，有 StrCat 和 Subsitute 方案，可以连接或者组装字符串。

Substitute

实际上 Subsitute (https://github.com/abseil/abseil-cpp/blob/master/absl/strings/substitute.h) 类似 python 格式化风格，但参数只支持 0 ~ 9 个：

```
auto s=Substitute("$1 purchased $0 $2. Thanks $1!", 5, "Bob", "Apples");
```

实现 Subsitute 的关键是遍历格式化参数，解析到 `$` 后获得参数位置，然后将特定的参数转变为 `Arg` ，Arg 类型重载支持不同的基本类型，以及字符串类型，将其转变为 `absl::string_view` 数组，然后由 `SubstituteAndAppendArray` 拼接在一起。

```
class Arg {
 public:
  // Overloads for std::string-y things
  //
  // Explicitly overload `const char*` so the compiler doesn't cast to `bool`.
  Arg(const char* value)  // NOLINT(runtime/explicit)
      : piece_(absl::NullSafeStringView(value)) {}
  template <typename Allocator>
  Arg(  // NOLINT
      const std::basic_string<char, std::char_traits<char>, Allocator>&
          value) noexcept
      : piece_(value) {}
  Arg(absl::string_view value)  // NOLINT(runtime/explicit)
      : piece_(value) {}

  // Overloads for primitives
  //
  // No overloads are available for signed and unsigned char because if people
  // are explicitly declaring their chars as signed or unsigned then they are
  // probably using them as 8-bit integers and would probably prefer an integer
  // representation. However, we can't really know, so we make the caller decide
  // what to do.
  Arg(char value)  // NOLINT(runtime/explicit)
      : piece_(scratch_, 1) { scratch_[0] = value; }
  Arg(short value)  // NOLINT(*)
      : piece_(scratch_,
               numbers_internal::FastIntToBuffer(value, scratch_) - scratch_) {}
  Arg(unsigned short value)  // NOLINT(*)
      : piece_(scratch_,
               numbers_internal::FastIntToBuffer(value, scratch_) - scratch_) {}
  Arg(int value)  // NOLINT(runtime/explicit)
      : piece_(scratch_,
               numbers_internal::FastIntToBuffer(value, scratch_) - scratch_) {}
  Arg(unsigned int value)  // NOLINT(runtime/explicit)
      : piece_(scratch_,
               numbers_internal::FastIntToBuffer(value, scratch_) - scratch_) {}
  Arg(long value)  // NOLINT(*)
      : piece_(scratch_,
               numbers_internal::FastIntToBuffer(value, scratch_) - scratch_) {}
  Arg(unsigned long value)  // NOLINT(*)
      : piece_(scratch_,
               numbers_internal::FastIntToBuffer(value, scratch_) - scratch_) {}
  Arg(long long value)  // NOLINT(*)
      : piece_(scratch_,
               numbers_internal::FastIntToBuffer(value, scratch_) - scratch_) {}
  Arg(unsigned long long value)  // NOLINT(*)
      : piece_(scratch_,
               numbers_internal::FastIntToBuffer(value, scratch_) - scratch_) {}
  Arg(float value)  // NOLINT(runtime/explicit)
      : piece_(scratch_, numbers_internal::SixDigitsToBuffer(value, scratch_)) {
  }
  Arg(double value)  // NOLINT(runtime/explicit)
      : piece_(scratch_, numbers_internal::SixDigitsToBuffer(value, scratch_)) {
  }
  Arg(bool value)  // NOLINT(runtime/explicit)
      : piece_(value ? "true" : "false") {}

  Arg(Hex hex);  // NOLINT(runtime/explicit)
  Arg(Dec dec);  // NOLINT(runtime/explicit)

  // `void*` values, with the exception of `char*`, are printed as
  // "0x<hex value>". However, in the case of `nullptr`, "NULL" is printed.
  Arg(const void* value);  // NOLINT(runtime/explicit)

  Arg(const Arg&) = delete;
  Arg& operator=(const Arg&) = delete;
  absl::string_view piece() const { return piece_; }

 private:
  absl::string_view piece_;
  char scratch_[numbers_internal::kFastToBufferSize];
};
```

对于 `double` 和 `float` 则使用 `SixDigitsToBuffer` 将浮点转变为 `%g` 的格式。如果要实现固定长度输出，则可以使用 `absl::Hex` `absl::Dec` 。

```
absl::Substitute("$0$1$2$3$4 $5", //
                 absl::Dec(0), absl::Dec(1, absl::kSpacePad2),
                 absl::Dec(0xf, absl::kSpacePad2),
                 absl::Dec(int16_t{-1}, absl::kSpacePad5),
                 absl::Dec(int16_t{-1}, absl::kZeroPad5),
                 absl::Dec(0x123456789abcdef, absl::kZeroPad16));
```

对于 bool 类型，则会输出 `true` 或者 `false` 。

本质上来说 `Substitute` 是一种简化的格式化输出方案，使用编译器重载解决了运行时检查类型的麻烦，因此，这种方案安全程度比较高，效率也非常不错。但 format 支持的参数个数比较有限。absl::Substitute 同样实现了编译器参数个数检查。

StrCat

在 Abseil 之中还有一个 absl::StrCat (https://github.com/abseil/abseil-cpp/blob/master/absl/strings/str_cat.h) 用来取代字符串格式化。在 C 语言标准库中， `strcat` 用于连接字符串，参数个数是固定的，并且只支持 C-Style 字符串，而 Abseil 团队利用 C++ 变参模板特性实现了现代的 StrCat.

```
auto result =
    absl::StrCat(1, 2, 3, 4, 5, 6, 7, 8, 9, "a", "b", "c", "d", "e", "f", "g",
                 "h", "i", "j", "k", "l", "m", "n", "o", "p", "q");
// 123456789abcdefghijklmnopq
```

与 Subsititute 一样，StrCat 也是用了类似 `Arg` 的 `AlphaNum` 将字符串和基本类型按照原因（或者 Hex,Dec）拼接成特定的字符串，并且在拼接字符串时能够提前 resize 从而减少内存分配次数，达到优化的目的，在 GNK 的代码中，凡是需要连接字符串的操作，我们都使用 StrCat 来操作，避免使用 snprintf 或者 strcat 操作。 在 Privexec (https://github.com/M2Team/Privexec), Clangbuilder (https://github.com/fstudio/clangbuilder) 和 Planck (https://github.com/fcharlie/Planck) 当中，我借鉴 absl::StrCat 实现了一个宽字符版本的 `base::StringCat` (https://github.com/M2Team/Privexec/blob/master/include/strcat.hpp) 不支持 double/float，不支持 Hex，Dec，仅支持其他基本类型和 `char*` `std::wstring_view` `std::wstring` 。

异步信号安全的字符串格式化

上述现代 C++ 格式化方案通常情况下令人满意，但是当我们需要实现一个异步信号安全的格式化输出方案时，则不得不重新打算，异步信号安全指的是在信号中断的回调函数中不得调用非异步安全的函数，由于信号随时可能发生，因此，在信号中断函数中必须不存在内存分配，不能拥有互斥锁等，在 Glibc 和 musl 之中，由于 snprintf 使用了文件对象和锁调用了 `vfprintf` 则不是异步信号安全的，这很容易理解，由于 `FILE` 使用了缓存，需要使用锁保证线程安全。在 OpenBSD 当中，snprintf 实现是异步信号安全的，在 Github 上有异步信号安全的 snprintf 实现，如 c99-snprintf (https://github.com/weiss/c99-snprintf) 和 safe_snprintf (https://github.com/idning/safe_snprintf)。前面所说的 `ngx_snprintf` 也可以轻松的实现异步信号安全。

在 absl::StrFormat 中，查看源码发现 `absl::SNPrintF` 是没有内存分配的，但异步信号安全还有待考察。

在 Chromium 项目中，也有一个基于现代 C++ 实现的异步信号安全的 SafeSNPrintf (https://github.com/chromium/chromium/blob/master/base/strings/safe_sprintf.h)。在这个实现中，使用 union 包装变量，并增加类型信息，这种常见于 Json, Toml 等格式文件的解析。在格式化时，解析 format 字符串，期望的格式与输入的参数匹配类型，一旦类型匹配，则正常格式化，不匹配则退出，这种方案比 snprintf 要好的多，毕竟 snprintf 只预期输入格式正确。

```
struct Arg {
  enum Type { INT, UINT, STRING, POINTER };

  // Any integer-like value.
  Arg(signed char c) : type(INT) {
    integer.i = c;
    integer.width = sizeof(char);
  }
  Arg(unsigned char c) : type(UINT) {
    integer.i = c;
    integer.width = sizeof(char);
  }
  Arg(signed short j) : type(INT) {
    integer.i = j;
    integer.width = sizeof(short);
  }
  Arg(unsigned short j) : type(UINT) {
    integer.i = j;
    integer.width = sizeof(short);
  }
  Arg(signed int j) : type(INT) {
    integer.i = j;
    integer.width = sizeof(int);
  }
  Arg(unsigned int j) : type(UINT) {
    integer.i = j;
    integer.width = sizeof(int);
  }
  Arg(signed long j) : type(INT) {
    integer.i = j;
    integer.width = sizeof(long);
  }
  Arg(unsigned long j) : type(UINT) {
    integer.i = j;
    integer.width = sizeof(long);
  }
  Arg(signed long long j) : type(INT) {
    integer.i = j;
    integer.width = sizeof(long long);
  }
  Arg(unsigned long long j) : type(UINT) {
    integer.i = j;
    integer.width = sizeof(long long);
  }

  // A C-style text string.
  Arg(const char* s) : str(s), type(STRING) { }
  Arg(char* s)       : str(s), type(STRING) { }

  // Any pointer value that can be cast to a "void*".
  template<class T> Arg(T* p) : ptr((void*)p), type(POINTER) { }

  union {
    // An integer-like value.
    struct {
      int64_t       i;
      unsigned char width;
    } integer;

    // A C-style text string.
    const char* str;

    // A pointer to an arbitrary object.
    const void* ptr;
  };
  const enum Type type;
};
```

如果要实现 `std::string` `std::string_view` 的格式化，我们也可以在 union 中使用如下结构取代 `const char* str` ：

```
struct {
    const char *data;
    size_t len;
}stringview;
```

这样的好处是 `std::string/std::string_view` 不再需要计算长度。我们还可以使用 `%v` 按照输入参数的类型自主格式化，这样就不存在类型不匹配了。

Bela StrFormat

2019-05-17 Update: 最近我编写了一个基于 C++17 的 Windows 系统上的工具库，叫 Bela (https://github.com/fcharlie/bela)，Bela 学习了 `Chromium SafeNPrintf` ，实现了类型安全的字符串格式化，任意长度整型 `%d` ， 字符串 `%s` , 浮点(float, double) `%f` , 十六进制 `%x` `%X` ， 还有 `%p` 输出指针。支持 Pading 输出。

```cpp
// https://github.com/fcharlie/bela/blob/master/include/bela/fmt.hpp
template <typename... Args>
ssize_t StrFormat(wchar_t *buf, size_t N, const wchar_t *fmt, Args... args) {
  const format_internal::FormatArg arg_array[] = {args...};
  return format_internal::StrFormatInternal(buf, N, fmt, arg_array,
                                            sizeof...(args));
}

template <size_t N, typename... Args>
ssize_t StrFormat(wchar_t (&buf)[N], const wchar_t *fmt, Args... args) {
  // Use Arg() object to record type information and then copy arguments to an
  // array to make it easier to iterate over them.
  const format_internal::FormatArg arg_array[] = {args...};
  return format_internal::StrFormatInternal(buf, N, fmt, arg_array,
                                            sizeof...(args));
}

template <typename... Args>
std::wstring StrFormat(const wchar_t *fmt, Args... args) {
  const format_internal::FormatArg arg_array[] = {args...};
  return format_internal::StrFormatInternal(fmt, arg_array, sizeof...(args));
}

// Fast-path when we don't actually need to substitute any arguments.
ssize_t StrFormat(wchar_t *buf, size_t N, const wchar_t *fmt);
std::wstring StrFormat(const wchar_t *fmt);
template <size_t N>
inline ssize_t StrFormat(wchar_t (&buf)[N], const wchar_t *fmt) {
  return StrFormat(buf, N, fmt);
}
```

支持的字符串类型：

- `std::wstring`
- `std::wstring_view`
- `const wchar_t *`
- `wchar_t *`
- `std::u16string`
- `std::u16string_view`
- `const char16_t *`
- `char16_t *`

以及以下会转换为 UTF-16 的 UTF-8 字符串类型：

- `std::string`
- `std::string_view`
- `const char *`
- `char *`

基于 `Bela Format` 我还编写了 `bela::FPrintF` 将格式化的数据输出到控制台终端或者文件，当输出到 Conhost 时，则会以 UTF-16 编码输出，若输出到文件或者 Cygwin 终端时，则会转为 UTF-8 输出。

```
// https://github.com/fcharlie/bela/blob/master/include/bela/stdwriter.hpp
ssize_t StdWrite(FILE *out, std::wstring_view msg);

template <typename... Args>
ssize_t FPrintF(FILE *out, const wchar_t *fmt, Args... args) {
  const format_internal::FormatArg arg_array[] = {args...};
  auto str =
      format_internal::StrFormatInternal(fmt, arg_array, sizeof...(args));
  return StdWrite(out, str);
}

inline ssize_t FPrintF(FILE *out, const wchar_t *fmt) {
  auto str = StrFormat(fmt);
  return StdWrite(out, str);
}
```

一个简单的例子（此代码能够在 Windows Terminal 和 Mintty 等终端上正常显示 emoji）：

```
// https://github.com/fcharlie/bela/blob/master/test/fmt/main.cc
#include <bela/stdwriter.hpp>

int wmain(int argc, wchar_t **argv) {
  auto ux = "\xf0\x9f\x98\x81 UTF-8 text \xE3\x8D\xA4"; // force encode UTF-8
  wchar_t wx[] = L"Engine \xD83D\xDEE0";
  bela::FPrintF(stderr, L"Argc: %d Arg0: %s W: %s UTF-8: %s __cplusplus: %d\n",
                argc, argv[0], wx, ux, __cplusplus);
  return 0;
}
```

结尾

在格式化函数的过程中，C++ 的不足在于没有反射，从而无法很好的获得对象的类型，这样传统的格式化方案就容易出现问题，而是用变参模板，在复杂的编码技巧加成后，使用编译器的检查能够很好的实现类型安全高效的格式化，但是也存在一个问题，那就是程序编译后提交较大，但都到了 9012，只要不超过 Clang 的体积都是能接受的。