Variadic templates and fold expressions

Folding, unfolding, such and such...

Prerequisites

You should

- Be familiar with templates in general and metaprogramming,
- Have used recursion, and template specialization,
- Have seen a lambda once,
- Heard about `if constexpr`,
- Have open heart to C++ and tasted VDD (Vodka Driven Development.)

printf

```
#include <stdio.h>
int printf(const char * restrict format, ...);
```

- Prone to UB if inconsistent types used,
- Loss of type information,
- The compiler might not be able to optimize it,
- No customization,
- Security issues,
- Very simple and has nice syntax.

iostreams

```
#include <iostream>
std::cout << “Sup?” << 42;
std::ostream & operator << (std::ostream &, Type);
```

- No loss of type information — type-safe,
- The compiler has more freedom to apply optimizations,
- Easy to extend,
- Doesn’t imply security concerns,
- Awkward and verbose syntax: `iomanip`-s.
- Performance problems — has to be synced with fstreams; tunable tho.

What if we want the best of both

We might want to

- Enjoy `printf`'s format string and `iostreams`' type-safety, or
- Maintain legacy code...

What if we want the best of both

We might want to

- Enjoy `printf`'s format string and `iostreams`' type-safety, or
- Maintain legacy code, or
- Slow down compilation time to have time for side-projects.

Step #1: Template

Variadic templates is a way to preserve type information.

```
template <typename ...Args>
void printf(char const *fmt, Args ...args) {
  ::printf(fmt, args...);
}
```

Step #1: Question

Variadic templates is a way to preserve type information passing any number of arguments.

```
template <typename ...Args>
void printf(char const *fmt, Args ...args) {
  ::printf(fmt, args...);
}
```

What's the deal with that?

Step #2: Validation

What if we have a function to validate.

```
template <typename Fmt, typename ...Args>
void printf(Fmt fmt, Args ...args) {
  validate(fmt, std::make_tuple(args...));
  ::printf(fmt, args...);
}
```

Step #2: Question

What if we have a function to validate.

```
template <typename Fmt, typename ...Args>
void printf(Fmt fmt, Args ...args) {
  validate(fmt, std::make_tuple(args...));
  ::printf(fmt, args...);
}
```

What's `Fmt` and how does `validate` work?

True string literals

- P0424R2: String literals as non-type template parameters
- N3599: Literal operator templates for strings

```
template <typename CharT, CharT ...Cs>
constexpr auto operator ""_literal();
```

Implemented in GCC and Clang as an extension.
Hopefully, it, or something similar, will make into the standard C++2a.

So what's the `Fmt`?

Think of it as

```
using Fmt = string_literal<char, ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’>;
```

or

```
typedef string_literal<char, ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’> Fmt;
```

```
Fmt fmt = “Hello”_literal;
validate(“Hello, %s\n”_literal, name);
```

```
template <typename CharT, CharT ...Cs>
struct string_literal {};
```

```
template <typename CharT, CharT ...Cs>
constexpr auto operator ""_literal() {
  return string_literal<CharT, Cs...>{};
}
```

Aha, and what's inside `validate`?

There's a loop inside…

```
template <typename Fmt, std::size_t N, std::size_t FmtI, typename Args, std::size_t ArgsI>
struct validate_helper {
  constexpr bool loop(Fmt &&fmt, Args &&args) const {
    if constexpr (std::get<FmtI>(fmt.array()) == '%') {
      constexpr auto const spec_off = specifier_offset<>(fmt);
      constexpr auto const spec = std::get<FmtI + spec_off>(fmt.array());

      if constexpr (spec == '%') {
        return validate_helper<Fmt, N, FmtI + spec_off + 1, Args, ArgsI>{}.loop(
            std::forward<Fmt>(fmt), std::forward<Args>(args));
      } else if constexpr (contains<'d', 'i'>(spec)) {
        validate_integer<decltype(std::get<ArgsI>(args))>();
        return validate_helper<Fmt, N, FmtI + spec_off + 1, Args, ArgsI + 1>{}.loop(
            std::forward<Fmt>(fmt), std::forward<Args>(args));
```

Fold-expressions (Reference)

Allows you to write expressions around template parameter packs without having to use recursive templates.

```
template <typename ...Args>
bool all(Args ...args) {
  return (... && args);
  return (true && ... && args);
}
```

```
all(true, false, true);  // returns false
```

```
template <typename ...Args>
auto avg(Args ...args) {
  return (... + args) / sizeof... (Args);
}
```

```
avg(1.0, 2.0, 3.0); // returns 2.0
```

Fold-expressions: lambda

You can use lambdas...

```
template <auto ...Specs, typename T>
constexpr bool contains(T c) {
  return (... || [c](auto spec) { return c == spec; }(Specs));
}

contains<'f', 'F', 'g', 'G'>('d');  // returns false
```

Fold-expressions: more lambdas

Can be combined with `std::variant`, inheritance, and type deduction guides...

```
template <typename ...Ops>
struct overloads: Ops... {
  using Ops::operator ()...;
};

template <typename ...Ops>
overloads(Ops...) -> overloads<Ops...>;
```

```
auto visitor = overloads{
  [](int x)  { "got int %d\n"_tsprintf(x); },
  [](auto x) { "got something %q\n"_tsprintf(x); }
};

std::variant<int, float> v{3.14};
std::visit(visitor, v);
```

Fold-expressions: lambdas everywhere

Can be combined with std::variant, inheritance, and type deduction guides...

```
template <typename ...Ops>
struct overloads: Ops... {
  // § 8.4.5.1 [expr.prim.lambda.closure]
  // (1) The type of a lambda-expression (which is also the type of the closure object) is a unique,
  // unnamed non-union class type, called the closure type, whose properties are described below.
  // (3) The closure type for a non-generic lambda-expression has a public inline function call operator…
  //
  // § 13.2 Member name lookup [class.member.lookup]
  // (6.2) — Otherwise, if the declaration sets of S(f,Bi) and S(f,C) differ, the merge is ambiguous:
  // the new S(f,C)  is a lookup set with an invalid declaration set and the union of the subobject sets.
  // In subsequent merges,  an invalid declaration set is considered different from any other.
  //
  using Ops::operator ()...;
};
```

```
struct X { void fn(int); };
struct Y { void fn(char const *); };

struct Z: X, Y {
  // using X::fn;
  // using Y::fn;
};

Z z;
z.fn(42);  // ambiguous call
```

```
// Deduction guide: http://en.cppreference.com/w/cpp/language/class_template_argument_deduction#User-defined_deduction_guides
template <typename ...Ops>
overloads(Ops...) -> overloads<Ops...>;
```

Q&A

https://github.com/deni64k/tsprintf

~~Example online (the link is huge)~~, or paste this to godbolt

https://github.com/deni64k

https://twitter.com/deni64k