hop

a language to design function-overload-sets

Bullet list

- One
- Two
- Three

Fragmented list

- One
- Two
- Three

outline

- prequel: homogeneous variadic functions
  - what are we talking about
  - historic view
  - best practices and limitations
  - P1219r2: Homogeneous variadic function parameters
  - solving the overload problem
  - what about concepts?
- hop: function-parameter building-blocks
  - applications of hop
  - how does all this work

homogeneous variadic functions (hfvs): what are we talking about

```
void logger("This", "is", "a", "log", message);
double max(0, d1, d2, d3, d4, d5);
Array slice(s1, s2, s3);
```

- as if a function has overloads for any (non-zero) number of arguments, but all of the **same type**
- should also work for overloads of hvfs, especially needed for ctors and `operator()`

homogeneous variadic functions: C++98

write overloads up to a given number

```
void f(string s1);
void f(string s1, string s2);
void f(string s1, string s2, string s3);
void f(string s1, string s2, string s3, string s4);
void f(string s1, string s2, string s3, string s4, string s5);
```

- implementation strategies
  - reduce parameters to a single one, then call `f(string const& s1)`
  - inductive: call predecessor, then process with remaining argument
  - put arguments into container and call helper
- pros
  - explicit: easy to write and understand
  - no fancy stuff (e.e. templates required)
  - overload resolution works out-of-the-box
- cons
  - very explicit
  - only up to a given number of arguments
  - DRY, DRY, DRY

homogeneous variadic functions: C++98

Boost.Preprocessor

```
#define PUSHBACK_ARG(Z, N, _)   vs.push_back( s ## N );

#define GENRATE_OVERLOAD_OF_F(Z, N, _)           \
void f(BOOST_PP_ENUM_PARAMS(N, string s)) {      \
    vector<string> vs;                           \
    BOOST_PP_REPEAT(N, PUSHBACK_ARG, _)          \
    /* ... */                                    \
}

BOOST_PP_REPEAT_FROM_TO(1, 6, GENRATE_OVERLOAD_OF_F, _)
```

- implementation strategies
  - reduce parameters to a single one, then call `f(string const& s1)`
  - inductive: call predecessor, then process with remaining argument
  - put arguments into container and call helper
- pros
  - overload resolution works out-of-the-box
- cons
  - declaration/definition are hidden in macros
  - hard to read
  - even harder to debug
  - only up to a given number of arguments

homogeneous variadic functions: C++11

variadic templates

```
#define PUSHBACK_ARG(Z, N, _)   vs.push_back( s ## N );

#define GENRATE_OVERLOAD_OF_F(Z, N, _)          \
void f(BOOST_PP_ENUM_PARAMS(N, string s)) {     \
    vector<string> vs;                          \
    BOOST_PP_REPEAT(N, PUSHBACK_ARG, _)         \
    /* ... */                                   \
}

BOOST_PP_REPEAT_FROM_TO(1, 6, GENRATE_OVERLOAD_OF_F, _)
```

- implementation strategies
  - reduce parameters to a single one, then call `f(string const& s1)`
  - reduce parameters to a single one, then call `f(string const& s1)`
  - reduce parameters to a single one, then call `f(string const& s1)`
  - reduce parameters to a single one, then call `f(string const& s1)`
  - inductive: call predecessor, then process with remaining argument
  - put arguments into container and call helper
- pros
  - overload resolution works out-of-the-box
- cons
  - declaration/definition are hidden in macros
  - hard to read
  - even harder to debug
  - only up to a given number of arguments