Named Arguments

From Scratch

Richard Powell

v0.2

Library in a Week 2017

- **A**rthur O’Dwyer
- **R**ichard Powell
- **G**ašper Ažman
- **O**din Holmes

Given:

```
int foo(int a, float b, std::string const& c);
```

Wouldn’t it be cool to be able to write:

```
foo(c = "hello world", b = 0.5, a = 10);
```

Why?

Learning

Thinking

Innovating

Tools

- C++17
- Compiler Explorer
- Boost Hana:
  - “Your standard library for metaprogramming”
  - “Hana is a header-only library for C++ metaprogramming suited for computations on both types and values.”

How?

```
foo(c = "hello world", b = 0.5, a = 10);
```

Construct

```
{ [ c, "hello world" ], [ b, 0.5 ], [ a, 10 ] }
```

Extract

```
{ 10, 0.5, "hello world" }
```

Unpack

```
int foo(int a, float b, std::string const& c);
```

Coding

Intro

```
foo(c = "hello world", b = 0.5, a = 10);
```

Construct

{ [*c*, "hello world"], [*b*, 0.5], [*a*, 10] }

Extract

{ 10, 0.5, "hello world" }

Unpack

```
int foo(int a, float b, std::string const& c);
```

Coding

Unpack

```
foo(c = "hello world", b = 0.5, a = 10);
```

Construct

{ [***c***, "hello world"], [***b***, 0.5], [***a***, 10] }

Extract

{ 10, 0.5, "hello world" }

Unpack

```
int foo(int a, float b, std::string const& c);
```

```
foo(c = "hello world", b = 0.5, a = 10);
```

Construct

```
{ "hello world", 0.5, 10 }
```

Extract

```
{ 10, 0.5, "hello world" }
```

Unpack

```
int foo(int a, float b, std::string const& c);
```

Coding

Reorder

Coding

Refactor1

Never have to write `template`!

```
template<typename T>
constexpr auto foo(T arg)
{
    ...
}
```

```
constexpr auto foo = [](auto args)
{
    ...
};
```

Coding

Refactor2

```
foo(c = "hello world", b = 0.5, a = 10);
```

Construct

```
{ "hello world", 0.5, 10 }
```

Extract

```
{ 10, 0.5, "hello world" }
```

Unpack

```
int foo(int a, float b, std::string const& c);
```

```
foo(c = "hello world", b = 0.5, a = 10);
```

Construct

{ [***c***, "hello world"], [***b***, 0.5], [***a***, 10] }

Extract

{ 10, 0.5, "hello world" }

Unpack

```
int foo(int a, float b, std::string const& c);
```

```
foo(c = "hello world", b = 0.5, a = 10);
```

Construct

```
{ [ 2_c, "hello world" ], [ 1_c, 0.5 ], [ 0_c, 10 ] }
```

Extract

```
{ 10, 0.5, "hello world" }
```

Unpack

```
int foo(int a, float b, std::string const& c);
```

Coding

Map

```
foo(c = "hello world", b = 0.5, a = 10);
```

Construct

```
{ [ 2_c, "hello world" ], [ 1_c, 0.5 ], [ 0_c, 10 ] }
```

Extract

```
{ 10, 0.5, "hello world" }
```

Unpack

```
int foo(int a, float b, std::string const& c);
```

```
foo(c = "hello world", b = 0.5, a = 10);
```

Construct

```
{ [ "c"_arg, "hello world" ], [ "b"_arg, 0.5 ], [ "a"_arg, 10 ] }
```

Extract

```
{ 10, 0.5, "hello world" }
```

Unpack

```
int foo(int a, float b, std::string const& c);
```

User Defined Literals

Literal operators

The function called by a user-defined literal is known as *literal operator* (or, if it's a template, *literal operator template*). It is declared just like any other function or function template at namespace scope (it may also be a friend function, an explicit instantiation or specialization of a function template, or introduced by a using-declaration), except for the following restrictions:

The name of this function can have one of the two forms:

operator "" *identifier*

https://en.cppreference.com/w/cpp/language/user_literal

```
template<typename CharT, CharT... Chars>
constexpr auto operator"" identifier();
```

GCC Extension

Coding

udl

Coding

Compose

```
foo(c = "hello world", b = 0.5, a = 10);
```

Construct

```
{ [ "c"_arg, "hello world" ], [ "b"_arg, 0.5 ], [ "a"_arg, 10 ] }
```

Extract

```
{ 10, 0.5, "hello world" }
```

Unpack

```
int foo(int a, float b, std::string const& c);
```

```
foo(c = "hello world", b = 0.5, a = 10);
```

Construct

```
{ [ "c"_arg, "hello world" ], [ "b"_arg, 0.5 ], [ "a"_arg, 10 ] }
```

Extract

```
{ 10, 0.5, "hello world" }
```

Unpack

```
int foo(int a, float b, std::string const& c);
```

```
my_foo([ "c"_arg, "hello world" ], [ "b"_arg, 0.5 ], [ "a"_arg, 10 ]);
```

Construct

```
{ [ "c"_arg, "hello world" ], [ "b"_arg, 0.5 ], [ "a"_arg, 10 ] }
```

Extract

```
{ 10, 0.5, "hello world" }
```

Unpack

```
int foo(int a, float b, std::string const& c);
```

Coding

Adapt

```
my_foo([ "c"_arg, "hello world" ], [ "b"_arg, 0.5 ], [ "a"_arg, 10 ]);
```

Construct

```
{ [ "c"_arg, "hello world" ], [ "b"_arg, 0.5 ], [ "a"_arg, 10 ] }
```

Extract

```
{ 10, 0.5, "hello world" }
```

Unpack

```
int foo(int a, float b, std::string const& c);
```

```
my_foo("c"_arg = "hello world", "b"_arg = 0.5, "a"_arg = 10);
```

Construct

```
{ [ "c"_arg, "hello world" ], [ "b"_arg, 0.5 ], [ "a"_arg, 10 ] }
```

Extract

```
{ 10, 0.5, "hello world" }
```

Unpack

```
int foo(int a, float b, std::string const& c);
```

["c"_arg, "hello world"] → hana::pair<hana::string, T>

```
template<typename CharT, CharT... Chars>
constexpr auto operator"" _arg() -> hana::string<Chars...>;
```

```
[ "c"_arg, "hello world" ]   ⟶   hana::pair<hana::string, T>
```

```
template<typename CharT, CharT... Chars>
constexpr auto operator"" _arg() -> named_param<Chars...>;
```

```
hana::pair<hana::string, T> named_param::operator=();
```

```
"c"_arg = "hello world"   ⟶   hana::pair<hana::string, T>
```

Coding

named_param
pre-baked

```
template<typename CharT, CharT... Chars>
constexpr auto operator"" _arg() -> hana::string<Chars...>;
```

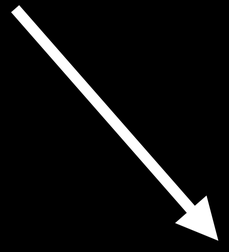
```
arg_spec =
{
  "a"_arg,
  "b"_arg,
  "c"_arg,
}
```

```
hana::tuple<
    hana::string,
    hana::string,
    hana::string
>
```

```
template<typename CharT, CharT... Chars>
constexpr auto operator"" _arg() -> named_param<Chars...>;
```

```
arg_spec =
{
  "a"_arg,
  "b"_arg,
  "c"_arg,
}
```

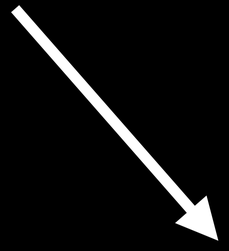
~~hana::tuple<~~
~~hana::string,~~
~~hana::string,~~
~~hana::string~~
~~>~~



```
hana::tuple<
    named_param,
    named_param,
    named_param
>
```

```
template<typename CharT, CharT... Chars>
constexpr auto operator"" _arg() -> named_param<Chars...>;
```

```
arg_spec =
{
  "a"_arg,
  "b"_arg,
  "c"_arg,
}
```



```
hana::tuple<
    hana::string,
    hana::string,
    hana::string
>
```

```
hana::tuple<
    named_param,
    named_param,
    named_param
>
```

Coding

get_names

```
my_foo("c"_arg = "hello world", "b"_arg = 0.5, "a"_arg = 10);
```

Construct

```
{ [ "c"_arg, "hello world" ], [ "b"_arg, 0.5 ], [ "a"_arg, 10 ] }
```

Extract

```
{ 10, 0.5, "hello world" }
```

Unpack

```
int foo(int a, float b, std::string const& c);
```

Why?

Learning

Thinking

Innovating

Helpful links

- Python Named Arguments
  - http://www.diveintopython.net/power_of_introspection/optional_arguments.html
- Compiler Explore:
  - https://gcc.godbolt.org
  - https://github.com/mattgodbolt/compiler-explorer
  - Jason Turner, “C++ Weekly: Episode 83 Installing Compiler Explorer” https://www.youtube.com/watch?v=I2cKVRzJhS0
- Hana:
  - http://www.boost.org/doc/libs/1_61_0/libs/hana/doc/html/index.html
- Argo:
  - https://github.com/rmpowell77/LIAW_2017_param