

This repository


Search


Pull requests


Issues


Marketplace

Gist







 splinterofchaos / fu

Watch

7

Star

40

Fork

4

Code

Issues1

Pull requests1

Projects0

Wiki

Insights

Branch: master **fu / include / fu /**

Create new fileUpload filesFind fileHistory

splinterofchaos utility: add ref/cref		Latest commit f0ebb43 on 15 Mar 2015
..		
logic	logic: learn either and both	2 years ago
make	auto: fu/make/make.h	2 years ago
tuple	meta: learn all<bool...>	2 years ago
README.md	README: add header	2 years ago
basic.h	to_functor -> forwarder	2 years ago
fu.h	Replace some ToFunctors/fn calls with invoke(f).	3 years ago
functional.h	learn sequence	2 years ago
invoke.h	invoke: split into invoke_member, maybe_deref	2 years ago
iseq.h	meta: use auto return	2 years ago
list.h	Add missing files: fu/list.h, fu/tuple.h	2 years ago
logic.h	add logic module	2 years ago
meta.h	meta: learn all<bool...>	2 years ago
tuple.h	Add missing files: fu/list.h, fu/tuple.h	2 years ago
utility.h	utility: add ref/cref	2 years ago

README.md

# Documentation

FU must be compiled with gcc 4.9 or later, or clang 3.5, with the flag `-std=c++14` . MSVC is not supported for lack of C++14 features.

The subdirectories contain modules with their own documentation, but can be included from the associated file in the main directory. For example, `fu/tuple.h` will include `fu/tuple/basic.h` and `fu/tuple/tuple.h` . See `fu/tuple/README.md` for its documentation.

## "fu/fu.h"

This file includes all other FU headers. Use this is you don't know what components you need, or you need them all (albeit unlikely).

## "fu/invoke.h"

This file contains an implementation of `n3727`'s `std::invoke` .

`invoke(f, x...)` simply calls `f` with the arguments, `x...`. `f` may be a regular function, member function pointer, or member object pointer. `invoke(f)` will simply call `f()`.

```
void f();
invoke(f); // calls f()

std::string s;
invoke(std::string::size, s); // calls s.size()
```

## "fu/basic.h"

This file contains miscellaneous utilities that the rest of the library builds off of. Components may be moved from this file into more granular units so it should not be included directly. The following are included as of this writing.

### identity(x)

Returns `x`

### forwarder(f)

If given a member pointer, `std::mem_fn` may be used to create a function object, but that doesn't help when one doesn't know if the function is a member pointer. `forwarder(f)` can be used to create an invocable object, whether `f` is a member pointer, regular function, or function object.

```
// Collecting the lengths of some strings:
std::transform(strs.begin(), strs.end(), std::back_inserter(lengths),
               forwarder(&std::string::size));
```

### closure(f,x...), part(f,x...), rclosure(f,y...), rpart(f,y...)

`closure(f, x...)` creates a partial function, `c`, such that `c(y...)` returns the result of `f(x..., y...)`. The arguments will be copied, but `std::ref` can be used to avoid this.

`part(f, x...)` works in the same way, except that it captures its arguments by perfectly forwarding. Because `part` may create references to temporaries, `closure` should be preferred if one is not sure.

`rpart` and `rclosure` apply the arguments at the right-hand side.

```
auto plus_one = closure(std::plus<>{}, 1);
plus_one(1); // equals two

int one = 1;
auto plus_ref = closure(std::plus<>{}, std::ref(one));
plus_ref(1); // returns two
one++;
plus_ref(1); // returns three

auto minus_one = rclosure(std::minus<>{}, 1);
minus_one(10); // computes 10 - 1
```

### multary(f)

Creates a function, `m`, such that `m(x)` returns a partial application of `f` and `x`. Applying `m` with more than one argument will not partially apply.

To create a function that will partially apply up to `n` arguments, use `multary_n<n>(f)`.

```

auto plus = multary(std::plus<>{});
auto plus_one = plus(1);
plus_one(1); // returns 2

void _g(int, int, int);
auto g = multary_n<2>(_g);
auto g_one = g(1); // same as multary_n<1>(closure(g, 1))
auto g_one_two = g_one(2); // same as closure(g, 1, 2)
g_one_two(3); // calls _g(1,2,3)

```

## "fu/functional.h"

### pipe(x, f, g, h...)

Computes `h(g(f(x)))`, or: applies `x` to each function, left-to-right.

### lassoc(f) and rassoc(f)

Makes `f` a left- or right-associative function so that it can be invoked with an arbitrary number of arguments.

`lassoc(f, x, y)` will compute `f(x, y)`, but generally the partially applied form, `lassoc(f)` is the most useful.

```

auto sum = lassoc(std::plus<>{});
sum(1, 2, 3, 4); // computes: ((1+2) + 3) + 4
rassoc(std::minus<>{}, 1, 2, 2); // computes: 1 - (2 - 2)

```

### transitive(binary, join = std::logical\_and)

Makes a function that preserves transitivity. The function, `binary` must preserve transitivity, `join` logically connects the results. FIXME: better explanation.

```

auto less = transitive(std::less<>{});
less(1, 2, 3, 4); // computes: 1 < 2 && 2 < 3 && 3 < 4
transitive(std::greater<>{}, 3, 2, 1); // computes: 3 > 2 && 2 > 1

```

### overload(f,g...) and ranked\_overload(f,g...)

Constructs a function object overloaded on `f` and `g...`. Because this may cause ambiguities, `ranked_overload` can be used so that `f` will be chosen by default, and `g...` if SFINAE forbids.

```

auto o = overload([](int x) { return x + 10; },
                  [](std::string s) { return s + "1"; });
o(1); // returns: 11
o("1"); // returns: "11"

auto ro = ranked_overload([](auto x) { std::cout << x; },
                           [](auto x) { for (auto&& y : x) std::cout << y; });
ro(1); // prints "1"
ro(std::vector<int>{1, 2, 3}); // prints "123"

```

### fix(f)

`fix(f)` can be used to define a recursive function without having it refer to itself using a concept called "[fixed point combinator](#)". This can be useful for lambdas, which may not be referred to in the statement that declares them.

Note that due to a [bug in clang](#), the result of `f` may not be constexpr.

```
constexpr auto pow2 = fix([](auto rec, int x) -> int {
    return x > 2 ? 2 * rec(x-1) : 1;
});
pow2(1); // returns 2
pow2(3); // returns 8

constexpr auto fact = fix([](auto rec, int x) -> int {
    return x > 2 ? x * rec(x-1) : x;
});
```

## compose(f,g), ucompose(f,g), compose\_n(f,g)

Many useful forms of composition exist, but `compose(f,g)` is the most general. It returns a function, `c`, that takes two tuples, `{x...}` and `{y...}`, such that `c({x...}, {y...})` computes `f(g(x...), y...)`. For most instances, `u = ucompose(f,g)` is much simpler; `u(x,y...)` computes `f(g(x), y...)` –it is short for "unary composition". It assumes that `g` is unary, but often it may not be. `compose_n<n>(f,g)` sends the first `n` arguments to `g` and the rest to `f`.

```
void f(int, int, int);
void g(int, int);
constexpr auto fg = compose(f,g);
constexpr auto fg2 = compose_n<2>(f,g)

using fu::tpl::tuple;
fg(tuple(1,2), tuple(3,4)); // computes: f(g(1,2), 3, 4)
fg2(1,2,3,4);              // computes: f(g(1,2), 3, 4)

void h(int);
constexpr auto fh = ucompose(f,h);
fh(1,2,3); // computes: f(h(1), 2, 3)
```

## proj(f,pf), proj\_less(pless), rproj(f,pf) and lproj(f,pf)

The `proj` family constructs projection functions taking a function, `f`, and a projection to `f`, `pf`. `l` and `rproj` only project the left- or right-hand arguments. Because it is the most common projection, `proj_less = proj(std::less<>())`.

```
// To sort a list by applying `*it1 < *it2`:
std::sort(first, last, proj(std::less<>(), f));
// equivalent:
std::sort(first, last, proj_less(f));

// Accumulate the sum of sizes of a list of std::strings.
std::accumulate(first, last, 0, rproj(std::plus<>(), &std::string::size));
```

## split(f,l,r) and join(f,l,r)

`split(f,l,r)` returns a function that takes a single argument, `x`, and computes `f(l(x), r(x))`. `join(f,l,r)` takes exactly two arguments, `x` and `y`, and computes `f(l(x), r(y))`.

## flip(f)

`flip(f)` reverses the order of arguments applied to `f`.

```
static_assert(fu::flip(fu::sub)(y,x) == fu::sub(x,y));

auto h(A, B, C, D);
flip(h, d, c, b, a); // invokes: h(a,b,c,d)
```

## constant(x)

`constant(x)` returns a nullary function (takes no arguments) that always returns `x`,

```
constexpr auto one = constant(1);
one(); // returns 1
```

## "fu/utility.h"

Implements transparent function objects for operators like `+`, `-`, `%`, etc., as well as utilities for containers like `size`, `push_back`, etc..

## size, index, back, front

These functions take a container or array as arguments.

```
int xs[5] = {0,1,2,3,4};
std::list<int> ys = {0,1,2,3,4};

fu::size(xs); // returns 5
fu::size(ys); // returns 5
fu::front(xs); // returns 0
fu::back(ys); // returns 4
```

## push\_back, push\_front, insert

These functions call the associated member function, but take the object to insert as the first argument.

```
std::set<int> ys(4);
fu::insert(10, ys); // ys = {10}

int xs[] = {0,1,2};
std::for_each(std::begin(ys), std::end(ys), fu::flip(fu::insert, std::ref(ys)));
// ys = {0, 1, 2, 10}
```

## min, max

```
auto five = fu::max(1,2,3,4,5);
auto one = fu::min(1,2,3,4,5);

constexpr auto one_or_more = fu::max(1);
constexpr auto ten_or_less = fu::min(10);
```

## add, sub(tract), mult(iply) div(ide), rem(ainder), lshift, rshift, or\_, and\_, xor\_bit\_or

These functions are all both `multary` and left-associative.

```
constexpr auto plus_one = add(1);
constexpr int two = plus_one(1);
constexpr int ten = add(1,2,3,4);
constexpr bool yes = or_(false, false, false, true);
constexpr bool no = and_(true, false, true, false);
```

Also implemented: xor\_eq, add\_eq, sub\_eq, mult\_eq, div\_eq, rem\_eq,

## less, greater, eq, neq, less\_eq, greater\_eq

These function are `multary` and `transitive`.

```
static_assert(less(1,2,3,4), "computes '1 < 2 && 2 < 3 && 3 < 4'");  
static_assert(less_eq(1,2,2,3), "computes '1 <= 2 && 2 <= 2 && 2 <= 3'");  
static_assert(eq(1)(1,1), "computes '1 == 1 && 1 == 1 && 1 == 1'");
```

