zero-overhead C++17 currying & partial application

23 january 2017

c++ c++17 lambda functional curry partial

As I mentioned in my previous article (https://vittorioromeo.info/index/blog/passing_functions_to_functions.html) many features introduced in the latest C++ standards allow *functional patterns* to thrive in your codebase. Two ideas from that programming paradigm that I really like are **currying** (https://en.wikipedia.org/wiki/Currying) and **partial application** (https://en.wikipedia.org/wiki/Partial_application).

In this article we're going to:

- Introduce and briefly explain the two aforementioned concepts.
- Write a generic `constexpr` zero-overhead `curry` function in C++17.
- Analyze the generated assembly of the implementation to prove the lack of overhead.

Currying

Let's begin by explaining currying.

> In mathematics and computer science, **currying** is the technique of translating the evaluation of a function that takes multiple arguments *(or a tuple of arguments)* into evaluating a sequence of functions, each with a single argument.

(from Wikipedia) (https://en.wikipedia.org/wiki/Currying)

The following `add3` function is an example of a *non-curried* function, as its *arity* (https://en.wikipedia.org/wiki/Arity) is 3.

```
auto add3(int a, int b, int c)
{
    return a + b + c;
}

add3(1, 2, 3); // Returns `6`.
```

We can *curry* it by returning nested *lambda expressions* (http://en.cppreference.com/w/cpp/language/lambda).

```
auto curried_add3(int a)
{
    return [a](int b)
    {
        return [a, b](int c)
        {
            return a + b + c;
        };
    };
}

curried_add3(1)(2)(3); // Returns `6`.
```

wandbox example (http://melpon.org/wandbox/permlink/HGACYHh1sV3knQGZ)

As you can see, the arity of every lambda is 1. This pattern is useful because it allows developers to intuitively *bind* arguments incrementally until the last one. If you're finding yourself constantly using the same arguments except one in a series of function calls, currying avoids repetition and increases readability.

```
auto add2_one = curried_add3(1);
auto add1_three = add2_one(2);

add1_three(3); // Returns `6`.
add1_three(4); // Returns `7`.
add1_three(5); // Returns `8`.
```

wandbox example (http://melpon.org/wandbox/permlink/7w4RUHqHEL1Y7Yx1)

Basically, `add1_three(5)` is equivalent to `add2_one(2)(5)`, which is equivalent to `curried_add3(1)(2)(5)`.

A slightly more realistic example could involve `std::find` (http://en.cppreference.com/w/cpp/algorithm/find):

```
std::vector<std::string> names{/* ... */}

auto find_in_names =
    curried_find(std::begin(names))(std::end(names));

auto jack = find_in_names("Jack");
auto rose = find_in_names("Rose");
```

In the above code snippet some repetition between `std::find` invocations is cleanly avoided thanks to *currying*.

(This short article (http://cukic.co/2013/08/07/curry-all-over-the-c11/) by Ivan Čukić has some additional interesting examples of currying in C++.)

Partial application

> In computer science, **partial application** (or **partial function application**) refers to the process of fixing a number of arguments to a function, producing another function of smaller arity.
>
> *(from Wikipedia)* (https://en.wikipedia.org/wiki/Partial_application)

Despite them being two separate concepts, *partial application* is very similar to *currying*. Even though I couldn't find a formal confirmation anywhere, I believe that thinking about *partial application* as a "generalized form of *currying*" can be helpful: instead of binding one argument and getting $(arity - 1)$ unary functions back, we can bind n arguments at once and get another *partially-applicable* function with $(arity - n)$ arity.

Imagine we had add a `partial_add3` function which allowed *partial application* to sum three numbers:

```
partial_add3(1, 2, 3); // Returns `6`.
partial_add3(1)(2, 3); // Returns `6`.
partial_add3(1, 2)(3); // Returns `6`.
partial_add3(1)(2)(3); // Returns `6`. (Currying!)
```

As you can see, we can decide how many arguments to bind *(including zero)*. We could easily implement this in C++17 using *recursion*, *generic lambdas* (http://en.cppreference.com/w/cpp/language/lambda), `if constexpr(...)` (http://en.cppreference.com/w/cpp/language/if#Constexpr_If), and *variadic templates* (http://en.cppreference.com/w/cpp/language/parameter_pack). *(We'll also use a fold expression (http://en.cppreference.com/w/cpp/language/fold) to compute the sum.)*

```
template <typename... Ts>
auto partial_add3(Ts... xs)
{
    static_assert(sizeof...(xs) <= 3);

    if constexpr (sizeof...(xs) == 3)
    {
        // Base case: evaluate and return the sum.
        return (0 + ... + xs);
    }
    else
    {
        // Recursive case: bind `xs...` and return another
        return [xs...](auto... ys)
        {
            return partial_add3(xs..., ys...);
        };
    }
}
```

wandbox example (http://melpon.org/wandbox/permlink/AFmdO0Cpkt5zRcJC)

Writing code that enables *currying* and *partial application* for every function is cumbersome. Let's write a generic `curry` function that, given a function object `f`, returns a *curried*/*partially-applicable* version of `f`!

C++17 `curry`

As mentioned in the beginning of the article, these are the goals for our `curry` function:

- Given a generic function object `f`, invoking `curry(f)` returns a *curried*/*partially-applicable* version of `f`.
- If `f` is `constexpr`-friendly, the returned one will be as well.
- `curry` should not introduce any overhead compared to hand-written *currying*/*partial application*.

Credit where it's due

Please note that the design and implementation of `curry` that I am going to cover is a *heavily-modified version* of this snippet that was tweeted by **Julian Becker** (https://twitter.com/awtem/status/804781466852950017) - in fact, it was that tweet that inspired me to write this article. **Thanks!**

(Julian also wrote an excellent answer (http://stackoverflow.com/questions/152005/how-can-currying-be-done-in-c/26768388#26768388) on the StackOverflow question "How can currying be done in C++?" - make sure to check it out.)

Example usage

Before we analyze the *declaration* and *definition* of `curry`, let's take a look at some usage examples.

- **Nullary functions:**

  ```
  auto greet = []{ std::puts("hi!\n"); };

  greet(); // Prints "hi!".
  curry(greet); // Prints "hi!".

  // Compile-time error:
  /* curry(greet)(); */
  ```

  As you can see, in the case of a *nullary function object* `f`, invoking `curry(f)` calls the original object immediately.

- **Unary functions**:

```cpp
auto plus_one = [](auto x){ return x + 1; };

plus_one(0); // Returns `1`.
curry(plus_one)(0); // Returns `1`.

// Returns a wrapper around `plus_one` that enables
// currying/partial application.
// `plus_one` is "perfectly-captured" in the wrapper.
auto curried_plus_one = curry(plus_one);

curried_plus_one(1); // Returns `2`.
```

What does *perfectly-captured* mean?

It means that if the captured object is an *lvalue*, it will be captured *by reference*. If the captured object is an *rvalue, it will be captured* by move*. I've written a comprehensive article on this topic: **"capturing perfectly-forwarded objects in lambdas"** (http://vittorioromeo.info/index/blog/capturing_perfectly_forwarded_objects_in_lambdas.html).

- **Binary functions**:

```cpp
auto add2 = [](auto a, auto b){ return a + b; };

// All of the invocations below return `3`.
add2(1, 2);
curry(add2)(1, 2); // Partial application.
curry(add2)(1)(2); // Currying.

// Example of "binding" an argument:
auto add_one = curry(add2)(1);
add_one(2); // Returns `3`.
add_one(3); // Returns `4`.
```

You should be starting to see the pattern now...

- **N-ary functions**:

```cpp
auto add3 = [](auto a, auto b, auto c)
{
    return a + b + c;
};

// All of the invocations below return `6`.
add3(1, 2, 3);
curry(add3)(1, 2, 3);
curry(add3)(1, 2)(3);
curry(add3)(1)(2, 3);
curry(add3)(1)(2)(3);
```

The example above shows that *currying* and *partial application* can be freely combined. Let's see another example of that with a `constexpr` lambda (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4487.pdf) of arity 5.

```cpp
auto add5 = [](auto a, auto b, auto c,
               auto d, auto e) constexpr
{
    return a + b + c + d + e;
};

constexpr auto csum5 = curry(sum5);

constexpr auto a = csum5(0, 1, 2, 3, 4, 5);
constexpr auto b = csum5(0)(1, 2, 3, 4, 5);
constexpr auto c = csum5(0, 1)(2, 3, 4, 5);
constexpr auto d = csum5(0, 1, 2)(3, 4, 5);
constexpr auto e = csum5(0, 1, 2, 3)(4, 5);
constexpr auto f = csum5(0, 1, 2, 3, 4)(5);
```

Note that the usages of `curry(sum5)` above are in no way exhaustive - more combinations such as `curry(sum5)(0, 1)(2, 3)(4, 5)` can be written, and every *intermediate step* can be given a name.

Now that you have an idea on how `curry` can be used, let's dive into its *declaration* and *definition*.

Declaration

Given the constraints listed earlier, we can easily write down the *declaration* of `curry`.

```
template <typename TF>
constexpr decltype(auto) curry(TF&& f);
```

Why `decltype(auto)` instead of `auto`?

Because the *final step* of `curry` needs to return exactly what the original function object does. Example:

```
auto f = [](auto, auto) -> auto&
{
    return some_global_variable;
};

// OK - can return an additional "curry wrapper" by value.
auto step0 = curry(f);

// Same as above.
auto step1 = step0('a');

// Now `step1` has to return a reference!
auto& that_same_global = step1('b');
```

Additionally, the `f` parameter is taken by *forwarding-reference* (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4164.pdf). I will assume you're familiar with *move semantics* (http://stackoverflow.com/questions/3106110/what-are-move-semantics), `std::forward` (http://en.cppreference.com/w/cpp/utility/forward), and **"forward captures"** (http://vittorioromeo.info/index/blog/capturing_perfectly_forwarded_objects_in_lambdas.html) for the rest of the article.

Definition

I'll show the complete definition of `curry` first, and then analyze all the parts one-by-one more closely.

```
template <typename TF>
constexpr decltype(auto) curry(TF&& f)
{
    if constexpr (std::is_callable<TF&&()>{})
    {
        return FWD(f)();
    }
    else
    {
        return [xf = FWD_CAPTURE(f)](auto&&... partials) mutable constexpr
        {
            return curry
            (
                [
                    partial_pack = FWD_CAPTURE_PACK_AS_TUPLE(partials),
                    yf = std::move(xf)
                ]
                (auto&&... xs) constexpr
                    -> decltype(forward_like<TF>(xf.get())(FWD(partials)...,
                                                          FWD(xs)...))
                {
                    return apply_fwd_capture([&yf](auto&&... ys) constexpr
                        -> decltype(forward_like<TF>(yf.get())(FWD(ys)...))
                    {
                        return forward_like<TF>(yf.get())(FWD(ys)...);
                    }, partial_pack, FWD_CAPTURE_PACK_AS_TUPLE(xs));
                }
            );
        };
    }
}
```

The first thing to notice is the **recursive** structure of `curry`.

```
template <typename TF>
constexpr decltype(auto) curry(TF&& f)
{
    if constexpr (std::is_callable<TF&&()>{})
    {
        return FWD(f)();
    }
    else
    {
        // ...
    }
}
```

The *base case* branch is taken when `std::is_callable<TF&&()>{}` evaluates to `true`. [`std::is_callable` (http://en.cppreference.com/w/cpp/types/is_f)](http://en.cppreference.com/w/cpp/types/is_f) is a new C++17 *type trait* that checks whether or not a particular object types can be called with a specific set of argument types.

- If `std::is_callable<TF&&()>{}` evaluates to `false`, then it means that `TF` needs some arguments in order to be called - those arguments can be *curried*/*partially-applied*.

- If it evaluates to `true`, it means that there are no more arguments to *curry*/*partially-apply* in `f`. Therefore, `f` can be invoked to get the final result:

  ```
  return FWD(f)();
  ```

  `FWD` is a macro that expands to `std::forward<decltype(f)>(f)`. It's being used as `TF` may have a *ref-qualified* (https://akrzemi1.wordpress.com/2014/06/02/ref-qualifiers/) `operator()` that behaves differently depending on `f`'s value category.

We will now focus on the *recursive case* of `curry`. The first step is allowing *partial application* of arguments - since we don't know how many arguments will be bound in advance, a *generic variadic lambda* will be returned:

```
return [xf = FWD_CAPTURE(f)](auto&&... partials) mutable constexpr
{
    return curry(/* ... */);
}
```

The returned lambda will:

- Capture `f` by *forward capture* into `xf`.
- Accept any amount of *forwarding references* in the `partials...` pack. These arguments will be *bound* for subsequent calls.
- Be marked as `mutable`: this is **important** as `xf` will be moved in the inner lambda's capture list.
- Be marked as `constexpr`: this allows `curry` to be used as a *constant expression* (http://en.cppreference.com/w/cpp/language/constant_expression) where possible.
  - Note that, since C++17, **lambdas are implicity `constexpr`** (http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4487.pdf) unless they fail to satisfy any `constexpr` function requirement (http://en.cppreference.com/w/cpp/language/constexpr).
- Recursively call `curry` in its body, returning a new *curried*/*partially-applicable* function.

Let's now focus on the `return curry(/*...*/)` statement. We want to return a *curried* version of a new intermediate function object where:

- The `partials...` pack values are *bound* for its invocation - these values will be captured by *forward capture* as `partial_pack`.
- The *forward-captured* `xf` from the "parent" lambda is captured *by move* into `yf`. `xf` doesn't need to be forwarded as `FWD_CAPTURE(f)` returns a movable wrapper that either stores an *lvalue reference* or a *value*.

```
return curry
(
    [
        partial_pack = FWD_CAPTURE_PACK_AS_TUPLE(partials),
        yf = std::move(xf)
    ]
    (auto&&... xs) constexpr
        -> decltype(forward_like<TF>(xf.get())(FWD(partials)...,
                                               FWD(xs)...))
    {
        // ...
    }
);
```

The lambda passed to `curry` will accept any number of *forwarding references* in the `xs...` pack that will be used alongside the captured `partials...` to call `f`. The expected function call can be easily understood by the lambda's *trailing return type*:

```
//          Unwrap `f` from the `xf` `FWD_CAPTURE` wrapper and propagate
//          the original function object's value category.
//          vvvvvvvvvvvvvvvvvvvvvvvvvv
-> decltype(forward_like<TF>(xf.get())(FWD(partials)..., FWD(xs)...))
//                                    ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
//          Unpack both the `partials` and `xs` argument packs in a
//          single function call to `f`.
```

`forward_like` is an utility function in my `vrm_core` library (https://github.com/SuperV1234/vrm_core/blob/437a0afb35385250cd75c22babaeeecbfa4dcacc/include/vrm/core/type_traits/forward_like.hp that *forwards* the passed argument with the same *value category* of the potentially-unrelated specified type. It basically copies the "*lvalue/rvalue*-ness" of the user-provided template parameter and *applies* it to its argument.

The expression inside the above return type essentially means: *"invoke the original function object by unpacking `partials...` and `xs...` one after another"*.

Lastly, let's analyze the body of the lambda.

```
return apply_fwd_capture([&yf](auto&&... ys) constexpr
    -> decltype(forward_like<TF>(yf.get())(FWD(ys)...))
{
    return forward_like<TF>(yf.get())(FWD(ys)...);
}, partial_pack, FWD_CAPTURE_PACK_AS_TUPLE(xs));
```

Remember: we're trying to call `f` by unpacking both `partials...` and `xs...` **at the same time**. The `partials...` pack is stored in a special wrapper returned by `FWD_CAPTURE_PACK_AS_TUPLE`. The `xs...` pack contains the arguments passed to the lambda.

The `apply_fwd_capture` takes any number of wrapped *forward-capture pack wrappers* and uses them to invoke an user-provided function object. The wrappers are unpacked at the same time, preserving the original value category. Since `xs...` is not wrapped, we're going to explicitly do so by using the `FWD_CAPTURE_PACK_AS_TUPLE` macro.

In short, `apply_fwd_capture` will invoke the *`constexpr` variadic lambda* by expanding `partials...` and `xs...` correctly - those values will be then forwarded to the wrapped callable object `yf`.

That's it! Eventually the recursion will end as one of the steps will produce an intermediate function objects that satisfies `std::is_callable<TF&&()>{}`, giving back a "concrete" result to the caller.

Generated assembly benchmarks

As I did in my previous **"passing functions to functions"** (https://vittorioromeo.info/index/blog/passing_functions_to_functions.html) article, I will compare the number of generated assembly lines for different code snippets where `curry` is used. The point of these "benchmarks" is giving the readers an idea on how easy it is for the compiler to optimize `curry` out - they are in no way exhaustive or representative of a real-world situation. *(The benchmarks were generated with this Python script (https://github.com/SuperV1234/vittorioromeo.info/blob/master/extra/cpp17_curry/bench/dobenchs.py), which also prints out the assembly.)*

The compiler used for these measurements is **g++ 7.0.0 20170113**, compiled from the SVN repository.

`constexpr` variables

When `curry` is used in a `constexpr` context it is trivial to prove that it gets completely optimized out by the compiler. Regardless, here's the snippet that's going to be measured:

```
int main()
{
    const auto sum = [](auto a, auto b, auto c, auto d, auto e, auto f, auto g,
        auto h) constexpr
    {
        return a + b + c + d + e + f + g + h;
    };

    constexpr auto expected = sum(0, 1, 2, 3, 4, 5, 6, 7);

#if defined(VR_BASELINE)
    constexpr auto s0 = sum(0, 1, 2, 3, 4, 5, 6, 7);
    constexpr auto s1 = sum(0, 1, 2, 3, 4, 5, 6, 7);
    constexpr auto s2 = sum(0, 1, 2, 3, 4, 5, 6, 7);
    constexpr auto s3 = sum(0, 1, 2, 3, 4, 5, 6, 7);
    constexpr auto s4 = sum(0, 1, 2, 3, 4, 5, 6, 7);
    constexpr auto s5 = sum(0, 1, 2, 3, 4, 5, 6, 7);
    constexpr auto s6 = sum(0, 1, 2, 3, 4, 5, 6, 7);
    constexpr auto s7 = sum(0, 1, 2, 3, 4, 5, 6, 7);
#elif defined(VR_CURRY)
    constexpr auto s0 = curry(sum)(0, 1, 2, 3, 4, 5, 6, 7);
    constexpr auto s1 = curry(sum)(0)(1, 2, 3, 4, 5, 6, 7);
    constexpr auto s2 = curry(sum)(0, 1)(2, 3, 4, 5, 6, 7);
    constexpr auto s3 = curry(sum)(0, 1, 2)(3, 4, 5, 6, 7);
    constexpr auto s4 = curry(sum)(0, 1, 2, 3)(4, 5, 6, 7);
    constexpr auto s5 = curry(sum)(0, 1, 2, 3, 4)(5, 6, 7);
    constexpr auto s6 = curry(sum)(0, 1, 2, 3, 4, 5)(6, 7);
    constexpr auto s7 = curry(sum)(0, 1, 2, 3, 4, 5, 6)(7);
#endif

    static_assert(s0 == expected);
    static_assert(s1 == expected);
    static_assert(s2 == expected);
    static_assert(s3 == expected);
    static_assert(s4 == expected);
    static_assert(s5 == expected);
    static_assert(s6 == expected);
    static_assert(s7 == expected);

    return s0 + s1 + s2 + s3 + s4 + s5 + s6 + s7;
}
```

- `sum` is a `constexpr` generic lambda with an arity of 8.
- When measuring the baseline, the `s0`...`s7` `constexpr` variables are initialized by simply calling `sum`.
- When using `curry`, `s0`...`s7` are initialized by using various invocations of `curry(sum)`.
- In the end, the expected sum result is statically asserted and returned from `main`.

Baseline

O0	O1	O2	O3	Ofast
14	2	2	2	2

Curry

O0	O1	O2	O3	Ofast
14 *(+0.0%)*	2 *(+0.0%)*	2 *(+0.0%)*	2 *(+0.0%)*	2 *(+0.0%)*

As shown by the tables above, using `curry` introduces no additional overhead when used in the initialization of `constexpr` variables.

[You can find the complete snippet on GitHub. (https://github.com/SuperV1234/vittorioromeo.info/blob/master/extra/cpp17_curry/bench/b0_constexpr.cpp)](https://github.com/SuperV1234/vittorioromeo.info/blob/master/extra/cpp17_curry/bench/b0_constexpr.cpp)

`volatile` variables

Let's now measure the eventual overhead of `curry` when initializing `volatile` variables. The snippet is almost identical to the previous one, except for a few differences:

- The `s0`...`s7` variables are now marked as `volatile` instead of `constexpr`.
- The `static_assert(x)` checks have been replaced with `if(!x){ return -1; }`.

Baseline

O0	O1	O2	O3	Ofast
68	56	42	42	42

Curry

O0	O1	O2	O3	Ofast
68 *(+0.0%)*	56 *(+0.0%)*	42 *(+0.0%)*	42 *(+0.0%)*	42 *(+0.0%)*

Even with `volatile`, there isn't any additional overhead introduced by `curry`!

[You can find the complete snippet on GitHub. (https://github.com/SuperV1234/vittorioromeo.info/blob/master/extra/cpp17_curry/bench/b1_volatile.cpp)](https://github.com/SuperV1234/vittorioromeo.info/blob/master/extra/cpp17_curry/bench/b1_volatile.cpp)

Intermediate `curry` steps

The above benchmarks never stored any intermediate `curry` return value - the entire expression was part of the `s0`...`s7` initializer expression. Let's see what happens when those intermediate steps are stored as follows:

```
auto i0 = curry(sum);
auto i1 = curry(sum)(0);
auto i2 = curry(sum)(0, 1);
auto i3 = curry(sum)(0, 1, 2);
auto i4 = curry(sum)(0, 1, 2, 3);
auto i5 = curry(sum)(0, 1, 2, 3, 4);
auto i6 = curry(sum)(0, 1, 2, 3, 4, 5);
auto i7 = curry(sum)(0, 1, 2, 3, 4, 5, 6);

volatile auto s0 = i0(0, 1, 2, 3, 4, 5, 6, 7);
volatile auto s1 = i1(1, 2, 3, 4, 5, 6, 7);
volatile auto s2 = i2(2, 3, 4, 5, 6, 7);
volatile auto s3 = i3(3, 4, 5, 6, 7);
volatile auto s4 = i4(4, 5, 6, 7);
volatile auto s5 = i5(5, 6, 7);
volatile auto s6 = i6(6, 7);
volatile auto s7 = i7(7);
```

Baseline

O0	O1	O2	O3	Ofast
68	56	42	42	42

Curry

O0	O1	O2	O3	Ofast
19141 *(+2804%)*	56 *(+0.0%)*	42 *(+0.0%)*	42 *(+0.0%)*	42 *(+0.0%)*

From optimization level `-O1` onwards everything is great: **zero overhead**! When using `-O0`, though, there is a quite noticeable overhead of $+2804\%$ extra generated assembly compared to the baseline.

[You can find the complete snippet on GitHub. (https://github.com/SuperV1234/vittorioromeo.info/blob/master/extra/cpp17_curry/bench/b2_intermediate.cpp)](https://github.com/SuperV1234/vittorioromeo.info/blob/master/extra/cpp17_curry/bench/b2_intermediate.cpp)

(Some additional benchmarks with `volatile` lambda parameters and values are available on the GitHub repository (https://github.com/SuperV1234/vittorioromeo.info/tree/master/extra/cpp17_curry/bench).)

Compiler bugs

> `curry` looks great! Zero run-time overhead, *partial application* and *currying* all in one... what's the catch?

Well, the hardest part is... getting `curry` to compile. As seen from these tweets (https://twitter.com/supahvee1234/status/811246691731042304) between me and Julian Becker, it seems that both `g++` and `clang++` fail with *internal compiler errors* for different reasons.

- This snippet on *gcc.godbolt.org* (https://godbolt.org/g/9rP7ZO) produces a g++ *internal compiler error*. Commenting out the *trailing return type* on line 158 fixes the ICE. I reported a minimal version of this issue as bug #78006 (https://gcc.gnu.org/bugzilla/show_bug.cgi?id=78006).
- clang++'s frontend crashes in versions 3.9 and 4.0 with this *wandbox* snippet (http://melpon.org/wandbox/permlink/ahl5bK74C86ddZga). I've reported this as bug #31435 (https://llvm.org/bugs/show_bug.cgi?id=31435).

I managed to compile `curry` and the snippets used for this article by cloning the latest version of gcc from SVN and compiling it on my machine - I assume that some of the crashes were fixed in on *trunk* and *gcc.godbolt.org* is still a little bit behind.

Acknowledgments

Many thanks to *Julian Becker* (https://twitter.com/awtem) and *Jackie Kay* (https://twitter.com/jackayline) for proofreading the article and providing very valuable feedback.

Barney Dellar • 2 months ago

In "Unary Functions", you have:

curried_plus_one(1); // Returns `1`.

Is that right?? Surely it returns 2..

• Reply • Share ›

Vittorio Romeo Mod → Barney Dellar • 2 months ago

Whoops. Will fix in a few hours :)

• Reply • Share ›

about me

Hello! My name is Vittorio.
I'm a modern C++ enthusiast who loves to share his knowledge by creating video tutorials and participating to conferences.
I have a BS in Computer Science from the *University of Messina*. I write libraries, applications and games.
Check out my GitHub page and feel free to contact me if you're interested in my projects.
Please consider donating if you enjoy my work.

about this site

This is my personal website. It's statically generated by a C++14 program, using a JSON library and a templating system both written from scratch. I will use this website both as a blog and as a hub for all of my projects.
You can find the source code on my GitHub page, which can be reached through the links below.
Enjoy your stay!

contact me

(mailto:vittorio.romeo@outlook.com) (http://www.facebook.com/vittoriromeovee)

(https://github.com/SuperV1234) (http://www.youtube.com/user/SuperVictorius)

(https://twitter.com/supahvee1234) (http://stackexchange.com/users/294676/vittorio-romeo)