Fit Documentation

Release 0.6

Paul Fultz II

Feb 16, 2018

Contents

Paul Fultz II

CHAPTER 1

Introduction

1.1 About

Fit is a header-only C++11/C++14 library that provides utilities for functions and function objects, which can solve many problems with much simpler constructs than whats traditionally been done with metaprogramming.

Fit is:

- Modern: Fit takes advantages of modern C++11/C++14 features. It support both `constexpr` initialization and `constexpr` evaluation of functions. It takes advantage of type deduction, varidiac templates, and perfect forwarding to provide a simple and modern interface.
- Relevant: Fit provides utilities for functions and does not try to implement a functional language in C++. As such, Fit solves many problems relevant to C++ programmers, including initialization of function objects and lambdas, overloading with ordering, improved return type deduction, and much more.
- Lightweight: Fit builds simple lightweight abstraction on top of function objects. It does not require subscribing to an entire framework. Just use the parts you need.

Fit is divided into three components:

- Function Adaptors and Decorators: These enhance functions with additional capability.
- Functions: These return functions that achieve a specific purpose.
- Utilities: These are general utilities that are useful when defining or using functions

Github: http://github.com/pfultz2/Fit

Documentation: http://fit.readthedocs.org

1.2 Motivation

- Improve the expressiveness and capabilities of functions, including first-class citzens for function overload set, extension methods, infix operators and much more.

- Simplify constructs in C++ that have generally required metaprogramming
- Enable point-free style programming
- Workaround the limitations of lambdas in C++14

1.3 Requirements

This requires a C++11 compiler. There are no third-party dependencies. This has been tested on clang 3.5-3.8, gcc 4.6-7, and Visual Studio 2015 and 2017.

1.3.1 Contexpr support

Both MSVC and gcc 4.6 have limited constexpr support due to many bugs in the implementation of constexpr. However, constexpr initialization of functions is supported when using the *FIT_STATIC_FUNCTION* and *FIT_STATIC_LAMBDA_FUNCTION* constructs.

1.3.2 Noexcept support

On older compilers such as gcc 4.6 and gcc 4.7, `noexcept` is not used due to many bugs in the implementation. Also, most compilers don't support deducing `noexcept` with member function pointers. Only newer versions of gcc(4.9 and later) support this.

1.4 Building

The Fit library uses cmake to build. To configure with cmake create a build directory, and run cmake:

```
mkdir build
cd build
cmake ..
```

1.4.1 Installing

To install the library just run the `install` target:

```
cmake --build . --target install
```

1.4.2 Tests

The tests can be built and run by using the `check` target:

```
cmake --build . --target check
```

The tests can also be ran using Boost.Build, just copy library to the boost source tree, and then:

```
cd test
b2
```

1.4.3 Documentation

The documentation is built using Sphinx. First, install the requirements needed for the documentation using `pip`:

```
pip install -r doc/requirements.txt
```

Then html documentation can be generated using `sphinx-build`:

```
sphinx-build -b html doc/ doc/html/
```

The final docs will be in the `doc/html` folder.

1.5 Getting started

1.5.1 Higher-order functions

A core part of this library is higher-order functions. A higher-order function is a function that either takes a function as its argument or returns a function. To be able to define higher-order functions, we must be able to refer functions as first-class objects. One example of a higher-order function is `std::accumulate`. It takes a custom binary operator as a parameter.

One way to refer to a function is to use a function pointer(or a member function pointer). So if we had our own custom `sum` function, we could pass it directly to `std::accumulate`:

```
int sum(int x, int y)
{
    return x + y;
}
// Pass sum to accumulate
std::vector<int> v = { 1, 2, 3 };
int total = std::accumulate(v.begin(), v.end(), 0, &sum);
```

However, a function pointer can only refer to one function in an overload set of functions, and it requires explicit casting to select that overload.

For example, if we had a templated `sum` function that we want to pass to `std::accumulate`, we would need an explicit cast:

```
template<class T, class U>
auto sum(T x, U y)
{
    return x + y;
}

auto sum_int = (int (*)(int, int))&sum;
// Call integer overload
int i = sum_int(1, 2);
// Or pass to an algorithm
std::vector<int> v = { 1, 2, 3 };
int total = std::accumulate(v.begin(), v.end(), 0, sum_int);
```

1.5.2 Function Objects

A function object allows the ability to encapsulate an entire overload set into one object. This can be done by defining a class that overrides the call operator like this:

```
// A sum function object
struct sum_f
{
    template<class T, class U>
    auto operator()(T x, U y) const
    {
        return x + y;
    }
};
```

There are few things to note about this. First, the call operator member function is always declared `const`, which is generally required to be used with Fit.(Note: The *mutable* adaptor can be used to make a mutable function object have a `const` call operator, but this should generally be avoided). Secondly, the `sum_f` class must be constructed first before it can be called:

```
auto sum = sum_f();
// Call sum function
auto three = sum(1, 2);
// Or pass to an algorithm
std::vector<int> v = { 1, 2, 3 };
int total = std::accumulate(v.begin(), v.end(), 0, sum);
```

Because the function is templated, it can be called on any type that has the plus + operator, not just integers. Futhermore, the `sum` variable can be used to refer to the entire overload set.

1.5.3 Lifting functions

Another alternative to defining a function object, is to lift the templated function using *FIT_LIFT*. This will turn the entire overload set into one object like a function object:

```
template<class T, class U>
auto sum(T x, U y)
{
    return x + y;
}

// Pass sum to an algorithm
std::vector<int> v = { 1, 2, 3 };
int total = std::accumulate(v.begin(), v.end(), 0, FIT_LIFT(sum));
```

However, due to limitations in C++14 this will not preserve `constexpr`. In those cases, its better to use a function object.

1.5.4 Declaring functions

Now, this is useful for local functions. However, many times we want to write functions and make them available for others to use. The Fit library provides *FIT_STATIC_FUNCTION* to declare the function object at the global or namespace scope:

```
FIT_STATIC_FUNCTION(sum) = sum_f();
```

The *FIT_STATIC_FUNCTION* declares a global variable following the best practices as outlined in N4381. This includes using `const` to avoid global state, compile-time initialization of the function object to avoid the static initialization order fiasco, and an external address of the function object that is the same across translation units to avoid possible One-Definition-Rule(ODR) violations. In C++17, this can be achieved using an `inline` variable:

```
inline const constexpr auto sum = sum_f{};
```

The *FIT_STATIC_FUNCTION* macro provides a portable way to do this that supports pre-C++17 compilers and MSVC.

1.5.5 Adaptors

Now we have defined the function as a function object, we can add new "enhancements" to the function. One enhancement is to write "extension" methods. The proposal N4165 for Unified Call Syntax(UFCS) would have allowed a function call of `x.f(y)` to become `f(x, y)`. Without UFCS in C++, we can instead use pipable function which would transform `x | f(y)` into `f(x, y)`. To make `sum_f` function pipable using the *pipable* adaptor, we can simply write:

```
FIT_STATIC_FUNCTION(sum) = pipable(sum_f());
```

Then the parameters can be piped into it, like this:

```
auto three = 1 | sum(2);
```

Pipable function can be chained mutliple times just like the `.` operator:

```
auto four = 1 | sum(2) | sum(1);
```

Alternatively, instead of using the `|` operator, pipable functions can be chained together using the *flow* adaptor:

```
auto four = flow(sum(2), sum(1))(1);
```

Another enhancement that can be done to functions is defining named infix operators using the *infix* adaptor:

```
FIT_STATIC_FUNCTION(sum) = infix(sum_f());
```

And it could be called like this:

```
auto three = 1 <sum> 2;
```

In addition, adaptors are provided that support simple functional operations such as partial application and function composition:

```
auto add_1 = partial(sum)(1);
auto add_2 = compose(add_1, add_1);
auto three = add_2(1);
```

1.5.6 Lambdas

Writing function objects can be a little verbose. C++ provides lambdas which have a much terser syntax for defining functions. Of course, lambdas can work with all the adaptors in the library, however, if we want to declare a function using lambdas, *FIT_STATIC_FUNCTION* won't work. Instead, *FIT_STATIC_LAMBDA_FUNCTION* can be used to

the declare the lambda as a function instead, this will initialize the function at compile-time and avoid possible ODR violations:

```
FIT_STATIC_LAMBDA_FUNCTION(sum) = [](auto x, auto y)
{
    return x + y;
};
```

Additionally, adaptors can be used, so the pipable version of `sum` can be written like this:

```
// Pipable sum
FIT_STATIC_LAMBDA_FUNCTION(sum) = pipable([](auto x, auto y)
{
    return x + y;
});
```

1.6 Examples

1.6.1 Print function

Say, for example, we would like to write a print function. We could start by writing the function that prints using `std::cout`, like this:

```
FIT_STATIC_LAMBDA_FUNCTION(print) = [](const auto& x)
{
    std::cout << x << std::endl;
};
```

However, there is lot of things that don't print directly to `std::cout` such as `std::vector` or `std::tuple`. Instead, we want to iterate over these data structures and print each element in them.

Overloading

The Fit library provides several ways to do overloading. One of the ways is with the *conditional* adaptor which will pick the first function that is callable. This allows ordering the functions based on which one is more important. So then the first function will print to `std::cout` if possible otherwise we will add an overload to print a range:

```
FIT_STATIC_LAMBDA_FUNCTION(print) = conditional(
    [](const auto& x) -> decltype(std::cout << x, void())
    {
        std::cout << x << std::endl;
    },
    [](const auto& range)
    {
        for(const auto& x:range) std::cout << x << std::endl;
    }
);
```

The `-> decltype(std::cout << x, void())` is added to the function to constrain it on whether `std::cout << x` is a valid expression. Then the `void()` is used to return `void` from the function. So, now the function can be called with a vector:

```
std::vector<int> v = { 1, 2, 3, 4 };
print(v);
```

This will print each element in the vector.

We can also constrain the second overload as well, which will be important to add more overloads. So a `for` range loop calls `begin` and `end` to iterated over the range, but we will need some helper function in order to call `std::begin` using ADL lookup:

```
namespace adl {

using std::begin;

template<class R>
auto adl_begin(R&& r) FIT_RETURNS(begin(r));
}
```

Now we can add `-> decltype(std::cout << *adl::adl_begin(range), void())` to the second function to constrain it to ranges:

```
FIT_STATIC_LAMBDA_FUNCTION(print) = conditional(
    [](const auto& x) -> decltype(std::cout << x, void())
    {
        std::cout << x << std::endl;
    },
    [](const auto& range) -> decltype(std::cout << *adl::adl_begin(range), void())
    {
        for(const auto& x:range) std::cout << x << std::endl;
    }
);
```

So now calling this will work:

```
std::vector<int> v = { 1, 2, 3, 4 };
print(v);
```

And print out:

```
1
2
3
4
```

Tuples

We could extend this to printing tuples as well. We will need to combine a couple of functions to make a `for_each_tuple`, which lets us call a function for each element. First, the *by* adaptor will let us apply a function to each argument passed in, and the *unpack* adaptor will unpack the elements of a tuple and apply them to the function:

```
FIT_STATIC_LAMBDA_FUNCTION(for_each_tuple) = [](const auto& sequence, auto f)
{
    return unpack(by(f))(sequence);
};
```

So now if we call:

```
for_each_tuple(std::make_tuple(1, 2, 3), [](auto i)
{
```

```
    std::cout << i << std::endl;
});
```

This will print out:

```
1
2
3
```

We can integrate this into our `print` function by adding an additional overload:

```
FIT_STATIC_LAMBDA_FUNCTION(print) = conditional(
    [](const auto& x) -> decltype(std::cout << x, void())
    {
        std::cout << x << std::endl;
    },
    [](const auto& range) -> decltype(std::cout << *adl::adl_begin(range), void())
    {
        for(const auto& x:range) std::cout << x << std::endl;
    },
    [](const auto& tuple)
    {
        for_each_tuple(tuple, [](const auto& x)
        {
            std::cout << x << std::endl;
        });
    }
);
```

So now we can call `print` with a tuple:

```
print(std::make_tuple(1, 2, 3));
```

And it will print out:

```
1
2
3
```

Recursive

Even though this will print for ranges and tuples, if we were to nest a range into a tuple this would not work. What we need to do is make the function call itself recursively. Even though we are using lambdas, we can easily make this recursive using the *fix* adaptor. This implements a fix point combinator, which passes the function(i.e. itself) in as the first argument.

So now we add an additional arguments called `self` which is the `print` function itself. This extra argument is called by the *fix* adaptor, and so the user would still call this function with a single argument:

```
FIT_STATIC_LAMBDA_FUNCTION(print) = fix(conditional(
    [](auto, const auto& x) -> decltype(std::cout << x, void())
    {
        std::cout << x << std::endl;
    },
    [](auto self, const auto& range) -> decltype(self(*adl::adl_begin(range)), void())
    {
```

```
        for(const auto& x:range) self(x);
    },
    [](auto self, const auto& tuple)
    {
        return for_each_tuple(tuple, self);
    }
));
```

This will let us print nested structures:

```
std::vector<int> v = { 1, 2, 3, 4 };
auto t = std::make_tuple(1, 2, 3, 4);
auto m = std::make_tuple(3, v, t);
print(m);
```

Which outputs this:

```
3
1
2
3
4
1
2
3
4
```

Variadic

We can also make this `print` function variadic, so it prints every argument passed into it. We can use the *by* adaptor, which already calls the function on every argument passed in. First, we just rename our original `print` function to `simple_print`:

```
FIT_STATIC_LAMBDA_FUNCTION(simple_print) = fix(conditional(
    [](auto, const auto& x) -> decltype(std::cout << x, void())
    {
        std::cout << x << std::endl;
    },
    [](auto self, const auto& range) -> decltype(self(*adl::adl_begin(range)), void())
    {
        for(const auto& x:range) self(x);
    },
    [](auto self, const auto& tuple)
    {
        return for_each_tuple(tuple, self);
    }
));
```

And then apply the *by* adaptor to `simple_print`:

```
FIT_STATIC_LAMBDA_FUNCTION(print) = by(simple_print);
```

Now we can call `print` with several arguments:

```
print(5, "Hello world");
```

Which outputs:

```
5
Hello world
```

1.6.2 Conditional overloading

Conditional overloading takes a set of functions and calls the first one that is callable. This is one of the ways to resolve ambiguity with overloads, but more importantly it allows an alternative function to be used when the first is not callable.

Stringify

Take a look at this example of defining a `stringify` function from stackoverflow here.

The user would like to write `stringify` to call `to_string` where applicable and fallback on using `sstream` to convert to a string. Most of the top answers usually involve some amount of metaprogramming using either `void_t` or `is_detected`(see n4502):

```
template<class T>
using to_string_t = decltype(std::to_string(std::declval<T>()));

template<class T>
using has_to_string = std::experimental::is_detected<to_string_t, T>;

template<typename T>
typename std::enable_if<has_to_string<T>{}, std::string>::type
stringify(T t)
{
    return std::to_string(t);
}
template<typename T>
typename std::enable_if<!has_to_string<T>{}, std::string>::type
stringify(T t)
{
    return static_cast<std::ostringstream&>(std::ostringstream() << t).str();
}
```

However, with the Fit library it can simply be written like this:

```
FIT_STATIC_LAMBDA_FUNCTION(stringify) = conditional(
    [](auto x) FIT_RETURNS(std::to_string(x)),
    [](auto x) FIT_RETURNS(static_cast<std::ostringstream&>(std::ostringstream() <<␣
↪x).str())
);
```

So, using *FIT_RETURNS* not only deduces the return type for the function, but it also constrains the function on whether the expression is valid or not. So by writing `FIT_RETURNS(std::to_string(x))` then the first function will try to call `std::to_string` function if possible. If not, then the second function will be called.

The second function still uses *FIT_RETURNS*, so the function will still be constrained by whether the << stream operator can be used. Although it may seem unnecessary because there is not another function, however, this makes the function composable. So we could use this to define a `serialize` function that tries to call `stringify` first, otherwise it looks for the member `.serialize()`:

```
FIT_STATIC_LAMBDA_FUNCTION(serialize) = conditional(
    [](auto x) FIT_RETURNS(stringify(x)),
    [](auto x) FIT_RETURNS(x.serialize())
);
```

static_if

In addition, this can be used with the *if* decorator to create `static_if`-like constructs on pre-C++17 compilers. For example, Baptiste Wicht discusses how one could write `static_if` in C++ here.

He wants to be able to write this:

```
template<typename T>
void decrement_kindof(T& value){
    if constexpr(std::is_same<std::string, T>()){
        value.pop_back();
    } else {
        --value;
    }
}
```

However, that isn't possible before C++17. With the Fit library one can simply write this:

```
template<typename T>
void decrement_kindof(T& value)
{
    eval(conditional(
        if_(std::is_same<std::string, T>())([&](auto id){
            id(value).pop_back();
        }),
        [&](auto id){
            --id(value);
        }
    ));
}
```

The `id` parameter passed to the lambda is the *identity* function. As explained in the article, this is used to delay the lookup of types by making it a dependent type(i.e. the type depends on a template parameter), which is necessary to avoid compile errors. The *eval* function that is called will pass this `identity` function to the lambdas.

The advantage of using the Fit library instead of the solution in Baptiste Wicht's blog, is that *conditional* allows more than just two conditions. So if there was another trait to be checked, such as `is_stack`, it could be written like this:

```
template<typename T>
void decrement_kindof(T& value)
{
    eval(conditional(
        if_(is_stack<T>())([&](auto id){
            id(value).pop();
        }),
        if_(std::is_same<std::string, T>())([&](auto id){
            id(value).pop_back();
        }),
        [&](auto id){
            --id(value);
        }
```

```
    ));
}
```

Type traits

Furthermore, this technique can be used to write type traits as well. Jens Weller was looking for a way to define a general purpose detection for pointer operands(such as `*` and `->`). One way to accomplish this is using Fit like this:

```
// Check that T has member function for operator* and ope
template<class T>
auto has_pointer_member(const T&) -> decltype(
    &T::operator*,
    &T::operator->,
    std::true_type{}
);

FIT_STATIC_LAMBDA_FUNCTION(has_pointer_operators) = conditional(
    FIT_LIFT(has_pointer_member),
    [](auto* x) -> bool_constant<(!std::is_void<decltype(*x)>())> { return {}; },
    always(std::false_type{})
);

template<class T>
struct is_dereferenceable
: decltype(has_pointer_operators(std::declval<T>()))
{};
```

Which is much simpler than the other implementations that were written, which were about 3 times the amount of code(see here).

The `has_pointer_operators` function works by first trying to call `has_pointer_member` which returns `true_type` if the type has member functions `operator*` and `operator->`, otherwise the function is not callable. The next function is only callable on pointers, which returns true if it is not a `void*` pointer(because `void*` pointers are not dereferenceable). Finally, if none of those functions matched then the last function will always return `false_type`.

1.6.3 Polymorphic constructors

Writing polymorphic constructors(such as `make_tuple`) is a boilerplate that has to be written over and over again for template classes:

```
template <class T>
struct unwrap_refwrapper
{
    typedef T type;
};

template <class T>
struct unwrap_refwrapper<std::reference_wrapper<T>>
{
    typedef T& type;
};

template <class T>
struct unwrap_ref_decay
```

```
: unwrap_refwrapper<typename std::decay<T>::type>
{};

template <class... Types>
std::tuple<typename unwrap_ref_decay<Types>::type...> make_tuple(Types&&... args)
{
    return std::tuple<typename unwrap_ref_decay<Types>::type...>(std::forward<Types>
↪(args)...);
}
```

The `construct` function takes care of all this boilerplate, and the above can be simply written like this:

```
FIT_STATIC_FUNCTION(make_tuple) = construct<std::tuple>();
```

1.6.4 More examples

As the Fit library is a collection of generic utilities related to functions, there is many useful cases with the library, but a key point of many of these utilities is that they can solve these problems with much simpler constructs than whats traditionally been done with metaprogramming. Lets take look at some of the use cases for using the Fit library.

Initialization

The *FIT_STATIC_FUNCTION* will help initialize function objects at global scope, and will ensure that it is initialized at compile-time and (on platforms that support it) will have a unique address across translation units, thereby reducing executable bloat and potential ODR violations.

In addition, *FIT_STATIC_LAMBDA* allows initializing a lambda in the same manner. This allows for simple and compact function definitions when working with generic lambdas and function adaptors.

Of course, the library can still be used without requiring global function objects for those who prefer to avoid them will still find the library useful.

Projections

Instead of writing the projection multiple times in algorithms:

```
std::sort(std::begin(people), std::end(people),
          [](const Person& a, const Person& b) {
            return a.year_of_birth < b.year_of_birth;
          });
```

We can use the *by* adaptor to project `year_of_birth` on the comparison operator:

```
std::sort(std::begin(people), std::end(people),
        by(&Person::year_of_birth, _ < _));
```

Ordering evaluation of arguments

When we write `f(foo(), bar())`, the standard does not guarantee the order in which the `foo()` and `bar()` arguments are evaluated. So with `apply_eval` we can order them from left-to-right:

```
apply_eval(f, [&]{ return foo(); }, [&]{ return bar(); });
```

Extension methods

Chaining many functions together, like what is done for range based libraries, can make things hard to read:

```
auto r = transform(
    filter(
        numbers,
        [](int x) { return x > 2; }
    ),
    [](int x) { return x * x; }
);
```

It would be nice to write this:

```
auto r = numbers
    .filter([](int x) { return x > 2; })
    .transform([](int x) { return x * x; });
```

The proposal N4165 for Unified Call Syntax(UFCS) would have allowed a function call of `x.f(y)` to become `f(x, y)`. However, this was rejected by the comittee. So instead pipable functions can be used to achieve extension methods. So it can be written like this:

```
auto r = numbers
    | filter([](int x) { return x > 2; })
    | transform([](int x) { return x * x; });
```

Now, if some users feel a little worried about overloading the *bitwise or* operator, pipable functions can also be used with *flow* like this:

```
auto r = flow(
    filter([](int x) { return x > 2; }),
    transform([](int x) { return x * x; })
)(numbers);
```

No fancy or confusing operating overloading and everything is still quite readable.

1.7 Point-free style programming

Point-free style programing(or tacit programming) is a style where the arguments to the function are not explicity defined. Rather, the function is defined as the composition of other functions where function adaptors manipulate the function arguments. The advantage of using point-free style in C++ is the template machinery involved with function arguments can be avoided.

1.7.1 Variadic print

For example, if we want to write a variadic print function that prints each argument, like this:

```
print("Hello", "World");
```

We would write something like the following, which would recursively iterate over the arguments using varidiac templates:

```
// Base case
void print()
{}

template<class T, class... Ts>
void print(const T& x, const Ts&... xs)
{
    std::cout << x;
    print(xs...);
}
```

Instead with point-free style, we can write this using the *by* adaptor, which calls a function on each arguments. Of course, `std::cout` is not a function, but we can make it one by using `FIT_LIFT`:

```
FIT_STATIC_FUNCTION(simple_print) = FIT_LIFT(std::ref(std::cout) << _);
```

This uses the *placeholders* to create a function that prints to `std::cout`. Then we can pass `simple_print` to the *by* adaptor:

```
FIT_STATIC_FUNCTION(print) = by(simple_print);
```

As the *by* adaptor calls the function for each argument passed in, `b(f)(x, y)` is the equivalent of calling `f(x)` and then `f(y)`. In this case, it will call `simple_print(x)` and then `simple_print(y)`:

```
print("Hello", "World");
```

Which prints out:

```
HelloWorld
```

Of course, this puts all the output together, but we can further extend this to print a new line for each item by composing it:

```
FIT_STATIC_FUNCTION(print_lines) = by(flow(simple_print, _ << std::integral_constant
↪<char, '\n'>{}));
```

The *flow* adaptor does function composition but the functions are called from left-to-right. That is `flow(f, g)(x)` is equivalent to `g(f(x))`. So in this case, it will call `simple_print` on the argument which returns `std::cout` and then pass that to the next function which calls the stream with the newline character. In the above, we write `std::integral_constant<char, '\n'>{}` instead of just `'\n'` because the function is statically defined, so all values must be defined statically.

So now calling `print_lines`:

```
print_lines("Hello", "World");
```

It will print out:

```
Hello
World
```

With each argument on its own line.

1.7.2 Variadic sum

Another example, say we would like to write a varidiac version of `max`. We could implement it like this:

```
// Base case
template<class T>
T max(const T& x)
{
    return x;
}

template<class T, class... Ts>
T max(const T& x, const T& y, const Ts&... xs)
{
    return std::max(x, max(y, xs...));
}
```

With point-free style programming, we can recognize this is a fold, and write it using the *fold* adaptor, which will do a fold over the variadic parameters:

```
FIT_STATIC_FUNCTION(max) = fold(FIT_LIFT(std::max));
```

FIT_LIFT is used to grab the entire overload set of `std::max` function, which is needed since `std::max` is templated and we want the variadic `std::max` function to handle any types as well. So now it can be called like this:

```
auto n = max(1, 2, 4, 3); // Returns 4
auto m = max(0.1, 0.2, 0.5, 0.4); // Returns 0.5
```

By using *fold*, `max(1, 2, 4, 3)` will call `std::max` like `std::max(std::max(std::max(1, 2), 4), 3)` and `max(0.1, 0.2, 0.5, 0.4)` will be called like `std::max(std::max(std::max(0.1, 0.2), 0.5), 0.4)`.

CHAPTER 2

Overview

2.1 Definitions

2.1.1 Function Adaptor

A *Function Adaptor* takes a function(or functions) and returns a new function with enhanced capability. Each adaptor has a functional form with a corresponding class with `_adaptor` appended to it:

```
template<class... Fs>
FunctionAdaptor_adaptor<Fs...> FunctionAdaptor(Fs...);
```

Both the functional form and the class form can be used to construct the adaptor.

2.1.2 Static Function Adaptor

A static function adaptor is a *Function Adaptor* that doesn't have a functional form. It is only a class. It has an additional requirement that the function is `DefaultConstructible`:

```
template<class... Fs>
class StaticFunctionAdaptor;
```

2.1.3 Decorator

A decorator is a function that returns a *Function Adaptor*. The *Function Adaptor* may be an unspecified or private type.

```
template<class... Ts>
FunctionAdaptor Decorator(Ts...);
```

2.1.4 Semantics

Some parts of the documentation provides the meaning(or equivalence) of an expression. Here is a guide of those symbols:

- `f`, `g`, `fs`, `gs`, `p` are functions
- `x`, `y`, `xs`, `ys` are parameters to a function
- `T` represents some type
- `...` are parameter packs and represent varidiac parameters

2.1.5 Signatures

All the functions are global function objects except where an explicit template parameter is required on older compilers. However, the documentation still shows the traditional signature since it is much clearer. So instead of writing this:

```
struct if_f
{
    template<class IntegralConstant>
    constexpr auto operator()(IntegralConstant) const;
};
const constexpr if_f if_ = {};
```

The direct function signature is written even though it is actually declared like above:

```
template<class IntegralConstant>
constexpr auto if_(IntegralConstant);
```

Its usage is the same except it has the extra benefit that the function can be directly passed to another function.

2.2 Concepts

2.2.1 ConstFunctionObject

Is an object with a `const` call operator:

```
concept ConstFunctionObject
{
    template<class... Ts>
    auto operator()(Ts&&...) const;
};
```

Requirements:

The type `F` satisfies `ConstFunctionObject` if

- The type `F` satisfies `std::is_object`, and

Given

- `f`, an object of type `const F`
- `args...`, suitable argument list, which may be empty

Expression	Requirements
`f(args...)`	performs a function call

2.2.2 NullaryFunctionObject

Is an object with a `const` call operator that accepts no parameters:

```
concept NullaryFunctionObject
{
    auto operator()() const;
};
```

Requirements:

- `ConstFunctionObject`

Given

- `f`, an object of type `const F`

Expression	Requirements
`f()`	performs a function call

2.2.3 UnaryFunctionObject

Is an object with a `const` call operator that accepts one parameter:

```
concept UnaryFunctionObject
{
    template<class T>
    auto operator()(T&&) const;
};
```

Requirements:

- `ConstFunctionObject`

Given

- `f`, an object of type `const F`
- `arg`, a single argument

Expression	Requirements
`f(arg)`	performs a function call

2.2.4 BinaryFunctionObject

Is an object with a `const` call operator that accepts two parameter:

```
concept UnaryFunctionObject
{
    template<class T, class U>
    auto operator()(T&&, U&&) const;
};
```

Requirements:

- `ConstFunctionObject`

Given

- `f`, an object of type `const F`
- `arg1`, a single argument
- `arg2`, a single argument

Expression	Requirements
`f(arg1, arg2)`	performs a function call

2.2.5 MutableFunctionObject

Is an object with a `mutable` call operator:

```
concept MutableFunctionObject
{
    template<class... Ts>
    auto operator()(Ts&&...);
};
```

Requirements:

The type `F` satisfies `MutableFunctionObject` if

- The type `F` satisfies `std::is_object`, and

Given

- `f`, an object of type `F`
- `args...`, suitable argument list, which may be empty

Expression	Requirements
`f(args...)`	performs a function call

2.2.6 EvaluatableFunctionObject

Is an object that is either a `NullaryFunctionObject`, or it is an `UnaryFuntionObject` that accepts the `identity` function as a parameter.

Requirements:

- `NullaryFunctionObject`

Given

- `f`, an object of type `const F`

Expression	Requirements
`f()`	performs a function call

Or:

- `UnaryFuntionObject`

Given

- `f`, an object of type `const F`
- `identity`, which is the identity function

Expression	Requirements
`f(identity)`	performs a function call

2.2.7 Callable

Is an object for which the `INVOKE` operation can be applied.

Requirements:

The type `T` satisfies `Callable` if

Given

- `f`, an object of type `T`
- `Args...`, suitable list of argument types

The following expressions must be valid:

Expression	Requirements
`INVOKE(f, std::declval<Args>()...)`	the expression is well-formed in unevaluated context

where `INVOKE(f, x, xs...)` is defined as follows:

- if `f` is a pointer to member function of class `U`:
  - If `std::is_base_of<U, std::decay_t<decltype(x)>>()` is true, then `INVOKE(f, x, xs...)` is equivalent to `(x.*f)(xs...)`
  - otherwise, if `std::decay_t<decltype(x)>` is a specialization of `std::reference_wrapper`, then `INVOKE(f, x, xs...)` is equivalent to `(x.get().*f)(xs...)`
  - otherwise, if x does not satisfy the previous items, then `INVOKE(f, x, xs...)` is equivalent to `((*x).*f)(xs...)`.
- otherwise, if `f` is a pointer to data member of class `U`:
  - If `std::is_base_of<U, std::decay_t<decltype(x)>>()` is true, then `INVOKE(f, x)` is equivalent to `x.*f`
  - otherwise, if `std::decay_t<decltype(x)>` is a specialization of `std::reference_wrapper`, then `INVOKE(f, x)` is equivalent to `x.get().*f`

  - otherwise, if `x` does not satisfy the previous items, then `INVOKE(f, x)` is equivalent to `(*x).*f`
- otherwise, `INVOKE(f, x, xs...)` is equivalent to `f(x, xs...)`

2.2.8 ConstCallable

Is an object for which the `INVOKE` operation can be applied.

Requirements:

The type `T` satisfies `ConstCallable` if

Given

- `f`, an object of type `const T`
- `Args...`, suitable list of argument types

The following expressions must be valid:

Expression	Requirements
`INVOKE(f, std::declval<Args>()...)`	the expression is well-formed in unevaluated context

where `INVOKE(f, x, xs...)` is defined as follows:

- if `f` is a pointer to member function of class `U`:
  - If `std::is_base_of<U, std::decay_t<decltype(x)>>()` is true, then `INVOKE(f, x, xs...)` is equivalent to `(x.*f)(xs...)`
  - otherwise, if `std::decay_t<decltype(x)>` is a specialization of `std::reference_wrapper`, then `INVOKE(f, x, xs...)` is equivalent to `(x.get().*f)(xs...)`
  - otherwise, if x does not satisfy the previous items, then `INVOKE(f, x, xs...)` is equivalent to `((*x).*f)(xs...)`.
- otherwise, if `f` is a pointer to data member of class `U`:
  - If `std::is_base_of<U, std::decay_t<decltype(x)>>()` is true, then `INVOKE(f, x)` is equivalent to `x.*f`
  - otherwise, if `std::decay_t<decltype(x)>` is a specialization of `std::reference_wrapper`, then `INVOKE(f, x)` is equivalent to `x.get().*f`
  - otherwise, if `x` does not satisfy the previous items, then `INVOKE(f, x)` is equivalent to `(*x).*f`
- otherwise, `INVOKE(f, x, xs...)` is equivalent to `f(x, xs...)`

2.2.9 UnaryCallable

Is an object for which the `INVOKE` operation can be applied with one parameter.

Requirements:

- `ConstCallable`

Given

- `f`, an object of type `const F`

- `arg`, a single argument

Expression	Requirements
`INVOKE(f, arg)`	the expression is well-formed in unevaluated context

2.2.10 BinaryCallable

Is an object for which the `INVOKE` operation can be applied with two parameters.

Requirements:

- `ConstCallable`

Given

- `f`, an object of type `const F`
- `arg1`, a single argument
- `arg2`, a single argument

Expression	Requirements
`INVOKE(f, arg1, arg2)`	the expression is well-formed in unevaluated context

2.2.11 Metafunction

Given

- `f`, a type or a template
- `args...`, any suitable type, which may be empty

Expression	Requirements
`f::type`	The type is the result of the metafunction
`f<args...>::type`	The type is the result of the metafunction

2.2.12 MetafunctionClass

Given

- `f`, a type or a template
- `args...`, any suitable type, which may be empty

Expression	Requirements
`f::apply::type`	The type is the result of the metafunction
`f::apply<args...>::type`	The type is the result of the metafunction

CHAPTER 3

Reference

3.1 Function Adaptors

3.1.1 by

Header

```
#include <fit/by.hpp>
```

Description

The `by` function adaptor applies a projection onto the parameters of another function. This is useful, for example, to define a function for sorting such that the ordering is based off of the value of one of its member fields.

Also, if just a projection is given, then the projection will be called for each of its arguments.

Note: All projections are always evaluated in order from left-to-right.

Synopsis

```
template<class Projection, class F>
constexpr by_adaptor<Projection, F> by(Projection p, F f);

template<class Projection>
constexpr by_adaptor<Projection> by(Projection p);
```

Semantics

```
assert(by(p, f)(xs...) == f(p(xs)...));
assert(by(p)(xs...) == p(xs)...);
```

Requirements

Projection must be:

- *UnaryCallable*
- MoveConstructible

F must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

struct foo
{
    foo(int x_) : x(x_)
    {}
    int x;
};

int main() {
    assert(fit::by(&foo::x, _ + _)(foo(1), foo(2)) == 3);
}
```

References

- *Projections*
- *Variadic print*

3.1.2 compose

Header

```
#include <fit/compose.hpp>
```

Description

The `compose` function adaptor provides function composition. It produces a function object that composes a set of functions, ie the output of one function becomes the input of the second function. So, `compose(f, g)(0)` is equivalent to `f(g(0))`.

Synopsis

```
template<class... Fs>
constexpr compose_adaptor<Fs...> compose(Fs... fs);
```

Semantics

```
assert(compose(f, g)(xs...) == f(g(xs...)));
```

Requirements

Fs must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

struct increment
{
    template<class T>
    T operator()(T x) const
    {
        return x + 1;
    }
};

struct decrement
{
    template<class T>
    T operator()(T x) const
    {
        return x - 1;
    }
};

int main() {
    int r = compose(increment(), decrement(), increment())(3);
    assert(r == 4);
}
```

References

3.1.3 conditional

Header

```
#include <fit/conditional.hpp>
```

Description

The `conditional` function adaptor combines several functions together. If the first function can not be called, then it will try to call the next function. This can be very useful when overloading functions using template constraints(such as with `enable_if`).

Note: This is different than the `match` function adaptor, which can lead to ambiguities. Instead, `conditional` will call the first function that is callable, regardless if there is another function that could be called as well.

Synopsis

```
template<class... Fs>
constexpr conditional_adaptor<Fs...> conditional(Fs... fs);
```

Requirements

Fs must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <iostream>
using namespace fit;

struct for_ints
{
    void operator()(int) const
    {
        printf("Int\n");
    }
};

struct for_floats
{
    void operator()(float) const
    {
        printf("Float\n");
    }
};
```

```
int main() {
    conditional(for_ints(), for_floats())(3.0);
}
```

This will print `Int` because the `for_floats` function object won't ever be called. Due to the conversion rules in C++, the `for_ints` function can be called on floats, so it is chosen by `conditional` first, even though `for_floats` is a better match.

So, the order of the functions in the `conditional_adaptor` are very important to how the function is chosen.

References

- POO51 - Proposal for C++ Proposal for C++ generic overload function
- *Conditional overloading*

3.1.4 combine

Header

```
#include <fit/combine.hpp>
```

Description

The `combine` function adaptor combines several functions together with their arguments. It essentially zips each function with an argument before calling the main function.

Synopsis

```
template<class F, class... Gs>
constexpr combine_adaptor<F, Gs...> combine(F f, Gs... gs);
```

Semantics

```
assert(combine(f, gs...)(xs...) == f(gs(xs)...));
```

Requirements

F and Gs must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>
#include <tuple>
#include <utility>

int main() {
    auto f = fit::combine(
        fit::construct<std::tuple>(),
        fit::capture(1)(fit::construct<std::pair>()),
        fit::capture(2)(fit::construct<std::pair>()));
    assert(f(3, 7) == std::make_tuple(std::make_pair(1, 3), std::make_pair(2, 7)));
}
```

3.1.5 fold

Header

```
#include <fit/fold.hpp>
```

Description

The `fold` function adaptor uses a binary function to apply a fold operation to the arguments passed to the function. Additionally, an optional initial state can be provided, otherwise the first argument is used as the initial state.

The arguments to the binary function, take first the state and then the argument.

Synopsis

```
template<class F, class State>
constexpr fold_adaptor<F, State> fold(F f, State s);

template<class F>
constexpr fold_adaptor<F> fold(F f);
```

Semantics

```
assert(fold(f, z)() == z);
assert(fold(f, z)(x, xs...) == fold(f, f(z, x))(xs...));
assert(fold(f)(x) == x);
assert(fold(f)(x, y, xs...) == fold(f)(f(x, y), xs...));
```

Requirements

State must be:

- CopyConstructible

F must be:

- *BinaryCallable*

- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>

struct max_f
{
    template<class T, class U>
    constexpr T operator()(T x, U y) const
    {
        return x > y ? x : y;
    }
};
int main() {
    assert(fit::fold(max_f())(2, 3, 4, 5) == 5);
}
```

References

- Fold
- *Variadic sum*

3.1.6 decorate

Header

```
#include <fit/decorate.hpp>
```

Description

The `decorate` function adaptor helps create simple function decorators.

A function adaptor takes a function and returns a new functions whereas a decorator takes some parameters and returns a function adaptor. The `decorate` function adaptor will return a decorator that returns a function adaptor. Eventually, it will invoke the function with the user- provided parameter and function.

Synopsis

```
template<class F>
constexpr decorate_adaptor<F> decorate(F f);
```

Semantics

```
assert(decorate(f)(x)(g)(xs...) == f(x, g, xs...));
```

Requirements

F must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>
#include <iostream>
#include <string>
using namespace fit;

struct logger_f
{
    template<class F, class... Ts>
    auto operator()(const std::string& message, F&& f, Ts&&... xs) const
        -> decltype(f(std::forward<Ts>(xs)...))
    {
        // Message to print out when the function is called
        std::cout << message << std::endl;
        // Call the function
        return f(std::forward<Ts>(xs)...);
    }
};
// The logger decorator
FIT_STATIC_FUNCTION(logger) = fit::decorate(logger_f());

struct sum_f
{
    template<class T, class U>
    T operator()(T x, U y) const
    {
        return x+y;
    }
};

FIT_STATIC_FUNCTION(sum) = sum_f();
int main() {
    // Use the logger decorator to print "Calling sum" when the function is called
    assert(3 == logger("Calling sum")(sum)(1, 2));
}
```

3.1.7 fix

Header

```
#include <fit/fix.hpp>
```

Description

The `fix` function adaptor implements a fixed-point combinator. This can be used to write recursive functions.

When using `constexpr`, a function can recurse to a depth that is defined by `FIT_RECURSIVE_CONSTEXPR_DEPTH`(default is 16). There is no limitiation on recursion depth for non-constexpr functions. In addition, due to the eagerness of `constexpr` to instantiation templates, in some cases, an explicit return type must be specified in order to avoid reaching the recursion limits of the compiler. This can be accomplished using *result*:

```
int r = fit::result<int>(factorial)(5);
```

Synopsis

```
template<class F>
constexpr fix_adaptor<F> fix(F f);
```

Semantics

```
assert(fix(f)(xs...) == f(fix(f), xs...));
```

Requirements

F must be:

- *ConstFunctionObject*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>

int main() {
    auto factorial = fit::fix(
        [](auto recurse, auto x) -> decltype(x) {
            return x == 0 ? 1 : x * recurse(x-1);
        }
    );
    int r = fit::result<int>(factorial)(5);
    assert(r == 5*4*3*2*1);
}
```

References

- Fixed-point combinator
- *Recursive*

3.1.8 flip

Header

```
#include <fit/flip.hpp>
```

Description

The `flip` function adaptor swaps the first two parameters.

Synopsis

```
template<class F>
flip_adaptor<F> flip(F f);
```

Semantics

```
assert(flip(f)(x, y, xs...) == f(y, x, xs...));
```

Requirements

F must be at least:

- *BinaryCallable*

Or:

- *Callable* with more than two argurments

And:

- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>

int main() {
    int r = fit::flip(fit::_ - fit::_)(2, 5);
    assert(r == 3);
}
```

3.1.9 flow

Header

```
#include <fit/flow.hpp>
```

Description

The `flow` function adaptor provides function composition. It is useful as an alternative to using the pipe operator `|` when chaining functions. It is similiar to `compose` except the evaluation order is reversed. So, `flow(f, g)(0)` is equivalent to `g(f(0))`.

Synopsis

```
template<class... Fs>
constexpr flow_adaptor<Fs...> flow(Fs... fs);
```

Semantics

```
assert(flow(f, g)(xs...) == g(f(xs...)));
```

Requirements

Fs must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

struct increment
{
    template<class T>
    T operator()(T x) const
    {
        return x + 1;
    }
};

struct decrement
{
    template<class T>
    T operator()(T x) const
    {
        return x - 1;
    }
};

int main() {
    int r = flow(increment(), decrement(), increment())(3);
    assert(r == 4);
}
```

References

3.1.10 implicit

Header

```
#include <fit/implicit.hpp>
```

Description

The `implicit` adaptor is a static function adaptor that uses the type that the return value can be converted to, in order to determine the type of the template parameter. In essence, it will deduce the type for the template parameter using the type of variable the result is assigned to. Since it is a static function adaptor, the function must be default constructible.

Synopsis

```
template<template <class...> class F>
class implicit<F>;
```

Semantics

```
assert(T(implicit<F>()(xs...)) == F<T>()(xs...));
```

Requirements

F must be a template class, that is a:

- *ConstFunctionObject*
- DefaultConstructible

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

template<class T>
struct auto_caster
{
    template<class U>
    T operator()(U x)
    {
        return T(x);
    }
};

static constexpr implicit<auto_caster> auto_cast = {};
```

```
struct auto_caster_foo
{
    int i;
    explicit auto_caster_foo(int i_) : i(i_) {}

};

int main() {
    float f = 1.5;
    int i = auto_cast(f);
    auto_caster_foo x = auto_cast(1);
    assert(1 == i);
    assert(1 == x.i);
}
```

3.1.11 indirect

Header

```
#include <fit/indirect.hpp>
```

Description

The `indirect` function adaptor dereferences the object before calling it.

Synopsis

```
template<class F>
constexpr indirect_adaptor<F> indirect(F f);
```

Semantics

```
assert(indirect(f)(xs...) == (*f)(xs...));
```

Requirements

F must be:

- MoveConstructible
- Dereferenceable

Example

```
#include <fit.hpp>
#include <cassert>
#include <memory>
using namespace fit;
```

```
struct sum
{
    template<class T, class U>
    T operator()(T x, U y) const
    {
        return x+y;
    }
};

int main() {
    int r = indirect(std::make_unique<sum>())(3,2);
    assert(r == 5);
}
```

3.1.12 infix

Header

```
#include <fit/infix.hpp>
```

Description

The `infix` function adaptor allows the function to be used as an infix operator. The operator must be placed inside the angle brackets(ie < and >).

Synopsis

```
template<class F>
constexpr infix_adaptor<F> infix(F f);
```

Semantics

```
assert(x <infix(f)> y == f(x, y));
```

Requirements

F must be:

- *BinaryCallable*
- MoveConstructible

Operator precedence

Infix operators have the precedence of relational operators. This means operators such as + or $\star$ have higher precedence:

```
assert((x + y <infix(f)> z) == ((x + y) <infix(f)> z));
assert((x * y <infix(f)> z) == ((x * y) <infix(f)> z));
```

However, operators such as | or == have lower precedence::

```
assert((x | y <infix(f)> z) == (x | (y <infix(f)> z)));
assert((x == y <infix(f)> z) == (x == (y <infix(f)> z)));
```

Also, infix operators have left-to-right associativity:

```
assert(x <infix(f)> y <infix(g)> z == ((x <infix(f)> y) <infix(g)> z));
```

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

struct plus_f
{
    template<class T, class U>
    T operator()(T x, U y) const
    {
        return x+y;
    }
};

int main() {
    constexpr infix_adaptor<plus_f> plus = {};
    int r = 3 <plus> 2;
    assert(r == 5);
}
```

3.1.13 lazy

Header

```
#include <fit/lazy.hpp>
```

Description

The `lazy` function adaptor returns a function object call wrapper for a function. Calling this wrapper is equivalent to invoking the function. It is a simple form of lambda expressions, but is constexpr friendly. By default, `lazy` captures all of its variables by value, just like `bind`. `std::ref` can be used to capture references instead.

Ultimately, calling `lazy(f)(x)` is the equivalent to calling `std::bind(f, x)` except the lazy version can be called in a constexpr context, as well. The `lazy` adaptor is compatible with `std::bind`, so most of the time `lazy` and `std::bind` can be used interchangeably.

Synopsis

```
template<class F>
constexpr lazy_adaptor<F> lazy(F f);
```

Semantics

```
assert(lazy(f)(xs...) == std::bind(f, xs...))
assert(lazy(f)(xs...)() == f(xs...))
assert(lazy(f)(_1)(x) == f(x))
assert(lazy(f)(lazy(g)(_1))(x) == f(g(x)))
```

Requirements

F must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

int main() {
    auto add = [](auto x, auto y) { return x+y; };
    auto increment = lazy(add)(_1, 1);
    assert(increment(5) == 6);
}
```

References

3.1.14 match

Header

```
#include <fit/match.hpp>
```

Description

The `match` function adaptor combines several functions together and resolves which one should be called by using C++ overload resolution. This is different than the *conditional* adaptor which resolves them based on order.

Synopsis

```
template<class... Fs>
constexpr match_adaptor<Fs...> match(Fs...fs);
```

Requirements

Fs must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
using namespace fit;

struct int_class
{
    int operator()(int) const
    {
        return 1;
    }
};

struct foo
{};

struct foo_class
{
    foo operator()(foo) const
    {
        return foo();
    }
};

typedef match_adaptor<int_class, foo_class> fun;

static_assert(std::is_same<int, decltype(fun()(1))>::value, "Failed match");
static_assert(std::is_same<foo, decltype(fun()(foo()))>::value, "Failed match");

int main() {}
```

References

- POO51 - Proposal for C++ Proposal for C++ generic overload function

3.1.15 mutable

Header

```
#include <fit/mutable.hpp>
```

Description

The `mutable` function adaptor allows using a non-const function object inside of a const-function object. In Fit, all the function adaptors use `const` call overloads, so if there is a function that has a non-const call operator, it couldn't be used directly. So, `mutable_` allows the function to be used inside of the call operator.

NOTE: This function should be used with caution since many functions are copied, so relying on some internal shared state can be error-prone.

Synopsis

```
template<class F>
mutable_adaptor<F> mutable_(F f)
```

Requirements

F must be:

- *MutableFunctionObject*
- MoveConstructible

3.1.16 partial

Header

```
#include <fit/partial.hpp>
```

Description

The `partial` function adaptor allows partial application of the function. If the function can not be called with all the parameters, it will return another function. It will repeatedly do this until the function can finally be called. By default, `partial` captures all of its variables by value, just like bind. As such all parameters must be `MoveConstructible` when the function is aprtial application. `std::ref` can be used to capture references instead.

Synopsis

```
template<class F>
constexpr partial_adaptor<F> partial(F f);
```

Semantics

```
assert(partial(f)(xs...)(ys...) == f(xs..., ys...));
```

Requirements

F must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

struct sum
{
    template<class T, class U>
    T operator()(T x, U y) const
    {
        return x+y;
    }
};

int main() {
    assert(3 == partial(sum())(1)(2));
}
```

References

3.1.17 pipable

Header

```
#include <fit/pipable.hpp>
```

Description

The `pipable` function adaptor provides an extension method. The first parameter of the function can be piped into the function using the pipe `|` operator. This can be especially convenient when there are a lot of nested function calls. Functions that are made pipable can still be called the traditional way without piping in the first parameter.

Synopsis

```
template<class F>
constexpr pipable_adaptor<F> pipable(F f);
```

Semantics

```
assert(x | pipable(f)(ys...) == f(x, ys...));
```

Requirements

F must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

struct sum
{
    template<class T, class U>
    T operator()(T x, U y) const
    {
        return x+y;
    }
};

int main() {
    assert(3 == (1 | pipable(sum())(2)));
    assert(3 == pipable(sum())(1, 2));
}
```

References

- *Extension methods*

3.1.18 protect

Header

```
#include <fit/protect.hpp>
```

Description

The `protect` function adaptor can be used to make a bind expression be treated as a normal function instead. Both `bind` and *lazy* eargerly evaluates nested bind expressions. The `protect` adaptor masks the type so `bind` or *lazy* no longer recognizes the function as bind expression and evaluates it.

Synopsis

```
template<class F>
constexpr protect_adaptor<F> protect(F f);
```

Semantics

```
assert(lazy(f)(protect(lazy(g)(_1)))() == f(lazy(g)(_1)))
```

Requirements

F must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

int main() {
    auto lazy_id = lazy(identity)(_1);
    auto lazy_apply = lazy(apply)(protect(lazy_id), _1);
    assert(lazy_apply(3) == 3);
}
```

See Also

- *lazy*

3.1.19 result

Header

```
#include <fit/result.hpp>
```

Description

The `result` function adaptor sets the return type for the function, which can be useful when dealing with multiple overloads. Since the return type is no longer dependent on the parameters passed to the function, the `result_adaptor` provides a nested `result_type` that is the return type of the function.

Synopsis

```
template<class Result, class F>
constexpr result_adaptor<Result, F> result(F f);
```

Requirements

F must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>

struct id
{
    template<class T>
    T operator()(T x) const
    {
        return x;
    }
};

int main() {
    auto int_result = fit::result<int>(id());
    static_assert(std::is_same<decltype(int_result(true)), int>::value, "Not the same␣
↪type");
}
```

3.1.20 reveal

Header

```
#include <fit/reveal.hpp>
```

Description

The `reveal` function adaptor helps shows the error messages that get masked on some compilers. Sometimes an error in a function that causes a substitution failure, will remove the function from valid overloads. On compilers without a backtrace for substitution failure, this will mask the error inside the function. The `reveal` adaptor will expose these error messages while still keeping the function SFINAE-friendly.

Sample

If we take the `print` example from the quick start guide like this:

```
namespace adl {

using std::begin;

template<class R>
auto adl_begin(R&& r) FIT_RETURNS(begin(r));
}
```

```
FIT_STATIC_LAMBDA_FUNCTION(for_each_tuple) = [](const auto& sequence, auto f) FIT_
↪RETURNS
(
    fit::unpack(fit::by(f))(sequence)
);

auto print = fit::fix(fit::conditional(
    [](auto, const auto& x) -> decltype(std::cout << x, void())
    {
        std::cout << x << std::endl;
    },
    [](auto self, const auto& range) -> decltype(self(*adl::adl_begin(range)), void())
    {
        for(const auto& x:range) self(x);
    },
    [](auto self, const auto& tuple) -> decltype(for_each_tuple(tuple, self), void())
    {
        return for_each_tuple(tuple, self);
    }
));
```

Which prints numbers and vectors:

```
print(5);

std::vector<int> v = { 1, 2, 3, 4 };
print(v);
```

However, if we pass a type that can't be printed, we get an error like this:

```
print.cpp:49:5: error: no matching function for call to object of type 'fit::fix_
↪adaptor<fit::conditional_adaptor<(lambda at print.cpp:29:9), (lambda at print.
↪cpp:33:9), (lambda at print.cpp:37:9)> >'
    print(foo{});
    ^~~~~
fix.hpp:158:5: note: candidate template ignored: substitution failure [with Ts = <foo>
↪]: no matching function for call to object of type 'const fit::conditional_adaptor
↪<(lambda at
      print.cpp:29:9), (lambda at print.cpp:33:9), (lambda at print.cpp:37:9)>'
    operator()(Ts&&... xs) const FIT_SFINAE_RETURNS
```

Which is short and gives very little information why it can't be called. It doesn't even show the overloads that were try. However, using the `reveal` adaptor we can get more info about the error like this:

```
print.cpp:49:5: error: no matching function for call to object of type 'fit::reveal_
↪adaptor<fit::fix_adaptor<fit::conditional_adaptor<(lambda at print.cpp:29:9),␣
↪(lambda at print.cpp:33:9),
      (lambda at print.cpp:37:9)> >, fit::fix_adaptor<fit::conditional_adaptor
↪<(lambda at print.cpp:29:9), (lambda at print.cpp:33:9), (lambda at print.cpp:37:9)>
↪ > >'
    fit::reveal(print)(foo{});
    ^~~~~~~~~~~~~~~~~~
reveal.hpp:149:20: note: candidate template ignored: substitution failure [with Ts =
↪<foo>, $1 = void]: no matching function for call to object of type '(lambda at␣
↪print.cpp:29:9)'
    constexpr auto operator()(Ts&&... xs) const
                   ^
```

```
reveal.hpp:149:20: note: candidate template ignored: substitution failure [with Ts =
↪<foo>, $1 = void]: no matching function for call to object of type '(lambda at␣
↪print.cpp:33:9)'
    constexpr auto operator()(Ts&&... xs) const
                   ^
reveal.hpp:149:20: note: candidate template ignored: substitution failure [with Ts =
↪<foo>, $1 = void]: no matching function for call to object of type '(lambda at␣
↪print.cpp:37:9)'
    constexpr auto operator()(Ts&&... xs) const
                   ^
fix.hpp:158:5: note: candidate template ignored: substitution failure [with Ts = <foo>
↪]: no matching function for call to object of type 'const fit::conditional_adaptor
↪<(lambda at
      print.cpp:29:9), (lambda at print.cpp:33:9), (lambda at print.cpp:37:9)>'
    operator()(Ts&&... xs) const FIT_SFINAE_RETURNS
```

So now the error has a note for each of the lambda overloads it tried. Of course this can be improved even further by providing custom reporting of failures.

Synopsis

```
template<class F>
reveal_adaptor<F> reveal(F f);
```

Requirements

F must be:

- *ConstCallable*
- MoveConstructible

Reporting Failures

By default, `reveal` reports the substitution failure by trying to call the function. However, more detail expressions can be be reported from a template alias by using `as_failure`. This is done by defining a nested `failure` struct in the function object and then inheriting from `as_failure`. Also multiple failures can be reported by using `with_failures`.

Synopsis

```
// Report failure by instantiating the Template
template<template<class...> class Template>
struct as_failure;

// Report multiple falures
template<class... Failures>
struct with_failures;

// Report the failure for each function
template<class... Fs>
struct failure_for;
```

```
// Get the failure of a function
template<class F>
struct get_failure;
```

Example

```
#include <fit.hpp>
#include <cassert>

struct sum_f
{
    template<class T, class U>
    using sum_failure = decltype(std::declval<T>()+std::declval<U>());

    struct failure
    : fit::as_failure<sum_failure>
    {};

    template<class T, class U>
    auto operator()(T x, U y) const FIT_RETURNS(x+y);
};

int main() {
    assert(sum_f()(1, 2) == 3);
}
```

3.1.21 reverse_fold

Header

```
#include <fit/reverse_fold.hpp>
```

Description

The `reverse_fold` function adaptor uses a binary function to apply a reverse [fold](https://en.wikipedia.org/wiki/Fold_%28higher-order_function%29)(ie right fold in functional programming terms) operation to the arguments passed to the function. Additionally, an optional initial state can be provided, otherwise the first argument is used as the initial state.

The arguments to the binary function, take first the state and then the argument.

Synopsis

```
template<class F, class State>
constexpr reverse_fold_adaptor<F, State> reverse_fold(F f, State s);

template<class F>
constexpr reverse_fold_adaptor<F> reverse_fold(F f);
```

Semantics

```
assert(reverse_fold(f, z)() == z);
assert(reverse_fold(f, z)(x, xs...) == f(reverse_fold(f, z)(xs...), x));
assert(reverse_fold(f)(x) == x);
assert(reverse_fold(f)(x, xs...) == f(reverse_fold(f)(xs...), x));
```

Requirements

State must be:

- CopyConstructible

F must be:

- *BinaryCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>

struct max_f
{
    template<class T, class U>
    constexpr T operator()(T x, U y) const
    {
        return x > y ? x : y;
    }
};

int main() {
    assert(fit::reverse_fold(max_f())(2, 3, 4, 5) == 5);
}
```

References

- *Projections*
- *Variadic print*

3.1.22 rotate

Header

```
#include <fit/rotate.hpp>
```

Description

The `rotate` function adaptor moves the first parameter to the last parameter.

Synopsis

```
template<class F>
rotate_adaptor<F> rotate(F f);
```

Semantics

```
assert(rotate(f)(x, xs...) == f(xs..., x));
```

Requirements

F must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>

int main() {
    int r = fit::rotate(fit::_ - fit::_)(2, 5);
    assert(r == 3);
}
```

3.1.23 static

Header

```
#include <fit/static.hpp>
```

Description

The `static_` adaptor is a static function adaptor that allows any default-constructible function object to be static-initialized. Functions initialized by `static_` cannot be used in `constexpr` functions. If the function needs to be statically initialized and called in a `constexpr` context, then a `constexpr` constructor needs to be used rather than `static_`.

Synopsis

```
template<class F>
class static_;
```

Requirements

F must be:

- *ConstFunctionObject*
- DefaultConstructible

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

// In C++ this class can't be static-initialized, because of the non-
// trivial default constructor.
struct times_function
{
    double factor;
    times_function() : factor(2)
    {}
    template<class T>
    T operator()(T x) const
    {
        return x*factor;
    }
};

static constexpr static_<times_function> times2 = {};

int main() {
    assert(6 == times2(3));
}
```

3.1.24 unpack

Header

```
#include <fit/unpack.hpp>
```

Description

The `unpack` function adaptor takes a sequence and uses the elements of the sequence for the arguments to the function. Multiple sequences can be passed to the function. All elements from each sequence will be passed into the function.

Synopsis

```
template<class F>
unpack_adaptor<F> unpack(F f);
```

Requirements

F must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

struct sum
{
    template<class T, class U>
    T operator()(T x, U y) const
    {
        return x+y;
    }
};

int main() {
    int r = unpack(sum())(std::make_tuple(3,2));
    assert(r == 5);
}
```

References

- std::apply - C++17 function to unpack a tuple
- *unpack_sequence*

3.2 Decorators

3.2.1 capture

Header

```
#include <fit/capture.hpp>
```

Description

The `capture` function decorator is used to capture values in a function. It provides more flexibility in capturing than the lambda capture list in C++. It provides a way to do move and perfect capturing. The values captured are prepended to the argument list of the function that will be called.

Synopsis

```
// Capture by decaying each value
template<class... Ts>
constexpr auto capture(Ts&&... xs);

// Capture lvalues by reference and rvalue reference by reference
template<class... Ts>
constexpr auto capture_forward(Ts&&... xs);

// Capture lvalues by reference and rvalues by value.
template<class... Ts>
constexpr auto capture_basic(Ts&&... xs);
```

Semantics

```
assert(capture(xs...)(f)(ys...) == f(xs..., ys...));
```

Example

```
#include <fit.hpp>
#include <cassert>

struct sum_f
{
    template<class T, class U>
    T operator()(T x, U y) const
    {
        return x+y;
    }
};

int main() {
    auto add_one = fit::capture(1)(sum_f());
    assert(add_one(2) == 3);
}
```

3.2.2 if

Header

```
#include <fit/if.hpp>
```

Description

The `if_` function decorator makes the function callable if the boolean condition is true. The `if_c` version can be used to give a boolean condition directly(instead of relying on an integral constant).

When `if_` is false, the function is not callable. It is a subtitution failure to call the function.

Synopsis

```
template<class IntegralConstant>
constexpr auto if_(IntegralConstant);

template<bool B, class F>
constexpr auto if_c(F);
```

Requirements

IntegralConstant must be:

- IntegralConstant

F must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>

struct sum_f
{
    template<class T>
    int operator()(T x, T y) const
    {
        return fit::conditional(
            fit::if_(std::is_integral<T>())(fit::_ + fit::_),
            fit::always(0)
        )(x, y);
    }
};

int main() {
    assert(sum_f()(1, 2) == 3);
    assert(sum_f()("", "") == 0);
}
```

References

- *static_if*

3.2.3 limit

Header

```
#include <fit/limit.hpp>
```

Description

The `limit` function decorator annotates the function with the max number of parameters. The `limit_c` version can be used to give the max number directly(instead of relying on an integral constant). The parameter limit can be read by using the *function_param_limit* trait. Using `limit` is useful to improve error reporting with partially evaluated functions.

Synopsis

```
template<class IntegralConstant>
constexpr auto limit(IntegralConstant);

template<std::size_t N, class F>
constexpr auto limit_c(F);
```

Requirements

IntegralConstant must be:

- IntegralConstant

F must be:

- *ConstCallable*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

struct sum_f
{
    template<class T>
    int operator()(T x, T y) const
    {
        return x+y;
    }
};
FIT_STATIC_FUNCTION(sum) = limit_c<2>(sum_f());

int main() {
    assert(3 == sum(1, 2));
}
```

See Also

- *Partial function evaluation*
- *function_param_limit*

3.2.4 repeat

Header

```
#include <fit/repeat.hpp>
```

Description

The `repeat` function decorator will repeatedly apply a function a given number of times.

Synopsis

```
template<class Integral>
constexpr auto repeat(Integral);
```

Semantics

```
assert(repeat(std::integral_constant<int, 0>{})(f)(xs...) == f(xs...));
assert(repeat(std::integral_constant<int, 1>{})(f)(xs...) == f(f(xs...)));
assert(repeat(0)(f)(xs...) == f(xs...));
assert(repeat(1)(f)(xs...) == f(f(xs...)));
```

Requirements

Integral must be:

- Integral

Or:

- IntegralConstant

Example

```
#include <fit.hpp>
#include <cassert>

struct increment
{
    template<class T>
    constexpr T operator()(T x) const
    {
        return x + 1;
    }
};

int main() {
    auto increment_by_5 = fit::repeat(std::integral_constant<int, 5>())(increment());
    assert(increment_by_5(1) == 6);
}
```

3.2.5 repeat_while

Header

```
#include <fit/repeat_while.hpp>
```

Description

The `repeat_while` function decorator will repeatedly apply a function while the predicate returns a boolean that is true. If the predicate returns an `IntergralConstant` then the predicate is only evaluated at compile-time.

Synopsis

```
template<class Predicate>
constexpr auto repeat_while(Predicate predicate);
```

Requirements

Predicate must be:

- *ConstFunctionObject*
- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>

struct increment
{
    template<class T>
    constexpr std::integral_constant<int, T::value + 1> operator()(T) const
    {
        return std::integral_constant<int, T::value + 1>();
    }
};

struct not_6
{
    template<class T>
    constexpr std::integral_constant<bool, (T::value != 6)>
    operator()(T) const
    {
        return std::integral_constant<bool, (T::value != 6)>();
    }
};

typedef std::integral_constant<int, 1> one;
typedef std::integral_constant<int, 6> six;

int main() {
```

```
    auto increment_until_6 = fit::repeat_while(not_6())(increment());
    static_assert(std::is_same<six, decltype(increment_until_6(one()))>::value, "Error
↪");
}
```

3.3 Functions

3.3.1 always

Header

```
#include <fit/always.hpp>
```

Description

The `always` function returns a function object that will always return the value given to it, no matter what parameters are passed to the function object. The nullary version(i.e. `always(void)`) will return `void`. On compilers, that don't support constexpr functions returning `void`, a private empty type is returned instead. This return type is specified as `FIT_ALWAYS_VOID_RETURN`.

Synopsis

```
template<class T>
constexpr auto always(T value);

template<class T>
constexpr auto always(void);
```

Semantics

```
assert(always(x)(xs...) == x);
```

Requirements

T must be:

- CopyConstructible

Example

```
#include <fit.hpp>
#include <algorithm>
#include <cassert>
using namespace fit;

int main() {
```

```
    int ten = 10;
    assert( always(ten)(1,2,3,4,5) == 10 );
}

// Count all
template<class Iterator, class T>
auto count(Iterator first, Iterator last)
{
    return std::count_if(first, last, always(true));
}
```

3.3.2 arg

Header

```
#include <fit/arg.hpp>
```

Description

The `arg` function returns a function object that returns the Nth argument passed to it. It actually starts at 1, so it is not the zero-based index of the argument.

Synopsis

```
template<class IntegralConstant>
constexpr auto arg(IntegralConstant);

template<std::size_t N, class... Ts>
constexpr auto arg_c(Ts&&...);
```

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

int main() {
    assert(arg(std::integral_constant<int, 3>())(1,2,3,4,5) == 3);
}
```

3.3.3 construct

Header

```
#include <fit/construct.hpp>
```

Description

The `construct` function returns a function object that will construct the object when the called. A template can also be given, which it will deduce the parameters to the template. The `construct_meta` can be used to construct the object from a metafunction.

Synopsis

```
// Construct by decaying each value
template<class T>
constexpr auto construct();

template<template<class...> class Template>
constexpr auto construct();

// Construct by deducing lvalues by reference and rvalue reference by reference
template<class T>
constexpr auto construct_forward();

template<template<class...> class Template>
constexpr auto construct_forward();

// Construct by deducing lvalues by reference and rvalues by value.
template<class T>
constexpr auto construct_basic();

template<template<class...> class Template>
constexpr auto construct_basic();

// Construct by deducing the object from a metafunction
template<class MetafunctionClass>
constexpr auto construct_meta();

template<template<class...> class MetafunctionTemplate>
constexpr auto construct_meta();
```

Semantics

```
assert(construct<T>()(xs...) == T(xs...));
assert(construct<Template>()(xs...) == Template<decltype(xs)...>(xs...));
assert(construct_meta<MetafunctionClass>()(xs...) == MetafunctionClass::apply
↪<decltype(xs)...>(xs...));
assert(construct_meta<MetafunctionTemplate>()(xs...) == MetafunctionTemplate
↪<decltype(xs)...>::type(xs...));
```

Requirements

MetafunctionClass must be a:

- *MetafunctionClass*

MetafunctionTemplate<Ts...> must be a:

- *Metafunction*

T, Template<Ts..>, MetafunctionClass::apply<Ts...>, and MetafunctionTemplate<Ts...>::type must be:

- MoveConstructible

Example

```
#include <fit.hpp>
#include <cassert>
#include <vector>

int main() {
    auto v = fit::construct<std::vector<int>>()(5, 5);
    assert(v.size() == 5);
}
```

3.3.4 decay

Header

```
#include <fit/decay.hpp>
```

Description

The `decay` function is a unary function object that returns whats given to it after decaying its type.

Synopsis

```
struct
{
    template<class T>
    constexpr typename decay<T>::type operator()(T&& x) const
    {
        return fit::forward<T>(x);
    }
} decay;
```

References

- n3255 - Proposal for `decay_copy`

3.3.5 identity

Header

```
#include <fit/identity.hpp>
```

Description

The `identity` function is an unary function object that returns whats given to it.

Semantics

```
assert(identity(x) == x);
```

Synopsis

```
template<class T>
constexpr T identity(T&& x);
```

3.3.6 placeholders

Header

```
#include <fit/placeholders.hpp>
```

Description

The placeholders provide `std::bind` compatible placeholders that additionally provide basic C++ operators that creates bind expressions. Each bind expression supports `constexpr` function evaluation.

Synopsis

```
namespace placeholders {
    placeholder<1> _1 = {};
    placeholder<2> _2 = {};
    placeholder<3> _3 = {};
    placeholder<4> _4 = {};
    placeholder<5> _5 = {};
    placeholder<6> _6 = {};
    placeholder<7> _7 = {};
    placeholder<8> _8 = {};
    placeholder<9> _9 = {};
}
```

Operators

- Binary operators: +,-,*,/,%,>>,<<,>,<,<=,>=,==,!=,&,^,|,&&,||
- Assign operators: +=,-=,*=,/=,%=,>>=,<<=,&=,|=,^=
- Unary operators: !,~,+,-,*,++,–

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

int main() {
    auto sum = _1 + _2;
    assert(3 == sum(1, 2));
}
```

3.3.7 unamed placeholder

Header

```
#include <fit/placeholders.hpp>
```

Description

The unamed placeholder can be used to build simple functions from C++ operators.

Note: The function produced by the unamed placeholder is not a bind expression.

Synopsis

```
namespace placeholders {
    /* unspecified */ _ = {};
}
```

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

int main() {
    auto sum = _ + _;
    assert(3 == sum(1, 2));
}
```

3.4 Traits

3.4.1 function_param_limit

Header

```
#include <fit/function_param_limit.hpp>
```

Description

The `function_param_limit` metafunction retrieves the maximum number of parameters for a function. For function pointers it returns the number of parameters. Everything else, it returns `SIZE_MAX`, but this can be changed by annotating the function with the *limit* decorator.

This is a type trait that inherits from `std::integral_constant`.

Synopsis

```
template<class F>
struct function_param_limit
: std::integral_constant<std::size_t, ...>
{};
```

See Also

- *Partial function evaluation*
- *limit*

3.4.2 is_callable

Header

```
#include <fit/is_callable.hpp>
```

Description

The `is_callable` metafunction checks if the function is callable with certain parameters.

Requirements

F must be:

- *Callable*

Synopsis

```
template<class F, class... Ts>
struct is_callable;
```

Example

```
#include <fit.hpp>
using namespace fit;

struct is_callable_class
{
    void operator()(int) const
    {
    }
};
static_assert(is_callable<is_callable_class, int>(), "Not callable");

int main() {}
```

3.4.3 is_unpackable

Header

```
#include <fit/is_unpackable.hpp>
```

This is a trait that can be used to detect whether the type can be called with `unpack`.

Synopsis

```
template<class T>
struct is_unpackable;
```

Example

```
#include <fit.hpp>
#include <cassert>

int main() {
    static_assert(fit::is_unpackable<std::tuple<int>>::value, "Failed");
}
```

3.4.4 unpack_sequence

Header

```
#include <fit/unpack_sequence.hpp>
```

How to unpack a sequence can be defined by specializing `unpack_sequence`. By default, `std::tuple` is already specialized. To implement this, one needs to provide a static `apply` function which will unpack the sequence to the parameters of the function.

Synopsis

```
template<class Sequence, class=void>
struct unpack_sequence;
```

Example

```
#include <fit.hpp>
#include <cassert>

struct my_sequence
{
    int x;
    int y;
};

namespace fit {
    template<>
    struct unpack_sequence<my_sequence>
    {
        template<class F, class Sequence>
        constexpr static auto apply(F&& f, Sequence&& s) FIT_RETURNS
        (
            f(s.x, s.y)
        );
    };
} // namespace fit

int main() {
}
```

See Also

- *unpack*
- *is_unpackable*

3.5 Utilities

3.5.1 apply

Header

```
#include <fit/apply.hpp>
```

Description

The `apply` function calls the function given to it with its arguments.

Synopsis

```
template<class F, class... Ts>
constexpr auto apply(F&& f, Ts&&... xs);
```

Semantics

```
assert(apply(f)(xs...) == f(xs...));
assert(fold(apply, f)(x, y, z) == f(x)(y)(z));
```

Requirements

F must be:

- *Callable*

Example

```
#include <fit.hpp>
#include <cassert>

struct sum_f
{
    template<class T, class U>
    T operator()(T x, U y) const
    {
        return x+y;
    }
};

int main() {
    assert(fit::apply(sum_f(), 1, 2) == 3);
}
```

3.5.2 apply_eval

Header

```
#include <fit/apply_eval.hpp>
```

Description

The `apply_eval` function work like *apply*, except it calls *eval* on each of its arguments. Each *eval* call is always ordered from left-to-right.

Synopsis

```
template<class F, class... Ts>
constexpr auto apply_eval(F&& f, Ts&&... xs);
```

Semantics

```
assert(apply_eval(f)(xs...) == f(eval(xs)...));
```

Requirements

F must be:

- *ConstCallable*

Ts must be:

- *EvaluatableFunctionObject*

Example

```
#include <fit.hpp>
#include <cassert>

struct sum_f
{
    template<class T, class U>
    T operator()(T x, U y) const
    {
        return x+y;
    }
};

int main() {
    assert(fit::apply_eval(sum_f(), []{ return 1; }, []{ return 2; }) == 3);
}
```

3.5.3 eval

Header

```
#include <fit/eval.hpp>
```

Description

The `eval` function will evaluate a "thunk". This can be either a nullary function or it can be a unary function that takes the identity function as the first parameter(which is helpful to delay compile-time checking). Also, additional parameters can be passed to `eval` to delay compiliation(so that result can depend on template parameters).

Synopsis

```
template<class F, class... Ts>
constexpr auto eval(F&& f, Ts&&...);
```

Requirements

F must be:

- *EvaluatableFunctionObject*

Example

```
#include <fit.hpp>
#include <cassert>

int main() {
    assert(fit::eval([]{ return 3; }) == 3);
}
```

References

- POO51 - Proposal for C++ Proposal for C++ generic overload function
- *static_if*
- *Ordering evaluation of arguments*

3.5.4 FIT_STATIC_FUNCTION

Header

```
#include <fit/function.hpp>
```

Description

The `FIT_STATIC_FUNCTION` macro allows initializing a function object from a `constexpr` expression. It uses the best practices as outlined in N4381. This includes using `const` to avoid global state, compile-time initialization of the function object to avoid the static initialization order fiasco, and an external address of the function object that is the same across translation units to avoid possible One-Definition-Rule(ODR) violations.

In C++17, this achieved using the `inline` keyword. However, on older compilers it is initialized using a reference to a static member variable. The static member variable is default constructed, as such the user variable is always default constructed regardless of the expression.

By default, all functions defined with `FIT_STATIC_FUNCTION` use the *reveal* adaptor to improve error messages.

Example

```
#include <fit.hpp>
#include <cassert>

struct sum_f
{
    template<class T, class U>
    T operator()(T x, U y) const
    {
        return x+y;
    }
};

FIT_STATIC_FUNCTION(sum) = sum_f();
FIT_STATIC_FUNCTION(partial_sum) = fit::partial(sum_f());

int main() {
    assert(sum(1, 2) == partial_sum(1)(2));
}
```

3.5.5 FIT_STATIC_LAMBDA

Header

```
#include <fit/lambda.hpp>
```

Description

The `FIT_STATIC_LAMBDA` macro allows initializing non-capturing lambdas at compile-time in a `constexpr` expression.

Example

```
#include <fit.hpp>
#include <cassert>

const constexpr auto add_one = FIT_STATIC_LAMBDA(int x)
{
    return x + 1;
};

int main() {
    assert(3 == add_one(2));
}
```

3.5.6 FIT_STATIC_LAMBDA_FUNCTION

Header

```
#include <fit/lambda.hpp>
```

Description

The `FIT_STATIC_LAMBDA_FUNCTION` macro allows initializing a global function object that contains non-capturing lambdas. It also ensures that the global function object has a unique address across translation units. This helps prevent possible ODR-violations.

By default, all functions defined with `FIT_STATIC_LAMBDA_FUNCTION` use the `fit::reveal` adaptor to improve error messages.

Example

```
#include <fit.hpp>
#include <cassert>

FIT_STATIC_LAMBDA_FUNCTION(add_one) = [](int x)
{
    return x + 1;
};
int main() {
    assert(3 == add_one(2));
}
```

3.5.7 FIT_LIFT

Header

```
#include <fit/lift.hpp>
```

Description

The macros `FIT_LIFT` and `FIT_LIFT_CLASS` provide a lift operator that will wrap a template function in a function object so it can be passed to higher-order functions. The `FIT_LIFT` macro will wrap the function using a generic lambda. As such, it will not preserve `constexpr`. The `FIT_LIFT_CLASS` can be used to declare a class that will wrap function. This will preserve `constexpr` and it can be used on older compilers that don't support generic lambdas yet.

Limitation

In C++14, `FIT_LIFT` doesn't support `constexpr` due to using a generic lambda. Instead, `FIT_LIFT_CLASS` can be used. In C++17, there is no such limitation.

Synopsis

```
// Wrap the function in a generic lambda
#define FIT_LIFT(...)

// Declare a class named `name` that will forward to the function
#define FIT_LIFT_CLASS(name, ...)
```

Example

```
#include <fit.hpp>
#include <cassert>
#include <algorithm>

// Declare the class `max_f`
FIT_LIFT_CLASS(max_f, std::max);

int main() {
    auto my_max = FIT_LIFT(std::max);
    assert(my_max(3, 4) == std::max(3, 4));
    assert(max_f()(3, 4) == std::max(3, 4));
}
```

3.5.8 pack

Header

```
#include <fit/pack.hpp>
```

Description

The `pack` function returns a higher order function object that takes a function that will be passed the initial elements. The function object is a sequence that can be unpacked with `unpack_adaptor` as well. Also, `pack_join` can be used to join multiple packs together.

Synopsis

```
// Decay everything before capturing
template<class... Ts>
constexpr auto pack(Ts&&... xs);

// Capture lvalues by reference and rvalue reference by reference
template<class... Ts>
constexpr auto pack_forward(Ts&&... xs);

// Capture lvalues by reference and rvalues by value.
template<class... Ts>
constexpr auto pack_basic(Ts&&... xs);

// Join multiple packs together
```

```
template<class... Ts>
constexpr auto pack_join(Ts&&... xs);
```

Semantics

```
assert(pack(xs...)(f) == f(xs...));
assert(unpack(f)(pack(xs...)) == f(xs...));

assert(pack_join(pack(xs...), pack(ys...)) == pack(xs..., ys...));
```

Example

```
#include <fit.hpp>
#include <cassert>
using namespace fit;

struct sum
{
    template<class T, class U>
    T operator()(T x, U y) const
    {
        return x+y;
    }
};

int main() {
    int r = pack(3, 2)(sum());
    assert(r == 5);
}
```

See Also

- *unpack*

3.5.9 FIT_RETURNS

Header

```
#include <fit/returns.hpp>
```

Description

The `FIT_RETURNS` macro defines the function as the expression equivalence. It does this by deducing `noexcept` and the return type by using a trailing `decltype`. Instead of repeating the expression for the return type, `noexcept` clause and the function body, this macro will reduce the code duplication from that.

Note: The expression used to deduce the return the type will also constrain the template function and deduce `noexcept` as well, which is different behaviour than using C++14's return type deduction.

Synopsis

```
#define FIT_RETURNS(...)
```

Example

```
#include <fit.hpp>
#include <cassert>

template<class T, class U>
auto sum(T x, U y) FIT_RETURNS(x+y);

int main() {
    assert(3 == sum(1, 2));
}
```

Incomplete this

Description

On some non-conformant compilers, such as gcc, the `this` variable cannot be used inside the `FIT_RETURNS` macro because it is considered an incomplete type. So the following macros are provided to help workaround the issue.

Synopsis

```
// Declares the type of the `this` variable
#define FIT_RETURNS_CLASS(...)
// Used to refer to the `this` variable in the FIT_RETURNS macro
#define FIT_THIS
// Used to refer to the const `this` variable in the FIT_RETURNS macro
#define FIT_CONST_THIS
```

Example

```
#include <fit.hpp>
#include <cassert>

struct add_1
{
    int a;
    add_1() : a(1) {}

    FIT_RETURNS_CLASS(add_1);

    template<class T>
    auto operator()(T x) const
    FIT_RETURNS(x+FIT_CONST_THIS->a);
};

int main() {
```

```
    assert(3 == add_1()(2));
}
```

Mangling overloads

Description

On older compilers some operations done in the expressions cannot be properly mangled. These macros help provide workarounds for these operations on older compilers.

Synopsis

```
// Explicitly defines the type for name mangling
#define FIT_MANGLE_CAST(...)
// C cast for name mangling
#define FIT_RETURNS_C_CAST(...)
// Reinterpret cast for name mangling
#define FIT_RETURNS_REINTERPRET_CAST(...)
// Static cast for name mangling
#define FIT_RETURNS_STATIC_CAST(...)
// Construction for name mangling
#define FIT_RETURNS_CONSTRUCT(...)
```

3.5.10 tap

Header

```
#include <fit/tap.hpp>
```

Description

The `tap` function invokes a function on the first argument passed in and then returns the first argument. This is useful in a chain of pipable function to perform operations on intermediate results. As a result, this function is *pipable*.

Synopsis

```
template<class T, class F>
pipable constexpr T tap(T&& x, const F& f);
```

Requirements

F must be:

- *UnaryCallable*

Example

```
#include <fit.hpp>
#include <cassert>
#include <iostream>
using namespace fit;

struct sum_f
{
    template<class T, class U>
    T operator()(T x, U y) const
    {
        return x+y;
    }
};

const pipable_adaptor<sum_f> sum = {};
int main() {
    // Prints 3
    int r = 1 | sum(2) | tap([](int i) { std::cout << i; }) | sum(2);
    assert(r == 5);
}
```

CHAPTER 4

Configurations

There are several configuration macros that control the behavior of the Fit library.

Name	Description
`FIT_CHECK_UNPACK_SEQUENCE`	Unpack has extra checks to ensure that the function will be invoked with the sequence. This extra check can help improve error reporting but it can slow down compilation. This is enabled by default.
`FIT_NO_EXPRESSION_SFINAE`	This controls whether the Fit library will use expression SFINAE to detect the callability of functions. On MSVC, this is enabled by default, since it does not have full support for expression SFINAE.
`FIT_RECURSIVE_CONSTEXPR_LIMIT`	Because C++ instantiates *constexpr* functions eagerly, recursion with *constexpr* functions can cause the compiler to reach its internal limits. The setting is used by Fit to set a limit on recursion depth to avoid infinite template instantiations. The default is 16, but increasing the limit can increase compile times.

CHAPTER 5

Discussion

5.1 Partial function evaluation

Many of the adaptors(such as *partial* or *pipable*) in the library supports optional partial evaluation of functions. For example, if we have the `sum` function adapted with the `partial` adaptor:

```
auto sum = partial([](int x, int y)
{
    return x+y;
});
```

So if we write `sum(1, 2)` it will return 3, however, if we write `sum(1)` it will return a new function, which when called again, it will evaluate the function and return 3:

```
int i = sum(1, 2); // Returns 3
auto f = sum(1);
int j = f(2); // returns 3
```

Of course due to limitations in C++, deciding whether evaluate the function or to partially evaluated it, is based on the callability of the function and not arity. So if we call `sum(1, 2, 3)`, it will return a function:

```
auto f = sum(1, 2, 3);
```

However, this can get out of hande as the function `f` will never be evaluated. Plus, it would be nice to produce an error at the point of calling the function rather than a confusing error of trying to use a partial function. The *limit* decorator lets us annotate the function with the max arity:

```
auto sum = partial(limit_c<2>([](int x, int y)
{
    return x+y;
}));
```

So now if we call `sum(1, 2, 3)`, we will get a compiler error. So this improves the situation, but it is not without its limitations. For example if we were to define a triple sum using the *pipable* adaptor:

```
auto sum = pipable(limit_c<3>([](int x, int y, int z)
{
    return x+y+z;
}));
```

So if we call `sum(1)`, there is no compiler error, not until we try to pipe in a value:

```
auto f = sum(1); // No error here
auto i = 2 | f; // Compile error
```

Of course, the goal may not be to use the pipable call, which could lead to some confusing errors. Currently, there is not a good solution to this.

5.2 FAQ

5.2.1 Q: Why is `const` required for the call operator on function objects?

Mutable function objects are not prohibited, they just need to be explicit by using the adaptor *mutable*. The main reason for this, is that it can lead to many suprising behaviours. Many times function objects are copied by value everywhere. For example,

```
struct counter
{
    int i;
    counter() : i(0)
    {}

    template<class... Ts>
    int operator()(Ts&&...)
    {
        return i++;
    }
};


counter c{};
by(mutable_(c))(1,1);
// Prints 0, not 2
std::cout << c.i << std::endl;
```

The example won't ever yield the expected value, because the function mutates a copy of the objects. Instead, `std::ref` should be used:

```
counter c{};
by(std::ref(c))(1,1);
// Prints 2
std::cout << c.i << std::endl;
```

Which will print the expected value.

Another reason why `const` is required is because of supporting `constexpr` on C++11 compilers. In C++11, `constexpr` implies `const`, so it would be impossible to provide a non-const overload for functions that is `constexpr`. Instead, `constexpr` would have to be made explicit. Considering the pitfalls of mutable function objects, it would be better to make mutability explicit rather than `constexpr`.

5.2.2 Q: Is the reinterpret cast in FIT_STATIC_LAMBDA undefined behaviour?

Not really, since the objects are empty, there is no data access. There is a static assert to guard against this restriction.

Now there could be an insane implementation where this doesn't work(perhaps the lambdas are not empty for some strange reason), which the library would have to apply a different technique to make it work. However, this is quite unlikely considering that C++ will probably get constexpr lambdas and inline variables in the future.

Alternatively, the factory pattern can be used instead of *FIT_STATIC_LAMBDA_FUNCTION*, which doesn't require an reinterpret cast:

```
struct sum_factory
{
    auto operator*() const
    {
        return [](auto x, auto y)
        {
            return x + y;
        };
    }
}

FIT_STATIC_FUNCTION(sum) = fit::indirect(sum_factory{});
```

CHAPTER 6

Acknowledgements

- Boost.Egg: A very powerful library for function objects in C++98.
  - Shunsuke Sogame
- Boost.Hana: A metaprogramming library with many functional constructs
  - Louis Dionne
- named-operators: A library to create named operators
  - Konrad Rudolph
- Pack/Unpack without Using Tuple
  - Jamboree
- for_each_argument
  - Sean Parent
- Suggested Design for Customization Points
  - Eric Niebler
- FC++: Functional Programming in C++
  - Brian McNamara and Yannis Smaragdakis
- Boost.Phoenix
  - Joel de Guzman, Dan Marsden, Thomas Heller, and John Fletcher

CHAPTER 7

License

Boost Software License - Version 1.0 - August 17th, 2003

Permission is hereby granted, free of charge, to any person or organization obtaining a copy of the software and accompanying documentation covered by this license (the "Software") to use, reproduce, display, distribute, execute, and transmit the Software, and to prepare derivative works of the Software, and to permit third-parties to whom the Software is furnished to do so, all subject to the following:

The copyright notices in the Software and this entire statement, including the above license grant, this restriction and the following disclaimer, must be included in all copies of the Software, in whole or in part, and all derivative works of the Software, unless such copies or derivative works are solely in the form of machine-executable object code generated by a source language processor.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE SOFTWARE BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER