Return values take a ”closure” walk

How to pass return values without specifying their type.

Calling a function within a context

```
void callWithin(const std::function<void()>& fn)
{
    ScopedContext context;
    try {
        fn();
    } catch (SomeException& e) {
        //  handle exception here
    }
}

void printLine(const std::string& text)
{
    std::cout << text << "\n";
}

callWithin([](){ printLine("Hello, CppCon"); });
```

Calling a function within a context as template

```
template <typename Callable>
void callWithin(const Callable& fn)
{
    ScopedContext context;
    fn();
}
```

Calling a function within a context as virtual function

```
class ContextManager
{
public:
    virtual void callWithin(const std::function<void()>& fn) = 0;
};
```

Function with a return value

```
double sum(double a, double b)
{
    return a + b;
}
```

```
double result = callWithin([](){ return sum(3.14, 2.71); });
```

Function with a return value (fixed type)

```
double callWithin(const std::function<double()>& fn)
{
    ScopedContext context;
    return fn();
}
```

Function with a return value (use given return type)

```
template <typename Callable>
auto callWithin(const Callable& fn) -> decltype(fn())
{
    callWithinImpl(fn);  // does not compile: function type mismatch
    // needs return value
}

void callWithinImpl(const std::function<void()>& fn);
```

Function with a return value (local variable)

```
template <typename Fn>
auto callWithin(const Fn& fn) -> decltype(fn())
{
    decltype(fn()) result{};
    ...
}

void callWithinImpl(const std::function<void()>& fn);
```

Function with a return value (wrapper captures result in closure)

```
template <typename Fn>
auto callWithin(const Fn& fn) -> decltype(fn())
{
    decltype(fn()) result{};
    auto wrapperFn = [&]()
    {
        result = fn();
    };
    …
}

void callWithinImpl(const std::function<void()>& fn);
```

Function with a return value (call wrapper, return saved result)

```
template <typename Fn>
auto callWithin(const Fn& fn) -> decltype(fn())
{
    decltype(fn()) result{};
    auto wrapperFn = [&]() -> void
    {
        result = fn();
    };
    callWithinImpl(wrapperFn);
    return result;
}

void callWithinImpl(const std::function<void()>& fn);
```

Member template and virtual function

```
class ContextManager
{
public:
    template <typename Fn>
    auto callWithin(const Fn& fn) -> decltype(fn())
    {
        decltype(fn()) result{};
        callWithinImpl([&](){ result = fn(); });
        return result;
    }
private:
    virtual void callWithinImpl(const std::function<void()>& fn) = 0;
};
// usage
double result = manager->callWithin([](){ return sum(3.14, 2.71); });
```

Return ”value” of a void() function

```
void printLine(const std::string& text)
{
    std::cout << text << "\n";
}

callWithin([](){ printLine("Hello, CppCon"); });
```

Return ”value” of a void() function (Compiler gives errors)

```
template <typename Fn>
auto callWithin(const Fn& fn) -> decltype(fn())  // deduced to ‘void’
{
    decltype(fn()) result{};    // variable declared ‘void’
    callWithinImpl([&](){ result = fn(); });  // undefined variable ‘result’
    return result;  // ‘void’ function returning a result
}
```

Return ”value” of a void() function (Metaprogramming)

```
template <typename Fn>
auto callWithin(const Fn& fn) -> decltype(fn())
{
    return _callWithin(fn, std::is_same<decltype(fn()), void>());
}
template <typename Fn>
void _callWithin(const Fn& fn, std::true_type)
{
   callWithinImpl([&] { fn(); });
}
template <typename Fn>
auto _callWithin(const Fn& fn, std::false_type) -> decltype(fn())
{
   decltype(fn()) result{};
   callWithinImpl([&] { result = fn(); });
   return result;
}
```

callWithin might fail

```
bool callWithinImpl(const std::function<void()>& fn)
{
    try
    {
        auto dbConnectionScope = database->openConnection(); // this might fail
        fn();
    }
    catch (DBException& e)
    {
        return false; // failure
    }
    return true; // ok
}
```

callWithin might fail (wrap result in boost::optional)

```
template <typename Fn>
auto callWithin(const Fn& fn) -> boost::optional<decltype(fn())>
{
    decltype(fn()) result{};
    bool ok = callWithinImpl([&]()
    {
        result = fn();
    });
    if (ok)
        return result;  // wrapped within boost::optional
    else
        return boost::none;
}
bool callWithinImpl(const std::function<void()>& fn);
```

callWithin might fail

```
auto result = callWithin([](){ return sum(1,2); });
if (result) // might be boost::none
{
    double resultValue = *result; // dereference boost::optional
    std::cout << resultValue << std::endl;
}
```

Got it?