Contract testing support

Document #:	P3183R0
Date:	2024-04-13
Project:	Programming Language C++
Audience:	SG21 Contracts

Reply-to:	Bengt Gustafsson <bengt.gustafsson@beamways.com>

1 Abstract

This proposal includes a few magic functions to perform *only* the pre or post conditions of a function given a set of function arguments.

2 Motivation

Contracts is a major feature targeting C++26. The current proposal is in [P2900R5]. Quite a lot of debate has been devoted to how to test the contract conditions for correctness, as a failed contract means either that the test program is aborted or that the function actually runs with the erroneous input, which can cause any type of behavior including UB. To offset this it has been proposed that the contract violation handler (which is linked into the test program) is allowed to throw an exception or perform a `longjmp` back to the test code calling the function under test. These methods have problems. Exceptions are problematic as we can't test contract conditions of `noexcept` functions while `longjmp` is problematic as it doesn't properly destroy the function parameters.

With this proposal one can easily test conditions of a function without running into trouble with `noexcept`or replacing the contract violation handler. There are also other use cases of separately checking conditions, for instance when performing an RPC call to a function in another process; checking the preconditions of the function to be called already in the calling process. It can also be used during development to check particular function calls while not checking contracts in general.

3 Proposal

This proposal provides the ability to perform only the contract checks for a function. This functionality is embodied in special magic *check functions*. Although magic standard library functions are generally not an advised strategy it seems that devoting a keyword to this purpose would be to ask too much. As a function call is the natural syntax for this functionality this is what is proposed:

```
template<auto F, typename... Args>
bool check_preconditions(Args&&... args);

template<auto F, typename R, typename... Args>
bool check_postconditions(const R& r, Args&&... args);
```

The `check_preconditions` function just performs all pre condition checks of F given the arguments and returns a bool which is true if they all pass. If any of the conditions throws this exception is propagated out of the `check_preconditions` call.

The `check_postconditions` function similarly performs all post-condition checks of F given the arguments and the return value r that F has supposedly returned.

If F is a member function pointer the first element of the Args pack must be an object from which the member function can be called, using the invoke paradigm. The contracts that get checked relate to the member function implementation in the declared type of the first pack element (Args…[0]) regardless of the actual type of the argument (which may be derived).

These functions, while magic, should be easy to implement in a compiler, especially if it already supports client side contract checks.

An advantage of this approach to contract testing is that these functions unconditionally evaluate the conditions, even if the compiler is instructed to not include the contract checking code in function implementations or their call sites and regardless of evaluation mode selected at runtime.

Note that the function pointer is a template parameter. This is important as it is not the type of the function (or member function) pointer that is relevant, but its exact identity. By making the function pointer a template parameter there is one instantiation per function which is what is needed to be able to generate the code for the contract checks correctly.

4 Example

Here is a simple example of a testing scenario. `CHECK` is a placeholder for the basic test macro of some testing framework.

```
void f(int x) pre(x > 0) post(r: r >= 0) {
    return sqrt(x);
}

// Check that the pre condition is correct.
CHECK(check_preconditions<f>(1));
CHECK(!check_preconditions<f>(0));
CHECK(!check_preconditions<f>(-1));

// Check post condition
CHECK(check_postconditions<f>(1, 0));
CHECK(check_postconditions<f>(0, 0));
CHECK(!check_postconditions<f>(-1, 0));
```

5 Overloaded functions

To use the check functions with overloaded functions `static_cast` of the function name is required in the template parameter list. Ideally the function should be selected based on the argument types `Args...` but this can't be done in C++23. Even the `__builtin_calltarget` operator of [P2825R0] does not solve this problem fully as the function itself must be a template parameter. A new test system macro would still be required:

```
#define CHECK_PRECONDITION_VIOLATION(F, ...) \
CHECK(!check_preconditions<__builtin_calltarget(F(__VA_ARGS__))>(__VA_ARGS__))
```

While this may not be so bad we don't really want to encourage macros. Note that the difference between `__builtin_calltarget` and `static_cast` is that `__builtin_calltarget` takes a list of function arguments and does full overload resolution while static_cast needs the exact parameter types, so static_cast is not tractable to use in this macro, we need `__builtin_calltarget`.

It would be possible to make the proposed functions *more* magic so that they do the overload resolution themselves. This would violate everything we know about function pointers and template parameters and would constitute some motivation for making the check functions language keywords. This proposal avoids this and suggests relying on `__builtin_calltarget` and test framework macros until we get a proper solution for passing overload sets around.

Another option could be to use reflection to basically do what `__builtin_calltarget` can do according to its proposal. However, to implement these functions using reflection as a pure library feature would require also the ability to reify the conditions of a function separately given the meta::info of the function. While not impossible this possibility is unlikely to be available in a MVP reflection feature, especially as contracts and reflection are targeted for the same standard revision.

6 Virtual functions

All currently proposed solutions for contracts on virtual functions check the contracts of the member function declaration in the declared type of the implicit object reference. This is what the check functions proposed here can check as the conditions are compiled into the check function instantiations. Some proposals also check the contracts of the virtual function that is actually called, but this can't be done by the check functions in this proposal as they don't actually call the function.

The system proposed in [P3169R0] only checks the contracts on the declared type and implements checking of subclass conditions by calling virtual methods overridden by subclasses from within the base class contract conditions. Such contracts can be tested using the check functions proposed here by providing different subclass objects as the first argument to the check function call.

7 Future extensions

This proposal was set up to match the functionality in the MVP contracts proposal, [P2900R5]. It can evolve with this proposal and it can also be extended in its own right.

7.1 Getting more information

It would be possible to complement the check functions with variants that take a `size_t& index` as an extra parameter and sets this value to the index of the condition that failed. If the check functions increment this value between each condition check the index variable value is valid even if an exception is thrown while evaluating a condition.

Another option is to have a `const char*& condition` parameter that receives a string representation of the failed contract condition (i. e. its source code). This has the drawbacks that more of the implementation is revealed by strings in the binary file and that the binary file size grows.

It is conceivable that an IDE may be able to convert the `index` to the corresponding string and display it as a tool tip or similar. To do this it could take help from contract related data stored with the debug information. This seems more appropriate than storing the strings in the data segment of the object file.

7.2 Assertions and Invariants

There seems to be no easy parallel to be made between the proposed check functions and something that could check `contract_assertion` conditions embedded in a function body.

Class invariants are very easy to test, once we get them.

```
template<typename C>
bool check_invariants(C& object);
```

The only problem with `check_invariants` is that the test code must be able to produce an object which violates the invariants, which is exactly what the class API should *not* be able to do. So this requires a special friend function for testing or similar to be able to set up member data values which are in violation of the invariants before calling `check_invariants`.

7.3 Postcondition captures

If captures done before the function runs are introduced into the standard it seems feasible to add handling of these into the `check_postconditions` function as those captures can only capture argument values, data member values and maybe global variable values when the function body starts executing. All of these values are available at the start of `check_postconditions` which means that the capture values can be computed and provided to the post check conditions without further action of the programmer.

8 Late References

[P3169R0] Jonas Persson. 2024-03-21. Inherited contracts.

https://wg21.link/p3169r0

9 References

[P2825R0] Gašper Ažman. 2023-03-15. calltarget(unevaluated-call-expression).
https://wg21.link/p2825r0

[P2900R5] Joshua Berne, Timur Doumler, Andrzej Krzemieński. 2024-02-15. Contracts for C++.
https://wg21.link/p2900r5