Compile time resolved contracts

Document #:	P3317R0
Date:	2024-05-22
Project:	Programming Language C++
Audience:	SG21 Contracts Working Group

Reply-to:	Jonas Persson <jonas.persson@iar.com>

Contents

1 Abstract

This paper tries to explore what is needed for static analysis of contracts how to ensure that all contract mechanisms of [P2900R5] and beyond will be designed in a way that allows that.

2 Motivation

For some types of applications terminating would be disastrous. But they will still leave enforce in the release build because incorrect data is even worse.

Others want their software checked, but there are no extra resources, memory or CPU, to run contract checks.

For these applications it is worth the effort to write the program in such a way that most contracts will be checked at compile time and any runtime cost and effect is elided.

But even for less demanding applications it adds a lot of confidence to know that many of your contracts has been proven correct already by the compiler.

Planned as a future extension is an assume mode that lets the optimizer assume that the contract is true and optimize based on that. It is not part of the MVP because it has dangerous implications in case that assumption

is not true. When contracts is resolved at compile time the optimizer has the same opportunity but without all the risks.

For all this to be an option we need to design contract so that static checks are possible and we need compilation modes to help us find where static analysis fails.

This is an important feature that should be considered when we expand the design of contracts.

It may be argued that this is better left as an QoI for tools vendors, but it is an essential part of working with contracts and it need to designed to play nice with the contracts in MVP and later extensions.

3 Diagnostics

In order to let the programmer know what the compiler finds out about potential failures and where it can or cannot resolve contracts at compile time there need to be some diagnostics. We can define three types of diagnostic that let the user know what needs to be done for resolving the contracts at compile time. They all depends on the compilers analysis skills, and may differ from compiler to compiler. The first two detects incorrect and insufficient contracts, the third reports on the compiler's inability to resolve the truth at compile time.

To support compile time resolution of contracts, All these diagnostics should be fixable without resorting to global analysis.

3.1 Known violation

This contract will fail if current function is called

```
void f0()
    post(false)  // Diagnostic: Will fail
{
}
```

Ill formed

```
void f1()
    pre(false)  // OK. No one has called it yet
{
}
```

OK, but using it is not:

```
void f2()
{
    f1();       // Diagnostic: Will fail
}
```

This is also ill formed

```
void f3()
{
    contract_assert(false)     // Diagnostic: Will fail
}
```

```
constexpr has_feature_x = false;
void f4()
    pre(has_feature_x)  // OK. No one has called it yet
{
}
void f5()
```

```
{
    f4();       // Diagnostic: Will fail
}
void f5()
    pre(has_feature_x)  // OK. No one has called it yet
{
    f4();       // OK. Will not be called
}
```

3.2 Unhandled conditions

There are valid input values to current function that trigger this contract.

```
int f0(int x)
    post(r: r > 5)  // Diagnostic: Invalid conditions exists
{
    return x;
}
```

3.3 Runtime conditions

There may be valid input values to current function that trigger this contract, but the compiler is unable to determine it at compile time. This is the fully static check warning.

```
void f0()
    post(secret_check())  // Diagnostic: Correctness unknown
{
}
```

3.4 Compilation modes

These diagnostics report problem in increasing strictness and can be seen as the static analysis mode. The first two of these diagnostics exposes incorrect code. Implementations should be allowed and encouraged to diagnose these errors and they should be part of the default mode. Since the ability to detect these issues is dependent on the compiler skill, it need to be possible to compile without as portability between compilers is not guaranteed.

The third is not always an error and may give different result on different optimization levels. Since Tuning down the analyser will generate more errors, it should standardized as an opt in analysis mode.

4 Implementation requirements

Static analysis can be done in many ways. Static analyzer tools, compilers, human reviewer. Here we mainly addresses compilers as one our main goals is to elide unnecessary contract checks. This means that we are bound to the c++ compilation model. The compiler will only see a limited number, or even a single function at a time, so each function must be independently checkable. It must also be given it's called functions needs and promises.

Preconditions, postconditions and invariants are all designed in a way that allows static checking according to the criteria above. The runtime checking part is only a recognition of the limits of compiler's symbolic evaluation abilities. For a human reviewer it was always supposed to be be compile time checkable.

4.1 Evaluation

Contracts on incoming data: A function's precondition and the postconditions of functions it call are compile time facts. Using theses fact in symbolic evaluation can prove contracts local to the function to be true. If these

cannot be relied on and must be checked, compile time resolution will not be possible.

Contracts on outgoing data: The function's postconditions, preconditions of functions that it calls and contract_asserts are the facts that needs to be verified, Normally by a runtime check but it should always be possible to turn them from runtime to compile time by changing **only** the local function or the incomming contracts;

Since we are bound by local reasoning, only the caller knows the context around the call and has a possibility to resolve preconditions in compile time.

Same goes for postconditions where only the callee side sees everything that is going on in the function and has a chance of being compile time resolved.

4.2 Propagation

To get any type of static checking to work, facts needs to be passed between caller and callee. This is the purpose of pre and post conditions. They defines what must be true on both sides of the call boundary. If a precondition is checked caller side, it must still be true when we have entered the function. A pre condition can be weaker on the callee side and still be true. Similarly, if a postcondition established but the implementation, the caller must be able to rely on it still being true when returned. A post condition can be weaker on the caller side and still be true.

5 Suggested polls

The first poll is if we should have standardized modes for compile time checking of contract.

> **Poll**: The compile time diagnostics behaviour of contracts should be standardized.

The second poll is if the suggested diagnostics is a good enough specification for compile time needs.

> **Poll**: The compile time enforcement should be one of "None", "Known violations", "Unhandled conditions", "Runtime conditions"

Last poll is about adopting compile time resolution as a design guideline for new (and existing) contract features

> **Poll**: All contract semantics should be designed to not prevent compile time resolution

6 Summary

Compile time resolution of contracts is an important property for safety and performance, and we should add the necessary mechanisms needed for it to be on par with the runtime semantics, and to commit to designing future contract features that allows that.

7 References

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