Inherited contracts

Document #: P3169R0
Date: 2024-03-21
Project: Programming Language C++
Audience: SG21 Contracts Working Group
Reply-to: Jonas Persson
<jonas.persson@iar.com>

Contents

1 Motivation

The contract MVP [P2900R5] specifies basic contract functionality but leaves virtual function contracts unspecified.
This proposal adds contracts for virtual functions. It does that in a mostly traditional “promise no less, require no more” way. This is a well established model that is easy to teach and reason about.
It also aim to take a minimal overhead path, where the evaluation of the contracts is placed where it introduces least overhead and most opportunity for optimization.

2 Goals

2.1 Checkable

We want contracts to be easily checkable by static checkers and by manual inspection. For this to be realistically possible in a complex system, we want as much as possible to be checkable in local context. Neither people nor static checkers (specialized such or compilers) are very good at global analysis. By requiring global analysis we effectively turn contracts into unpredictable runtime errors that cannot be handled.

2.2 Minimum overhead

A system designed with contracts can get a lot of small contract checks. We want it to be possible to keep these contracts in release builds. This means that they should be easy to inline and easy to access for the optimizer so it can remove redundant checks.

2.3 Optimization help

The hints given to the optimizer by contracts may give significant performance gains. For full effect all contract conditions must be visible to the optimizer without relying on link time optimizations.

2.4 No ABI change

Preferrably there should be no effect on ABI for enabled/disabled checks, for invoking checks or for propagating optimization hints.

2.5 Clear responsibilities

One of the great properties of classic pre- and postconditions is the clear division of responsibility. A broken postcondition is always the called function's problem and a broken precondition is always the caller's. With some of the relaxed views of the resent papers, preconditions turns into anyone but the called functions problems.

3 Design

The interface has "exactly what I say" semantics, implementation has "promise no less, require no more semantics".

3.1 Preconditions

When calling a function, virtual or not, the preconditions should describe what the caller needs to fulfill. No less, no more. If in enforce mode, the truth of that condition should be enforced. No less, No more.

3.2 Postconditions

When calling a function, virtual or not, the postconditions should describe what the caller can rely on after the call. If in enforce mode, the truth of that promise should be enforced.

3.3 Sequence of a virtual call

1. caller checks the preconditions on the called function's static type
2. the function implementation of the dynamic type is executed
3. callee checks postconditions for all overridden functions

Seen from a single function's perspective this means that it checks all its (normal) outgoing control flows.

From the the implementation's point of view the preconditions are then always the logical disjunction of all precondition sets of all methods it overrides. It is not checked at the callee side. The implementations cannot know through which base class the call comes, so it must be ready to handle any data that pass the precondition of at least one possible caller.

Since the implementation does not from where it is called it checks all postcondition sets from all overriden bases. This may sound excessive, but it is needed for full correctenss, and if each override has followed the rules and never promised less than its base. The optimizer can reduce the check to only the implementation's own postcondition.

The interface is always statically exact what was declared on the called override. Only the implementations follows the widen preconditions and narrow postconditions rule. It falls out correctly from the pre ORing and post ANDing.

Only one of set of preconditions is ever evaluated in a call, and if the compiler can verify that overriding postcondition sets are equal or narrower, only one overriding check needs to be evaluated on callee return.

3.4 Notation

Each function declares all of its pre and post conditions. It does not inherit any pre or postcondition when overriding functions. It always uses only the declared contract of the function of the static type.

3.4.0.1 Base class

```
struct C1 {
    virtual R1 f1(A1)
        pre(c1)
        post(c2)
    {
        contract_assert(c1);  // Will never fail
        return ...; // (c2) must be true
        // Post check (c2)
    }
};
auto g1(C1& c,A1& a) {
    // Pre check (c1)
    auto r = c.f1(a);
    contract_assert(c2);  // Will never fail
}
```

3.4.0.2 Single inheritace

```
struct C2 : C1 {
    R1 f1(A1) override
        pre (c3)
        post (c4)
    {
        contract_assert(c3 || c1);  // Will never fail
        return ...; // (c4 && c2) must be true
        // Post checks (c4 && c2)
    }
};

auto g1(C2& c,A1& a) {
    // Pre check (c3)
    auto r = c.f1(a);
    contract_assert(c4);  // Will never fail
    return r;
}
```

3.4.0.3 Multi inheritace

```
struct C3 {
    virtual R1 f1(A1)
        pre (c5)
        post (c6);
};

struct C4 : C2, C3 {
    virtual R1 f1(A1)
        pre (c7)
        post (c8);
    {
        contract_assert(c7 || c3 || c1 || c5);  // Never fail
        return ...; // (c8 && c4 && c2 && c6) must be true
        // Post checks (c8 && c4 && c2 && c6)
    }
};

auto g1(C4& c,A1& a) {
    // Pre check (c7)
    auto r = c.f1(a);
    contract_assert(c8);  // Will never fail
    return r;
}
```

Each contract stands on its own, and are not required to refine the ones it overrides. The reason is that, in the general case, it is hard to impossible for the compiler to verify that.
But in some cases it can and then we leave to the compiler to warn if it can detect it to be not true or false.

4 Discussions

This proposal has a strict contract enforcement on the caller side as opposed to widening and narrowing. Here are some elaborations on why this is the better choise.

4.1 Widening preconditions

Is is always safe to loosen the preconditions relative to the static check, it will allow us to serve more clients. But it will remove optimization opportunities. The strict caller interpretation allow for more static reasoning.

```
struct A {
  virtual void g() pre (c()) = 0;
  virtual void h() pre (c()) = 0;
};
// Where c() is a condition known by the compiler to not change by a call to g();
```

With strict preconditions

```cpp
void fun(A& a) {
  // c() evaluated
  a.g();
  // [[assume(c())]] // with assume sematics
  // c() check elided
  a.h();
}
```

With widening preconditions

```cpp
void fun(A& a) {
  // c() evaluated
  a.g();
  // c() evaluated
  a.h();
}
```

4.2 Narrowing preconditions

Allowing narrowing would prevent many possibilities for efficient static analysis.

```cpp
struct A {
  virtual void g() pre (c()) = 0;
};
// Where c() is a condition known by the compiler to not change by a call to g();
```

With strict preconditions

```cpp
void fun(A& a) pre(c()) {
  a.g(); // fully safe call

}
```

With narrowing preconditions

```cpp
void fun(A& a) pre(c()) {
  a.g(); // potential pre condition break.
         // Needs code review.
}
```

It is possible to break out the condition to a virtual function and get narrowing in practice without disturbing the static checker

```cpp
struct A {
  virtual void h() = 0
    pre (c())
    pre (h_pre());
  virtual bool h_pre() const = 0;
};
// Where c() is a condition known by the compiler to not change by a call to h();
```

With strict preconditions

```cpp
void fun(A& a) pre(h_pre()) {
  a.h(); // fully safe call
}
```

With narrowing preconditions

```cpp
void fun(A& a) pre(h_pre()) {
  a.h(); // potential pre condition break.
         // Needs code review.
}
```

In the case where the base class is not ours to modify, it is always possible to fall back to an assert.
which is then also good because it do not push the error to users of A, who has nothing to do with this.

```cpp
struct A {
  virtual void h() = 0
}

struct B : A {
  void h() override {
    contract_assert(c());
  }
};
```

4.3 Narrowing postconditions

Promise more for the return value of subclasses is always safe. It is on the dynamic type so we don't expect the static checker or optimizer to have much use of such narrowing.

4.4 Widening postconditions

If this is allowed no conclusion can be drawn about the nature of returned values.

```
struct X {
    bool d_pre() const;
    void d() pre(d_pre());
};
struct A {
  virtual X g() post (r: r.d_pre()) = 0;
};
```

With strict postconditions

```
void fun(A& a) {
  auto x = a.g();
  x.d(); // fully safe call
}
```

With widening postconditions

```
void fun(A& a) {
  auto x = a.g();
  x.d(); // potential pre condition break.
         // Needs code review.
}
```

Widening postconditions can thus be worked around the same way as a narrowing precondition with virtual functions so there is no reason to allow it.

4.5 Comparing to the P3097R0 checking model

This paper was initially a reaction on [**P3097R0?**], so I'll give here a view on the problems in its checking sequence that are avoided in this paper.

P3097R0:

1. caller checks the preconditions on the called functions static type
2. callee checks the preconditions on the implementation functions static type
3. the function implementation in the dynamic type is executed
4. callee checks the preconditions on the implementation functions static type
5. caller checks the postconditions on the called functions static type

P3169R0:

1. caller checks the preconditions on the called functions static type
2. the function implementation in the dynamic type is executed
3. callee checks postcondition for all overriden functions

4.5.1 Caller checks the preconditions

P3097R0:
This is the same in both proposals. But, since 2) is also checked and is allowd to be either narrowing or widening, 1) is neither nessessary nor sufficient.

It still sometimes makes violations happen in the right place and might in some cases contribute information to elide the callee check.

P3169R0:
This is the only check done for preconditions. It is minimal and sufficient.

4.5.2 Callee checks the preconditions

P3097R0:
The problem with 2 is that allows implicit narrowing. Because narrowing is always a possibility, no warnings can be issued on narrowing when that is not intended.

A reviewer will have a hard time spotting if the narrowing was intended or not and there is no way of grepping a codebase for suspect cases.

It dilutes the meaning of preconditions. Traditional DbC gives a clear responsibility in case of a precondtion violation. The caller has made some error in how it uses the interface.

But when using precondition to check for conditions belonging to other interfaces this is no longer valid. The error can originate from the use of any other interface implemented by the dynamic type. It can be in in a surrounding scope controlled by the same client, but it can also be in inaccessible code outside of the client control.

While these checks has its place, thay are not really precondtions and it would be better if they had a clearly different syntax.

P3169R0:
This is not checked. But since the preconditions are supposed to not narrow, a warning can be issued if the declared precondition does so anyway. Or in a more strict setting, if it cannot be proven not to. With strict here meaning attempting to resolve all contracts at compile time.

4.5.3 callee checks the preconditions

P3097R0:
Does not verify correctness for all callers only for the implementation.

It has all the same problems as the precondition where the missing "promise no less" semantics prevents warnings and makes reviewing hard.

When testing the postcondition for such a class it is not enough to feed it with all data permitted by the precondition. It must be done by using all interfaces that can dispatch to this implementation.

P3169R0:
Checks postcondition for all overridden functions. This gives a guarantee that wherever a call is coming from, any error is caught here.

Since the postconditions are supposed to not widen, a warning can be issued if the declared postconditions do so anyway. Or in a more strict setting, if it cannot be proved not to.

4.5.4 Caller checks the postconditions

P3097R0:
Because the callee side check is not total, this check can only be elided in cases where devirtualization can be achieved.

A violation here when source location is not propagated from the callee will make it hard to pinpoint the cause of the error. And a debug break on the violation will stop after the problematic code has gone out of scope.

P3169R0:
Nothing needs to be done here. the correctness of the postconditions is already established.

4.6 Repeating or inheriting contract declarations

What happens when a overriding function has no contract declarations but the overridden function has?

There are two possibilities. The first is that such function has no contract and contracts need to be repeated. This has the advantage that the contract for a function is always explicit and visible to the user of the function without inspecting base classes. The contract will also not change unexpectedly because of changes in a base class.
On the flip side it makes it hard for a reader to know if it is the same contract or not. Without the compiler enforcing correctness in this is just another source of errors. One forgotten precondition will remove all guarantees for the implementation.

The other possibility is that the function gets the same contracts as the parent.

If an overriding function has no explicit precondition it inherits the ones from its parent. If it has more than one parent where at least one has a precondition, a new precondition must be supplied.
If an overriding function has no explicit postcondition it inherits the ones from its parent. If it has more than one parent where at least one has a postcondition, a new postcondition must be supplied.

This proposal suggest inheriting parents contract, but works with any of these alternatives.
As there are pros and cons for each it would be good candidate for a separate poll.

4.7 The blame game

The contracts check all outgoing flows from a function, which mean that the fault is in this function or earlier in the flow. If all function do the same checks we know it is in this exact function. For this reason it is important that the source location a precondition report is always the calling function and for a postcondition the called function.

4.7.0.1 Contract propagation

When a function calls another function it needs to ensure the state needed to fulfill its preconditions. If it cannot do this locally it can push the responsibility up by adding a new precondition on itself. This technique of lifting the checks a step upstream moves error detection one step closer to the actual fault and it turns more checks into something that can be statically resolved.

This is a very useful bottom up design method to discover contract requirements.

For this to work strict and well behaved checks are needed.

4.7.0.2 Up the call graph

```cpp
struct A {
  virtual void g() pre (c()) = 0;
}
// Where c() is a condition known by the compiler to not change by a call to g();
```

```cpp
void fun(A& a) {
  a.g(); // #1
}
```

At #1 here it is obvious that the call to g() may fail. And there is not enough information to be sure it does not. We can fix this by reviewing every place calling fun and see if it can possible break c(). Or, we can propagate the contract one step up:

```
void fun(A& a) pre(c()) {
  a.g(); // #2
}
```

We are now safe here and has pushed the problem one step up. Unless we allow widening preconditions. Then we might now disallow calls that previously worked, introducing runtime aborts. But if we have narrowing preconditions #2 can still fail.

The same logic also goes for propagating pre/post into invariants or into better argument types.

4.7.0.3 Example

One example of how narrowing is necessary has floated around discussions. Here is how it is handled with this proposal and contract propagation.

```
struct Car {
    virtual void drive(int speed);
};

struct MotorCar : Car {
  private:
    bool isEngineRunning = false;
  public:
    void drive(int speed) pre(isEngineRunning) override;
    void start() post(isEngineRunning);
};


void driveMotorCar() {
    auto car = MotorCar{};
    car.start();
    useCar(car);
}

void useCar(Car & car) {
    car.drive(45);
}
```

This may seem like a good use of a narrowing precondition. Writing driveMotorCar is safe since we know how a motor car works and, by knowing how useCar is implemented, we can avoid triggering the precondition of MotorCar::drive.

But if you are given a new type of car it is useful if there is a way for you to check that it is ok to drive before you start.

```
struct Car {
    bool ok_to_drive() const { return true; }
    virtual void drive(int speed) pre(ok_to_drive());
};

struct MotorCar : Car {
  private:
    bool isEngineRunning = false;
  public:
    bool ok_to_drive() const override { return isEngineRunning; }
    void drive(int speed) pre(ok_to_drive()) override;
    void start() post(ok_to_drive());
};
```

This will efectively implement a narrowing precondition. And also, it will provie a mean to propagate the contract.

```
void useCar(Car & car) pre(car.ok_to_drive()) {
    car.drive(45);
}
```

Now we have made the requirement to call useCar clear and its caller no longer need to inspect its implementatation to knbow what is safe or not.

Moreover, since we have propagated the precondition in each step, never hiding behind a function call, it is reasonable to expect an optimizer to use this and remove all checks.

In summary: safer, faster and clearer.

4.8 Checking incoming flows

Incoming flows are controlled by the function's preconditions and the postconditions of the functions it calls.

It could be argued that checking the incoming flows is just as important, so why does this proposal suggest to only check outgoing flows?

Firstly, checks would get a source location at the entry point, which is not close to where the error happened, or we need to always create and propagate source location information through the call and return paths, which has a runtime cost and causes an ABI break when the first condition is added.

Secondly, incoming flows are always first in its block and can never be statically resolved unless function is inlineable or if LTO is used.

Thirdly, we cannot know from where a call comes and therefore need to OR check possible preconditions, making all calls get widening preconditions. And we want strict preconditions. With the current design on checking each precondition separately and calling the handler if it fails. Implementing OR is not straightforward. The alternative to only check the preconditions of its static type is incorrect as we have no enforced no narrowing. Checking it to ensure it is not narrowing can be of interest, but is not a precondition tests as it is not the responsibility of the caller.

4.9 ABI independence

Nothing in this design require changes to the abi. Every contracts is placed where it needs to be with regards to the source location it needs to report and to the contexts needed by static analysers or optimizers.

This contract model also achieves full modularity. It is possible to enable contracts on a single function and get the full capabilities of the contracts, including reporting the correct locations, without affecting any calling or called functions or ABIs.

5 Conclusion

We have a lot to loose from adopting an overly flexible model. The strict model presented here is safe and optimizer and static checking friendly and can serve as a minimal starting point for the MVP.

Narrowing preconditions and widening postconditions can easily be implemented using virtual condition evaluation functions, and if desired this can be hidden behind some syntactic sugar in a future iteration of contracts.

6 Acknowledgements

Bengt Gustafsson for the in depth review and suggestions.

7 References

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