Doc. no. P2698R0
Date: 2022-11-18
Project: Programming Language C++
Audience: EWG
Reply to: Bjarne Stroustrup (Bjarne@stroustrup.com)

Unconditional termination is a serious problem

Bjarne Stroustrup

P2521R2 entitled "Contract support — Working Paper" allows just two translation modes:

- ***No_eval***: compiler checks the validity of expressions in contract annotations, but the annotations have no effect on the generated binary. Functions appearing in the predicate are odr-used.
- ***Eval_and_abort*** : each contract annotation is checked at runtime. The check evaluates the corresponding predicate; if the result equals `false`, the program is stopped an error return value.

[The grammatical error is copied rather than corrected.]

This renders such contracts unusable for run-time checking in programs that are not allowed to unconditionally terminate. In particular, it renders libraries, such as the standard library, unusable in programs that are not allowed to unconditionally terminate if such contracts are used as **Eval_and_abort** in their implementation. Consequently, the use of such contracts must be opposed for a large number of libraries that might otherwise be useful in such programs.

Programs that may not unconditionally terminate are not uncommon, some are foundational, others are critical applications. In most cases, such code has strict requirements for reliability and performance – it is a kind of code that can benefit significantly from the use of contracts. Supporting such code was a major reason for my work on contracts pre-C++20 (e.g., P0542R5).

Just turning off run-time checking of contracts for such programs is not a general solution. We cannot be 100% sure that "the last bug" has been found and a major rationale for contracts is to move checking of invariants from "random code" to more formal and easily identified contracts. Separating contracts into those that we'd like to be run-time checked in production from those we'd like just for debugging and/or for static analysis is hard to scale.

In addition, there will always remain a residue of tests needed to offer some protection against wholly unanticipated software errors and hardware malfunctions. The question is whether tests are contracts or "ordinary code." Ada SPARK may be the most widely used contract system in critical applications; it uses exceptions to report run-time contract violations. Ada2002 has adopted that into the standard in the form or the **Assertion_error** exception. Having tests that can become run-time as contracts means that they are distinguished from "ordinary logic" and available to static analyzers.

A contract violation is best handled by a separate system (a different process, or better yet, a separate processor). However, there isn't always a second separate "system" to which we can delegate the handling of the “fatal” error, so we must somehow proceed. The Linux kernel is one such example. I have seen critical financial systems that are not allowed to terminate unconditionally because that might leak objects representing financial entities. Examples that I have heard of but not personally experienced tend to come from relatively small critical embedded systems, such as scuba equipment.

Could we delegate the management of such problems to termination handlers? No. That would compromise the necessary simplicity of the termination handlers and force termination handlers to distinguish between different kinds of terminations:

```
if (real_termination) {
        // … really terminate …
}
else if (initialize_and_restart) {
        // …
}
else if (report_and_terminate) {
        // …
}
else if (something_else) {
        // …
}
else {
        // … really terminate …
}
```

This looks bad, but I’m confident that some resulting messes will be worse. Termination handlers would have to deal with the messy issues that couldn’t be expressed by an inadequate contracts design.

The P2521R2 design has been characterized as “minimally viable.” It may be minimal, but it is not viable. It fails to address a class of important motivational use cases. I consider it “sub-minimal.” I don’t usually quote Linus Torvalds but have a look what he (in his usual polite way) says about unconditional termination in the Linux kernel: https://lkml.org/lkml/2022/9/19/1105 . The Linux kernel is not the only such case.

It has been said that the P2521R2 design is minimal and can be improved by later additions, such as adding a mechanism to address the concerns articulated here. I don’t think that is realistic. If P2521R2 was added to C++ the odds are that the motivation for further improvements will be less and that any proposal involving a fundamental extension would fail or be bogged down in controversies for years. This would prevent the use of the “minimal viable” contracts in key code bases. A mechanism for not terminating after a contract violation is part of any minimally acceptable contract design.

Alternatives

Give the system builder a choice and the programmer an opportunity to express it in code. Here is a simple design that I and others have used for code that needed to run under different rules of error handling (in the absence of contracts):

```
// error-handling alternatives:
enum class Error_action { ignore, throwing, terminating, logging };

constexpr Error_action default_Error_action = Error_action::throwing; // a default

enum class Error_code { range_error, length_error };              // individual errors

string error_code_name[] { "range error", "length error" };       // names of individual errors

template<Error_action action = default_Error_action, class C>
constexpr void expect(C cond, Error_code x)     // take "action"
                                                // if the expected condition "cond" doesn't hold
{
   if constexpr (action == Error_action::logging)
      if (!cond()) std::cerr << "expect() failure: " << int(x) << ' ' <<error_code_name[int(x)]<<'\n';
   if constexpr (action == Error_action::throwing)
      if (!cond()) throw x;
   if constexpr (action == Error_action::terminating)
      if (!cond()) terminate();
   // or no action
}
```

A use:

```
double& Vector::operator[](int i)         // not std::vector
{
        expect([i,this] { return 0<=i && i<size(); }, Error_code::range_error);
        return elem[i];
}
```

This is an example and an illustration, and obviously not a proposal for the standard.

One possible remediation of P2521R2 would seem to be to add a third alternative to the possible actions after a violation:

- ***Eval_and_throw***: each contract annotation is checked at runtime. The check evaluates the corresponding predicate; if the result equals `false`, an exception is thrown.

Obviously, this option would not be used by people rejecting all uses of exceptions.

Assuming that exceptions are usable for reporting contract violations, design questions will include

- *How is the exception to be thrown chosen?* In particular, is there a way to have different contracts throw different exceptions?
- *How do you set the action used for a program?* The uses I have seen rely on there being a default action in a program, but that that action can be changed in one place (e.g., from debug behavior to minimal run-time checking, to no checking) to address changing needs during development and deployment.

- *Should it be possible to overrule the default action for a specific contract?* That option is allowed for **expect()** where it was used for things like controlling the effect of range errors separately from other violations.

Please note that I am not re-opening the discussion about continuation of execution past the point of violation. C++ exceptions are not resuming. Nor am I suggesting that it should be possible to modify the method of violation handling at run time.

Noexcept shows the problem

A **noexcept** can be seen as a contract that no exception is thrown out of the **noexcept** function. More specifically, **noexcept** is a kind of postcondition: this function will return. A **noexcept** turns a **throw** into a termination. This has repeatedly led to crashes when people overenthusiastically sprinkled **noexcept** over code in hope of performance or simplicity but missed some possible exception throws. Obviously, people shouldn't overuse **noexcept** in this way in code that is not supposed to terminate unconditionally. However, people have caused crashes in programs that didn't use to crash.

Contracts are more complex than **noexcept** in that they depend on run-time properties if run-time evaluated and are potentially more plentiful than **noexcept**. This limits the use of contracts in **noexcept** functions or the use of **noexcept** functions in programs that may not unconditionally terminate. In addition, could introduce crashes in programs that traditionally used recovery strategies. Given the experience with **noexcept**, we must expect run-time checked contracts to become a source of crashes. Such crashes must be made avoidable to allow wide use of contracts.

I would like to

- *see contracts widely used*, including in the implementations of widely used libraries (to help static analysis, to catch problems at run time, and to separate assumptions from ordinary conditional actions).
- *be able to recover from serious errors that can be caught only at run time*, such as range errors, and recover from them at least to the point of recording that the error happened and restoring the system to a safe state.

In the discussions leading up to the C++20 contracts (withdrawn at the last minute), it was made clear that exceptions couldn't escape a **noexcept** function. I agree with that decision; it does not seem practical to have a special rule for exceptions thrown from contracts.

This complicates solutions to the classical problem of specifying (the now **noexcept**) **vector::operator[]** using a contract. In one case, we wrapped every use of **vector::operator[]** in access functions using **expect()** in access operators of more specialized contains to express preconditions, hoping that no time-travel optimizations would defeat our efforts.

Acknowledgements

Thanks to John Spicer for pointing out that SG21 needed a paper to be willing to discuss my objection that unconditional termination would be a serious problem.

Thanks to Ville Voutilainen for filling me in on some of the reasoning behind the P2521R2 proposal that I had not fully appreciated and to encourage me to phrase this note to try not to offend anyone.

Thanks to the participants in the work on the C++20 contracts proposal and in the discussions before and after that.