P3361: Class invariants and contract checking philosophy

P3361R3

Esa Pulkkinen
esa.pulkkinen@iki.fi

August 14, 2024

Contents

Revision History

Revision	Date	Author(s)	Description
v1.0	3.7.2024	Esa Pulkkinen	Initial version
v1.1	4.7.2024	Esa Pulkkinen	Add citation to N2440, and Linux man pages
v1.2	7.7.2024	Esa Pulkkinen	Remove example that didn't serve any purpose. Add more clear explanation of constrained types in category theory
v1.3	8.7.2024	Esa Pulkkinen	Require class invariants referring to operations with preconditions to propagate the precondition
v1.4	9.7.2024	Esa Pulkkinen	Expand on object serialization
v1.5	13.7.2024	Esa Pulkkinen	Typographical changes
P3361R0	18.7.2024	Esa Pulkkinen	Allocated number from C++ standardization. Expanded description of theoretical model.
P3361R1	23.7.2024	Esa Pulkkinen	Clarify fibers in equivalence classes. Describe in detail how the equalizer and coequalizer give rise to satisfaction of the equation.
P3361R2	7.8.2024	Esa Pulkkinen	Consider how to escalate preconditions to class invariants
P3361R3	14.8.2024	Esa Pulkkinen	Fix problems modelling C++ in category theory

1 Overview

This paper is in response to C++ standardization on contract checking [8], and is intended for discussion in ISO JTC1/SC22/WG21 and in particular SG21 for contracts.

Latest version of the document is available at https://esapulkkinen.github.io/ciflmath-library/C++/contracts.pdf

1.1 Key takeaways

- Contract checks are test cases mixed with the tested code.
- Contract violations from libraries that terminate the process are considered unacceptable.
- Contract violation handling in libraries should throw exceptions.

- Contract violation handling in libraries must be flexible to many different use cases.
- Contract checks are testing of private parts of code.
- Functions with wide contract can have preconditions.
- Wide contracts are preferred in public interfaces.
- Every non-trivial piece of software contains bugs.
- Terminating the process is not an error reporting mechanism in libraries.
- Defensive checks can be guaranteed to succeed only if input validation already exists.
- Defensive checks are considered as a form of input validation.
- `main` function should choose what the contract violation can do.
- White box testing of code containing private portions is considered misguided.
- Tests and production code should use same contract violation handler.
- Contract violations are not a reason to skip calling destructors.
- Writing test mocks for class invariants for abstract classes produces lots of boilerplate.
- Class invariants are constraints on the class data members.
- Class invariants are preconditions on destructors.
- Contructors and destructor do not need to lock mutex unless they manipulate global state.

2 Philosophy versus practice

The programming literature was, at least in the 90's, much concerned with a correct way that software should be written. Usually one would specify rules to ensure various useful properties, with expectation that if you adhere strictly with the informal rules, then the code would be of better quality. Often such rules would take the form of rejection of a particular programming language construct. What is called a "programming paradigm" here, means the set of such rules intended for particular domain. These are things that often arise as comments in code or design reviews.

Couple of examples:

- In procedural programming paradigm, The classic letter "Dijkstra: Goto considered harmful" [5] established that implementation code should not branch out of the function/procedure in unstructured way, in contrast with assembly language constructs which are full of unstructured control jumps. Instead, "while loops", "if statements", "break" statements, and all the other control structures we are familiar with, are used to support structured programming.

- In object oriented programming paradigm, there is an informal consensus that a function or method that contains "switch-case statement" is considered bad. The reason is that then one function would contain code that should really be placed in a method in many separate classes. So making a function with such switch-case is not object-oriented. So instead of a switch, a "class hierarchy" with a class for each branch of the switch is advocated. This is because once there are many functions that have a very similar switch on the same *stored* data, the functions themselves are no longer cohesive, and much complexity in the program would be distributed to such functions, so they become too complex. So the object-oriented solution is to organize the code in terms of objects, and their types, classes, rather than functions.
- In functional programming paradigm, it's acknowledged that "global variables" are considered bad. Instead it's considered better that data flows through the program, and *values* should be passed to functions that need them for processing. When function has no access to global variables, the restriction limits the dependencies that each function is allowed to access, and produces good "functional abstraction" where function's dependencies can be read from its type, which is considered its interface [2].
- Test cases can be considered as a paradigm, in a sense that tests are limited to describing requirements. If something is not a (code-level, technical) requirement, then testing for it in a test case is considered bad.
- State machines are a paradigm where program is organized in terms of states and state transitions, where state machine states describe current stable situation of the external world, and state transitions describe changes in environment. It would be considered bad to confuse repeatable operations on (stable) states and (changes due to) state transitions in a state machine model.

I consider each of these example a limited point of view. As a limited point of view, describing some code as written with some paradigm can explain to other programmers what the rules are, for a small component. However, problems occur when mixing the paradigms because people often evaluate quality of various software design approaches from a point of view of the rules of one of the paradigms, which means solutions that are good in another paradigm will not be appreciated.

However, any set of rules you can devise will not be valid once you take a larger point of view. The practice of programming has moved on, and we all must know and use multiple paradigms. Typically components at the different side of an interface choose different rules, and possibly different languages and paradigms. Then I must deal with the problem in how to get these components to work together.

Unfortunately, explaining things what you should not do in a piece of code cannot be usually read from the code. If you shouldn't use some language construct, then the code just won't have that, and you can't read the rule that you shouldn't use it from code written to follow that rule.

In this context there are three primary solutions. Code reviews, static type checking and testing. In theory, comments are also a solution, but the code and comments can be out of sync, which is less than perfect.

So how do contracts fit in this? I consider contracts to be a method for testing with constraints of the "testing paradigm" where the tests are packaged together with the production code. It's a kind of mixing of paradigms to place tests together with the implementation code. However, contracts can *express* the restrictions people expect in the code. Contracts are not about not using particular language constructs, but about expectations on data visible in interfaces.

However, given the background that people want to set rules on the programming paradigm, the distinctions in paradigms can cause misunderstanding when discussing the solutions from the different paradigm point of view. I try to use all paradigms here, but I'll also try to explain which paradigm is being described, to avoid confusion.

3 Environment

First, I state some assumptions that are considered true, just to describe the environment and assumptions where software development occurs:

- The system has physical component. That is, it interacts with machinery, physical environment and with real people in a manner where safety of the people is at stake, and must be guaranteed, and where it is not automatically satisfied because some other system isolates the software components from the physical environment. Such isolation is considered useful but not sufficient for handling the problem.
- When software is written, the production code does not contain test cases. The test cases are distinct from the code tested. The test environments where tests are executed are distinct from production environments. That said, it's possible to perform testing in the production environment. But the character of tests performed in the production environment is much different from the character of tests performed to validate that the code works.
- Unit testing is required. In particular, any code that has no unit tests should not be used. Unfortunately, in reality, I see code and libraries where proper testing was not performed. But I don't want to encourage that approach.
- Often third-party software is used for test libraries and test reporting.
- Automatic tests are considered better than manual tests. Reason is automated tests are repeatable, and therefore satisfy normal criteria for science, and the results are measurable. In contrast, manual tests often suffer from differences in environments, differences based on who executes them, ambiguities in defining the tests, and problems measuring or reporting the results.
- Test scope should always be chosen so that test execution time is minimized. There are nonetheless many kinds of tests which take different time for execution. All time must be accounted for, including time spent for writing production code, writing test cases, writing test drivers, writing documentation, waiting for builds to indicate test results, debugging, etc.

- Libraries are an important part of the ecosystem. However, they are typically more difficult than applications because libraries are often acquired from third parties, who are considered to be responsible for their correct operation. That said, libraries are occasionally produced, so facilities intended to ensure libraries can be well written are considered valuable.

- Application code is an important part of the ecosystem as well. But I make distinction between libraries, modules and applications. Application code is defined as code component that can control the 'main' function of a process. A library, in contrast, has API style interface. A non-library module is a typically component that can send or receive messages in its interface (messaging interface).

- Integration code is code that links software written in different languages, are run in different operating systems, or that are written by different organizations. Shell scripts are a typical example of code intended for integration.

In practice, I expect unit test execution time for one subsystem to be less than 1 second. This is because unit test execution is part of the build, will always be performed during the build, and is restricted to be implemented in the same programming language that is used for the code being tested. I expect 10..20 builds per day per developer in a local environment. In addition, I expect 10..20 builds per day in the automated build.

I expect "integration test" execution time for 2-5 subsystems to be 1-2 hours. This is because a person has to manually perform operations with the user interface of the product. This means whoever performs the test must build the product, start it, invoke operations that normal users would perform on it, and then manually check that the constraints from the tests are satisfied. Simulators can be sometimes used to restrict scope of the testing, so that it's not necessary to test the part of the system that are in responsibility of other organizations. I expect large systems to involve many independent organizations that all play their part.

I expect "acceptance tests" execution time for the system to be 1-2 weeks. This is because somebody must use "official build results" that are known to pass previous test steps for all components of the system. Then those must be executed using an environment that is similar to the production environment. Setting up a production environment takes time. It has both manual and automatic steps, but usually it involves setting up several machines that are installed from scratch. The software from multiple organizations are installed to the same environment, and the test validates that the whole system performs according to its original requirements.

4 Contracts in procedural paradigm

4.1 On defensive tests

None of this includes any specification for what is called "defensive tests". By definition, a defensive test checks for unlikely scenario which is assumed *currently* (where failing the test causes havoc), but which might be accidentally

broken later. I consider defensive tests to primarily a procedural construct. There is a distrust in solutions that claim some real-world situations cannot happen. If the code has a execution path that can lead to such situation, then it **can** happen. Perhaps it only occurs after somebody updates a version of the software to a version that is not compatible. Or the user inputs invalid data. Or an unauthorized user attempts to break the security of the product. But all of those situations can happen. From this perspective, any difference between "defensive tests" and "input validation" is not existing. In principle it can be different person who writes code for those checks, but such allocation can change more often than the resulting code. Defensive check failures are not at all less likely to occur than input validation failures, even after comprehensive testing. The question there is only scope. That is, defensive checks are input validation checks, but for smaller scope. When the larger scope performs proper input validation, it *can* mean some defensive checks never fail - while that input validation is performed. But there can in general be no guarantee that every user of a component performs such input validation. Worse, different users of the same component can perform different input validation. The connection between such input validation that makes defensive checks succeed and the defensive checks themselves is difficult and uncertain.

What this means is I typically want to link the input validation from larger scope with defensive checks from the smaller scope, in such way that defensive check failures become failures in the larger scope. This is in stark contrast with proposals to terminate the process if defensive checks fail! Arguably the input validation could have caught the failures beforehand if they were perfect. But I don't assume perfect forecast of uncertain events.

Systems can evolve by introducing new users for the existing components. Any assumptions that there is just a single user for a particular component are just wishful thinking. In particular, the unit test driver is always a user for a component, since unit testing is required. Unit testing environment is distinct from production environment, so code must be designed to work in both environments. Also "integration testing environment" and "acceptance testing environments" are distinct from the unit testing environment, similar considerations apply to those.

4.2 Prefer wide contracts

So I would propose that only public interfaces that contain the "wide contract" should exist. Interfaces that have the narrow contract should be declared private to a subsystem or library. This is a strong statement to say that undefined behaviour is considered unacceptable.

What do I mean with "private" here? Obviously, classes that have "private" data members or methods are private in this sense for the private portion of the class. Unnamed namespaces are also considered private to a source code file. Also in DLLs or libraries produced by modules that do not "export" some symbols, such non-exported operations are considered private. The problem with anything that is declared "private" is that testing them directly from unit test that has access to public interface only is usually not possible. This is to say "white box testing" of code containing private portions is considered misguided.

Tests therefore have access to the "interface" or "public methods" of the code only. Private part of a component cannot be independently tested, they are

only tested as part of the tests for public interface. There are some approaches for testing that attempt to skip the "public/private" distinction by overriding the private keyword with a macro for purpose of tests. This is not acceptable because the tested code in that case does not represent what occurs in the production environment, so test results would not produce useful data on what happens in production.

In practice, searching C++ code for "private" keyword is a good way of detecting when some component is not properly tested. More comprehensive methods exist, e.g. measuring test coverage. Similarly, searching for the "final" keyword can detect operations that are intended to be extended within the implementation of the operation itself (rather than separately in derived classes). I welcome introduction of additional such mechanisms where doing `grep` on the codebase can detect certain patterns of assumptions from the code.

But here's the catch: Adding contract checks is a useful way of testing such "private" portions of code. When you hit that code during testing that is invoked by unit tests, even when those can only invoke the public methods, failures in contract checks from internal operations means the private interface expectations were violated. Those checks cannot be written to unit tests that are outside the component that declares it private, since they cannot access the parts declared private, and therefore tests for those MUST be written as contract checks within the code.

4.3 Preconditions on functions with wide contract

Now, I still believe that preconditions on functions are possible and reasonable. This is in contrast to propositions that with wide contract, preconditions do not exist [7]. For example, `vector<T>::at(size_type i)` is an example that has wide contract with a precondition. It is still only allowed to call it when the argument i is within $0 \leq i < v.\textbf{size}()$, where v is the vector. What the wide contract is expressing in this case is that failures in preconditions are not allowed to produce undefined behavior, so I expect proper error handling at run-time (or before). In this case, this is accomplished using exceptions. There are multiple ways of handling errors that could have been used, e.g. `errno` style thread-local storage, return values indicating error codes, reference parameters indicating error code (as "out" parameters), marking the object as invalid, throwing exceptions. However, I require these mechanisms to be limited to things that a single library function or method at issue can decide. These do not, in particular, include error handling mechanisms that cause termination of the process, as that is considered as the decision of the `main` function. The point is, if `main` is running, I consider it unacceptable to terminate `main` except by decision performed within the source code of the `main` function itself. This means, throwing an exception is ok, because then `main`, or some other function with interest in handling the exception will catch that exception and performs logging, and decide to terminate or continue as required for the case.

Notice also that joining all concurrently running threads must be performed before `main` exits. Also on POSIX, `shutdown` operation on listening TCP sockets need to be performed before exiting the process [1]. I'm expanding on this problem later.

4.4 How noexcept interacts with contracts

The `noexcept` specifier has subtle interaction with contract based assertion. In particular, if a function is declared as noexcept, the runtime enforces that thrown exceptions cause termination of the process.

However, it's trivial for the programmer to ensure such termination never occurs. You simply add a `try { ... } catch (...) { /* error handling that doesn't throw */ }` around such function.

Unfortunately, this useful property is no longer true, if contracts are allowed to cause termination of the process on contract failure.

Also, this may be somewhat simplistic as the part `/* error handling that doesn't throw */` can be arbitrarily complex, and definitely implies that the interface of the function must explicitly declare mechanisms to indicate to caller what happened.

Even though I have strong objections to terminating the process in contract violation cases, there is still a point in `noexcept`. The noexcept specifier can be used to declare when the writer of the code has control over the 'main' function, and therefore can choose what contract violation handling mechanism is used - and in particular - has chosen it to be a call to `std::terminate`.

Notice that if contract violation causes exception, and the exception is propagated far enough that a `noexcept` function would exit with such exception, then the `noexcept` function declares that it is *known* that it's allowed to terminate the process if such exceptions occur. But translation of contract checks in libraries is not such situation. So I propose that contract checks in libraries should throw an exception, and once that exception is propagated and reaches the application code (that by definition has control over main) during stack unwinding, the application code should choose how the contract violation is processed.

Unfortunately, this limits strongly when `noexcept` can be used. Any libraries that do not have control over the contents of the `main` function should not attempt specify noexcept (or attempt to call `std::terminate`), because if the function nonetheless raises an exception, an unacceptable behaviour occurs. Of course, analysis of the code can sometimes prove that no such exception can occur, and those are still valid basis for declaring 'noexcept'.

4.5 Classification of undefined behaviour

Any execution of the program or its components that can cause one of the following behaviors (that are all part of undefined behaviour as defined in C++) are considered unacceptable:

- termination of a process while the production system is running
- termination of a thread without indication to other threads
- exhaustion of resources
- segmentation fault or a core dump of processes
- unauthorized or uncontrolled changes to file system or database
- unauthorized shut down of the machine or downtime of a production system

- corrupted or uninitialized data in messages sent to other processes
- unauthorized physical changes to execution environment caused by failure
- deadlock where all threads are waiting for each other
- livelock that spends CPU resources without producing output
- Nonterminating computation that doesn't produce output

It is not always possible to distinguish between these problems. The reason is, termination of process does not necessarily have sufficient safeguards, and restarting the process is not possible because repeated restarts will eventually cause exhaustion of resources, because it cannot be guaranteed that a unacceptably failing process does not spend resources, such as listening sockets. Exhaustion of those resources may cause that new processes to replace the failed one cannot be started at all. When resources are limited, this is unacceptable. When it's possible to distinguish some of those behaviours based on severity, that's useful, but cannot always be ensured.

It is not a solution to testing that small tests would be executed as separate processes. Starting new processes has non-trivial cost. Running 10000 tests in 10000 distinct processes is not feasible. But running those same tests in 1 process is definitely possible, and this makes it possible to run it several times per day.

But if one of the tests fails, it is **not** acceptable that the test causes any of the unacceptable behaviours.

Any "defensive" mechanism that attempts to escalate local problems to one of the above unacceptable problems is considered unacceptable. I consider such mechanisms as trying to exploit the high severity of these unacceptable behaviours to handle less severe problems. Even if possible, this is not to be accepted, because consequences of handling high severity problems are grave. In particular, less severe problems should have less severe solutions. Such escalation will prevent use of better error reporting mechanisms.

However, exceptions are still acceptable as an escalation mechanism. The reason is, it is possible to recover from exception locally. Then the cost of the problem can be limited. In production environment, ultimately, this means the problem is logged and the system goes on processing any further requests that do not cause such failures. In testing environments this means, the rest of the test cases are executed even if one of the tests fail due to contract violation. But I would normally enable defensive tests in same way in production environments and in testing environment to avoid difference in behaviour that is caused by changing compile-time flags to prevent validity of the executed tests.

So what's the difference between an assertion and if statement that used to guard throwing an exception? The difference is that the latter is part of implementation logic, whereas an assertion is or should be part of an interface. These are different. I could easily *implement* an assertion declared in an interface using the if statement that throws an exception if the condition fails. But the assertion is used to communicate requirements. Due to this, the approach where assertions are hidden in some black-box logic which is the only place where the condition can be checked seems unreasonably obscure. Those requirements should be visible every time you look at the interface of a component. At

minimum, the test cases should test for them. Of course, even inside the black-box logic, if you fail the assertion in some way, the problem becomes visible.

Normally, the cost of test failure should be the test is marked failed, the test execution is continued, and then somebody checks the log of the whole suite of tests. Based on the reports, analysis is performed as to where the problem is, and somebody fixes the problem.

If the test suite crashes, produces segmentation fault, causes the machine to reboot or does not properly report the test result, then no useful information is obtained from testing, and detecting where the problem occurred is much more difficult.

The debugger is normally not part of that process. Because it's much too difficult for determining where the problem is. I normally expect automated test suites. In particular, core dumps are often lost after crash (because the OS stores them to place that gets cleaned up). Also processes that crash do not typically produce any readable reports on where the problem might be.

This means "assert" macro from C is not useful in libraries.

The "defensive checking" mechanisms are considered test cases that are embedded with the production code, but not separated from the production code. That would be considered mixing test cases and production code. If such check fails in the production environment, there is no guarantee that unacceptable behaviors do not occur.

If the checks are disabled in the production environment in situation where the check would fail, there is no guarantee that unacceptable behaviours do not occur. Rather it would only make production environment different from the test environments, and cause discrepancy in test results, so you would need to execute the tests twice - once with runtime contract violation checking enabled and once with runtime contract violation checking disabled.

In either case, the defensive check is unnecessary, since production code would not enable the checks. OTOH, if I can never enable them, then why would they exist in the first place? If the checks are enabled in production code but cause termination of the process, there is no guarantee that unacceptable behaviours do not occur. It's impossible to determine if such checks always succeed. Except if such conditions never exist in code in the first place! When tests succeed, I get additional confidence that the test cases and the tested code agree. But this is not same as ensuring that bugs do not exist.

I consider that all non-trivial pieces of code have bugs in the sense that there can be future requirements that are currently unknown but which will prove the code doesn't work once the requirement is discovered. Most code never reaches the stage where even the stated and known requirements are always satisfied.

However, when exceptions are thrown due to what is called a"input validation", the exception handler usually can ensure that the system recovers. What this means is, it reports to actual end users that there is a problem in the system. And the system still continues to perform its job. Typically the first place where input validation failures are caught is the user interface. This recovery is not the naive approach for "exception retry" which attempts to continue from the point of throw if an exception occurs [14]. If that was reasonable approach, then why would the code throw an exception in the first place? If that was nonetheless reasonable, then coroutines should be used instead. Rather it attempts to ensure the system can continue after catching the exception even though a small component in it is claiming failure. That small component is not necessarily a

process, can be larger or smaller component than that. Such mechanism should exist.

4.6 Destructors should be called before terminating the process

An argument was raised that allowing process to continue on contract violation is unreasonable. I reject that argument on the ground that the scope is wrong. It's not the process that should be terminated in that situation. Rather if a function fails its contract, the execution of the function should not continue. But this is very different than terminating the process! So at each level of abstraction, the proper error handling mechanism should be used.

4.7 Assertion recovery in different scopes

So let's consider how each type of assertion should be recovered from:

- If an assertion placed on a *output value* from an expression fails, the value cannot be used as *input* for subsequent computations.
- If an assertion placed before a statement fails, the subsequent statements in the same function should not be executed (function should exit).
- If an assertion placed as precondition for a function fails, the function should not be executed (client must handle failure).
- If an assertion placed as postcondition for a function fails, the function cannot be successfully exited (failure must be indicated).
- If an assertion placed as precondition for entering a critical section fails, the critical section cannot be performed.
- If an assertion placed as postcondition for leaving a critical section fails, the changes made in the critical section must be undone before exiting the critical section.
- If an assertion placed as a class invariant fails, the object private data should be considered invalid, and object scope cannot be entered/left before class invariant on the private data is re-established (or object is destroyed).
- If an assertion placed on a entry point function of a thread fails, the thread should terminate with an error code sent to the the thread that created it.
- If an assertion placed on an interface in a library API fails, an exception should be produced to the caller.
- If an assertion placed on implementation of 'main' function fails, the process should exit with an error exit code.
- If an assertion placed on a test case fails, the test case should fail.

- If an assertion placed on a GUI component fails, the presentation in the GUI should be refreshed and retried.
- If an assertion placed on request message received a communication channel in a server fails, an error reply message should be produced, and server should continue.
- If an assertion placed on reply message sent from a communication channel from server to a client fails, the reply should be declared invalid in the client and retried after timeout.
- If an assertion placed on an operation that is called by precondition, class invariant or postcondition check fails, the corresponding failure should propagate to the outermost contract check and cause its failure.

It is definitely unacceptable that destructors whose execution is pending would not be executed before terminating the process.

5 Class invariants: Contracts in object oriented paradigm

Class invariants are by definition requirements that are applied to the state of the object, and that restrict what data stored in instances are considered instances of the particular class. Failing such invariant means that *some other* class should handle the instance.

5.1 Destructor preconditions are class invariants

I disagree with the proposition that destructors can have preconditions that would prevent destruction of the object in case of contract violation. I think destructors as such must always be invocable without regard for contract checks. However, it's the class invariant that enforces that requirement.

When class invariants exist, contract violation for pre and post conditions of the methods of the class must be distinguished from violation of the class invariant.

The destructor is by default 'noexcept'. So when destructor precondition check fails, often this would lead to terminate being called. This would effectively destroy any attempt to report contract failures for class invariants with exceptions. However, if I check the destructor precondition as postcondition of the constructor and as postcondition in every method, the failure would be diagnosed by throwing exceptions from those methods. Notice that it can only be that method where the problem is, since object's stored data in private data members can only be modified in methods of the class or its friends. Friends are assumed to behave nicely, so shouldn't break the class invariant. So the method postcondition should detect it. Now having the class invariant as postcondition in every method should guarantee that destructor precondition is always satisfied when destructor is being invoked. Therefore destructor doesn't *need* to check it, and destructor can always be called. However, if destructor would check it, I might detect cases where recovery from violation of the class invariant fails to restore the object state to valid state, and then crash the program when

destructor is called. So this would be useful information that recovery from postcondition check failure for class invariant was incorrectly implemented.

Nonetheless, violating the class invariant is a grave situation, since object's stored data no longer satisfies its requirements.

However, throwing exception from method postconditions seems better than throwing exceptions when the object is being destroyed, since destructors are often invoked from other destructors, anyway in situation where exceptions are not expected.

The destructor would typically depend on the class invariant, but should not depend on the method-specific contracts. There are many methods but only one destructor. So what must be done to ensure destructor can be called is to ensure that modifications to object state are not performed that would break the class invariant. Therefore, destructor precondition is the class invariant. And it makes sense to check that as postcondition of every method. After all, after method is called, it should be allowed to destroy the object.

Notice that here I am *only* talking about constraints on member variables. If a contract depends on global variables or class-scope static variables, or something that's not part of the object being destroyed by destructor, those are not considered part of class invariant. Class invariant is a constraint on what kind of objects can exist, and making that depend on unrelated global state is often bad design. Or at least it's not object-oriented design.

However, there are cases where code execution needs to be distinguished to particular phases of processing, e.g. initialization, operation, shutdown. This in particular happens in a "state machine paradigm". Some initialization needs to have occurred before any instances of a class can be created. So class invariants that depend on global state (or state machine state) may still be useful, to express situation where object instances can exist after initialization, but before shutdown. That can be used to block access to operations during initialization or during shutdown. So I would not advocate mechanisms that enforce that class invariants can only refer to class member variables, or things that are only accessible by class members via some other classes declaring access control via friend declarations.

Therefore, any preconditions on the destructor are also postconditions for the constructor, and methods should preserve such invariants to ensure that object can be destroyed when needed. Notice that "preserving" an invariant is much easier than "establishing" it. The constructor must establish the class invariant, the methods must preserve it, and the destructor is allowed to rely on it and break it while it destroys the object. For raw memory after destroying the object, only few requirements apply.

With concurrency, even preserving the invariant can be a problem, because concurrent modifications might modify object state in unexpected way. Sufficient mutex locks or atomics are a solution here. It's important to see that when the class invariant talks about "private" object state, the state is private because the object instance must be considered as indivisible. Concurrent modifications must still preserve that indivisibility, and consider changes to object state made by one non-const method to be indivisible.

In theory, if mutex locking or the indivisibility requirements are handled at larger granularity or framework, then you might be able to ignore that constraint in methods and in class invariant. But that's not common.

For move operations, the source of the move often gets partially destroyed.

This does not mean move operation can break the class invariant, destruction of the object is still required after the move has occurred.

Notice that if a method raises exception, typically it is not allowed to modify the object state, or at minimum state modifications must nonetheless satisfy the class invariant after the exception was thrown. This applies even if method contract violations occur, e.g. violations of postconditions. If you violate both a class invariant and a method contract, then you've violated two contracts, and skipped check for one of them.

5.2 Generalization from assertions to relational constraints

Assertions express conditions, similar to conditions in `if` statements. However, there are certain situations where I want to express more complicated relationships with assertions.

Suppose I had a language construct for declaring class invariants:

```
class InvertedX;
class X {
  const InvertedX &invert() const;
  class_invariant( this->invert().invert() == *this );
};
class InvertedX {
  const X &invert() const;
  class_invariant(this->invert().invert() == *this);
};
```

Problem of course is that within X, InvertedX is not yet defined, and neither is its invert method, so the invariant seems out of place.

Notice I've skipped definition of "operator==()" that the invariant requires.

5.3 Invariants on interfaces expressed as abstract base classes

Invariants should be possible also for interfaces. Here's an example, written in the way you'd ideally want to declare the invariants:

```
template <class Value, class Expression, class Type>
class Language {
public:
  virtual unique_ptr<Expression> parse(std::string input) const = 0;
  virtual unique_ptr<Type> parsetype(std::string input) const = 0;
  virtual unique_ptr<Value> evaluate(const Expression &e) const = 0;
  virtual bool typecheck(const Expression &e, const Type &t) const = 0;
  virtual bool valid(const Value &v) const = 0;
  virtual bool validtype(const Type &t) const = 0;

  class_invariant( validtype(*parsetype("integer")) );
  class_invariant( valid(*evaluate(*parse("1+1"))) );
  class_invariant( typecheck(*parse("1+1"), *parsetype("integer")) );
```

```
  bool interfaceInvariant() const;  // described soon
};
```

Now of course checking the class invariant seems very difficult here. When should the invariant conditions be evaluated? The compiler can't actually check those when I invoke the methods, since each invariant invokes several. It can't evaluate them in a constructor, since this class is abstract. It can't evaluate them in most derived class constructor, since compiler doesn't know which one is the most derived class. It could check it after a full instance has been constructed. But it would be very inefficient.

In such cases, the programmer can help, by declaring a member function or a static member function instead.

```
template <class Value, class Expression, class Type>
bool Language<Value,Expression,Type>::interfaceInvariant() const {
    return validtype(*parsetype("integer"))
        && valid(*evaluate(*parse("1+1")))
        && typecheck(*parse("1+1"), *parsetype("integer"));
}
```

Then you can use this as assertion `contract_assert( language.interfaceInvariant() );` in appropriate situations, or invoke it inside a test case with a test framework that can handle failures in different way.

However, there is an effect on the visibility of the conditions. The conditions are now hidden inside the `interfaceInvariant()` function. Of course declaring it inline makes the conditions visible in the header file. You could split this function so each requirement is separate, so maybe the name of the operation would describe the requirement sufficiently.

5.4 Propagation of preconditions to class invariants

For the uses of `unique_ptr<T>::operator*()` in the class invariant, using that operation brings its own requirements: The pointer cannot be null, at least the expected precondition on this operation would prevent the null case. If we would see such case where the operations would return null pointers, somehow this should mean that the class invariant is not satisfied. However, typically you would get undefined behaviour from the operation. So preconditions for the operations referred in class invariants might bring their own requirements that should be thought of in same way than the class invariant itself to cause the class invariant to fail. Somehow this seems a derived requirement though, and might be related to the specific situation described in the class invariant. I view this as escalation of the preconditions to larger scope.

On the other hand, thinking of each class invariant as independent requirement in the sense of testing for the requirement means that there should be a clearly limited scope for each class invariant. So you could intepret the use of `unique_ptr::operator*()` to indicate such limit. So if the operation returns null in such way that subsequent dereference will cause undefined behaviour, and there is a precondition defined for `operator*()`, say

```
template <class element_type>
typename add_lvalue_reference<element_type>::type
```

```
unique_ptr<element_type>::operator*()
  const noexcept(noexcept(*declval<pointer>()))
pre(get() != 0);
```

Then the effect of the precondition should be that the class invariant doesn't attempt to consider the situation where the precondition fails. There are two interpretations for how a precondition might impact an enclosing class invariant:

- The precondition limits the scope of the class invariant to successful precondition. The class invariant might succeed while the precondition fails, but before that can happen, precondition check prevents continuing, causing effects of the contract violation handler. The class invariant is not checked since processing doesn't reach full evaluation of that check. Errors are reported as failures of precondition for `unique_ptr<T>::operator*()`.
- The precondition can fail, but such failures propagate to produce a failure of the class invariant, avoiding undefined behaviour, but escalating the problem. Then the failure of the class invariant limits the instances of the class to cases that do not produce such precondition failures, as failures of postconditions generated from the class invariant.

Of course, being explicit in the class invariant can resolve this. However, this makes the class invariants more complicated.

```
class_invariant( nullptr != parsetype("integer") && validtype(*parsetype("integer")) );
```

Arguably this is better, despite being more verbose. However the examples where `operator*()` is used multiple times become quite complicated.

5.5 Generating mocks for the class invariant

This is not yet the full story here. Given the interface, what in reality the implementations for the interface do? They invent new types for the template arguments, and must implement the abstract methods.

The traditional approach is to instrument the real implementation to be full of `contract_assert` operations, then run the real implementation in a test case. And if any asserts fail, then there is a bug to be fixed.

However I can try something else. So I write a very complex piece of code:

```
struct ValueMock {
  bool is_required_to_be_two = false;
};
struct ExpressionMock {
  bool is_required_to_be_one_plus_one = false;
};
struct TypeMock
{
  bool is_required_to_be_integer = false;
};

class LanguageMock : public virtual Language<ValueMock, ExpressionMock, TypeMock>
```

```
{
public:
  virtual unique_ptr<ExpressionMock> parse(std::string input) const;
  virtual unique_ptr<TypeMock> parsetype(std::string input) const;
  virtual unique_ptr<ValueMock> evaluate(const ExpressionMock &e) const;
  virtual bool typecheck(const ExpressionMock &e, const TypeMock &e) const;
  virtual bool valid(const ValueMock &v) const;
  virtual bool validtype(const TypeMock &t) const;
};
```

Now I can implement these to produce the mock (this implementation has not been tested, and is only conceptual):

```
unique_ptr<ExpressionMock> LanguageMock::parse(std::string input) const
{
   if (input == "1+1") {
     return std::unique_ptr<ExpressionMock>(
         new ExpressionMock{ .is_required_to_be_one_plus_one = true });
   } else {
     return std::unique_ptr<ExpressionMock>(new ExpressionMock);
   }
}

unique_ptr<TypeMock> LanguageMock::parsetype(std::string input) const
{
  if (input == "integer") {
    return {new TypeMock{ .is_required_to_be_integer = true } };
  } else {
    return {new TypeMock};
  }
}

unique_ptr<ValueMock> LanguageMock::evaluate(const ExpressionMock &e) const {
   if (e.is_required_to_be_one_plus_one) {
       return {new ValueMock{ .is_required_to_be_two = true }};
   } else {
       return {new ValueMock{}};
   }
}
```

You can see where this is going to. Using the class invariants defined in the interface as guide, each of the mocks keep track whether the class invariant conditions to be checked should be true at the particular phase of the invocations, and encodes it to the mock as data, so that the next method can check that data, and validate further conditions, and then return another mock which encodes data as to whether the invariants need to be checked. What I then want to do is to check, the the implementations of the 'valid' operations, whether the class invariants are valid:

```
bool LanguageMock::valid(const ValueMock &v) {
  return v.is_required_to_be_two;
}
```

Of course, definition of when the value is valid must be that the class invariants are satisfied for an instance of our value, so I just return 'true' when all the class invariants are true.

Now this valid operation returns true exactly when the class invariant:

```
class_invariant(valid(*evaluate(*parse("1+1"))));
```

should be true, since to get the boolean in the mock to be true, the whole process indicated by these evaluations must have occurred, and the input to `parse` operation must have been the input in the invariant.

However, these are just mocks, they don't invoke any real code to be tested. So what these mocks amount to is ability to check that *when* somebody invokes the interface with the string argument `"1+1"`, and invokes the `parse` function, then unique pointer's `operator*`, then evaluate, then again unique pointer's `operator*`, then `valid` operation, that the functions were actually executed in that order where the class invariant is relevant.

Now I actually realize at this point, that I left out `std::unique_ptr` from the template arguments. I can't write mock for that since it's in the standard library, and the interface doesn't abstract over it. So it's possible to do this for unique pointer by also abstracting over it. But here, the standard library is considered already tested, so I'm not testing its functionality. However that might be a mistake! But it's outside the scope of this paper to consider that case.

Here's the point of all these complex mocks. Once I have such mock which can determine when to check, I can now use this machinery to test the real implementation. So suppose I have a real derived class of `Language<Value, Expression, Type>` and associated concrete classes `Value`, `Expression` and `Type`.

I can add an instance of these types to each of the mocks. But I should do it in way that allows customizing the mocks with these types. So I'll add a template parameter to each of the mocks:

```
template <class RealValue> struct ValueMock
{
  bool is_required_to_be_two = false;
  RealValue _real_value;
};
// ... similarly for the two other mocks ...
```

So now I can instantiate:

```
template <class V, class E, class T>
class LanguageMock
  : public virtual Language<ValueMock<V>,
                    ExpressionMock<E>,
                    TypeMock<T> >,
{
  // ...
private:
   std::unique_ptr<Language<V,E,T> > _real_impl;
};
```

Now, the implementation of the operations are exactly like previously, but now I must also invoke the real implementation. For example:

```
template <class V, class E, class T>
unique_ptr<ExpressionMock<E> >
LanguageMock<V,E,T>::parse(std::string input) const
{
   std::unique_ptr<E> real_expr = _real_impl->parse(input);
   if (input == "1+1") {
     return std::unique_ptr<ExpressionMock<E> >(
         new ExpressionMock{ real_expr, /* .is_required_to_be_one_plus_one = */ true });
   } else {
     return std::unique_ptr<ExpressionMock<E> >(new ExpressionMock<E>(real_expr));
   }
}
```

This starts to look like something you would write to a test case.

So I wrap the real implementation with the mock, and collect enough information to know when the class invariant should be true.

Of course the `valid` operation must now check that the real implementation does the correct thing:

```
template <class V,class E, class T>
bool LanguageMock<V,E,T>::valid(ValueMock<V> e)
{
  bool value = _real_impl->valid(e._real_value);
  contract_assert(not e.is_required_to_be_two || value);
  return value;
}
```

Here this `contract_assert` matches the class invariant that was described in the interface.

Once I have done all these mocks, and once I invoke the test by giving the concrete classes there:

```
  LanguageMock<Value, Expression, Type> language;
  contract_assert(language.interfaceInvariant());
```

the test should succeed exactly when the implementation satisfies the invariant.

Of course, it's required here that each method on the mocks update the booleans correctly.

Of course, doing all this boilerplate in the mock implementations is annoying, and it's long enough that Tony tables are not going to be useful. Also wrapping the code in a mock is not always possible, and in particular this case cannot be done if the code doesn't use abstract interfaces, or where the interface has dependencies to implementation that is not replaceable with a mock.

However, when considering contract support for C++, ideally class invariants declared in interface would already handle all this. And somehow magically we would know when to check for the class invariant. That would be useful, but might be a big project.

So here is the challenge: How can these mocks be written automatically, based on declaration of the class invariant defined on an interface?

I don't have very good solution, this stuff gets written in manually written test cases.

5.6 Class invariants

An important part of a contract checking mechanism in object oriented paradigm is class invariants. What I mean by class invariant is a constraint on the private state stored in an object. [13]

The class invariants for private state are very different than the previously described class invariants for interfaces.

Constructors must fail, if they cannot establish the class invariant.

The execution of public methods can assume the invariant is valid after entering the critical section of the object, but must also preserve the invariant before exiting the critical section. Notice that postcondition violations for method contracts must be handled by not modifying the object from its original state, or at minimum, without breaking the class invariant. So methods that can fail to re-establish class invariant at exit from the method must throw an exception before modifying any data in the object itself, thus leaving the object in a valid state. This is definitely non-trivial for postconditions when the postcondition fails. So method postcondition and class invariant are different. When a method postcondition fails, the code must exit the method with failure. However that failure cannot break the class invariant, as that would make the object undestructable, so subsequent operations during the propagation of the exception that might want to destroy the object that failed, would fail.

Destructor can assume the class invariant as a pre-condition. Destructor should not have other preconditions. A destructor is guaranteed to leave the object in a destructed state (raw memory), such that associated memory can be freed.

If a destructor throws, it's a special case. It would mean the destructor wasn't able to turn the object into raw memory. I don't expect that to be a common case. However, there is an exception. If I have an "inverse" RAII wrapper which intends to undo behaviour of a previous RAII wrapper (e.g. unlock a mutex), to produce gaps in the RAII behaviour, then a destructor of such inverse RAII wrapper must often allocate resources, which could fail.

Notice that updates to private state should establish a critical section if concurrent modifications are possible. I conjecture that contract violations about the private data stored in objects should be checked within such critical sections. Typically this involves storing a mutex in the objects, and establishing a critical section for any updates and reads for the private data. Based on this idea, it might be useful to consider how `lock_guard` and `mutex` could be enhanced to establish class invariants, or an alternative `class_invariant_guard` should be introduced. This is outside the scope of this paper though.

Notice constructors are not normally written to lock a mutex since there typically cannot be multiple threads executing during execution of the constructor, since access to the data of the object is restricted during construction. Notice this changes if the constructor registers the object to a global repository that could be concurrently accessed. In such case, locking must be performed appropriately to prevent concurrent access during construction.

The contract checks for the class invariant can be associated with entering and leaving the critical section.

5.7 Contracts for global variables

Outside the functional paradigm, contracts should also be available for data stored in global variables. This is in particular the case when such global variables are used as interface, for example with the singleton pattern.

The thing that should be seen about (public) global variables is that they act as an "endpoint" where data can be stored. So usually useful contracts are used to link more than one such endpoint together to describe functionality.

```
std::atomic<double> bank_account;
std::atomic<double> wallet;
[[ invariant: wallet.exchange(bank_account.exchange(wallet.exchange(0.0))) == 0.0 ]];
```

However, similarly to the interface contract, it's difficult to determine when the contract should be checked when more than one global variable is involved in the constraint condition.

Here's another example:

```
std::atomic<double> x;
std::atomic<double> y;
[[ invariant: x*x + y*y == 10*10 ]];
```

(I'm ignoring the problems with comparing double for equality).

Obviously the meaning is clear that 'x' and 'y' should be the coordinates of a point on a radius 10 circle. But given that these constraints are too global, implementation seems difficult.

Placing similar things in more restricted scope may be useful:

```
struct point_on_circle {
  std::atomic<double> x;
  std::atomic<double> y;
  class_invariant( x*x + y*y == 10*10 );
};
```

Now this can already be expressed in C++:

```
struct point_on_circle {

  point_on_circle(double x_, double y_) : x(x_), y(y_)
  { if (x_*x_+y_*y_ != 10*10) { throw range_error(); } }
  double get_x() const { return x.load(); }
  double get_y() const { return y.load(); }
private:
  std::atomic<double> x;
  std::atomic<double> y;
};
```

If the variable 'x' and 'y' would be public, assignments to them individually might break the invariant. So the variables should be declared private to enforce the invariant. The const methods here can't break the invariant. However from outside an instance, the call such as

```
point_on_circle p(10.0,0.0);
// ...
contract_assert(p.get_x()*p.get_x()+p.get_y()*p.get_y() == 10*10);
```

may still fail if concurrent modification of the fields could happen.

6 Contracts in component integration

6.1 Specific concerns about contracts in libraries

What should be done if a contract check fails in a library? The library itself doesn't know. In particular, when compiling the library, it's not possible to know what contract violation handler should be used in the application that uses the library. Different applications that choose to use same library use different answers. Normally libraries handle all error cases by throwing exceptions. This is because the library doesn't know what error handling approach the application wants, e.g. does it want to exit the process, or continue after recovery.

The argument has been made that contract violations are so serious problems that exiting the process is always the correct recovery action. I respectfully disagree. If you have control of the `main` function of the process, then you can do that choice. But if not, then it's not acceptable to terminate the process. Consider, for example what would happen if the web browsers allowed their GUI libraries to terminate the process on contract violations when the operating system support for GUI doesn't provide sufficient color model for the user interface to look good - which would be typical thing that a GUI library might use contract violation for - as all GUI layout failures would be considered violations of the GUI's contract for producing good looking user interfaces, and therefore such things would be considered as bugs in that scope. In larger scope however, this is not sufficient reason to terminate the browser. As consequence, instead, browsers should support multiple color models.

The policy to terminate the process on contract violations would force a process division to every place where errors must be always recovered from. Even if for some use cases, processes can be restarted when errors occur, this assumption is definitely not true for all cases. I suppose the policy to terminate the process is motivated by assumption that when contract violations occur, the operating system is in best position to clean up resources allocated by the failing function. This is definitely not true. That is why c++ has destructors. If it was sufficient to clean up always using operating system facilities, it would be possible to remove destructors from the language, and handle all out-of-memory (and other error) situations by restarting the process. Note Linux kernel actually does that for out-of-memory situations, if configured to use "overcommit" feature of the virtual memory. When overcommit is disabled, this doesn't happen. In the context of C++, it's often necessary to support both cases, and cannot rely on such behaviour from the operating system. However, this choice by Linux kernel actually feels like it was motivated by failures in memory management in

applications, to avoid the impression that software typically fails due to lack of memory. So I would suggest disabling that feature and considering how error handling for out of memory situations can be properly handled in its absence.

Also, cleaning up TCP sockets must occur before termination of the process, by calling `shutdown` function of the POSIX interface. If this is not properly done, the operating system imposes an additional timeout before the listening ports can be reused, which is typically too long to allow the next instance of the process after retry to setup listening sockets. This means that in a server software that handles recovery, any library that checks contracts and terminates the process when contracts fails cannot be used. Because those contract violations would block TCP connections for too long time, which is too much for a heavily used server.

The whole approach of using `terminate()`, `exit()` and `atexit` handlers to terminate the process seems to be good approach when the infrastructure has only components written in C programming language.

However, C++ frameworks seem to fail in spectacular way when a infrastructure written in C++ is combined with libraries written in C. One such failure scenario is case where C code exits the process with `exit()`, with expectation that all cleanup occurs in `atexit()` registered handlers. While useful assumption in code exclusively intended for C language code, this approach then causes havoc when you must ensure that all destructors for C++ code have been properly executed before exiting the process. Usually the order of execution of the C cleanup occurs in different place compared to cleanup expected in C++ code. What you see when this failure scenario occurs problems with closing listening sockets, dynamically loaded shared libraries, shared memory, or other kernel-maintained resources. I think this was partially solved by the `at_quick_exit()` functions added relatively recently [4].

I view the insistence to terminate the process with `terminate()` to be an instance of the same problem, assumptions that are only valid in C code and designs based on limitations there, but which are not good when writing C++. I think it should be considered if it would be reasonable, in C++, for calls to `exit()` to throw an exception, which would propagate to `main`, exit main and then cause the real C language exit operation to be invoked? I mean require destructors to be invoked properly when `exit()` is invoked, all threads to be joined, and all listening sockets to be shutdown before exit from main?

Therefore, in libraries, contract failures should not cause termination of the process, unless the destructors in all threads are also properly executed!

7 Theory on preconditions and postconditions

Preconditions, postconditions, assertions and class invariants are normally all considered to be boolean tests. However there are several things that, purely on theoretical basis, make these checks different from normal code.

In theory, the preconditions and postconditions are considered to be part of *interface* of functions. Similarly, class invariants are considered to be part of *serialization interface* of classes [6]. If you serialize an object that fails its class invariant, you would be sending serialized data to another process that contains corrupted data. So the class invariants should be accounted for when serializing and deserializing data, for example in the iostreams library. Could contract

violation handler be specified in a custom `locale` facet, which would allow specifying how contract violations should be processed in context of iostreams and related frameworks? Perhaps this might be a mechanism for linking testing frameworks and other libraries that have interest in contract checking to the contract processing? It's not obvious this is the correct mechanism, linking it to locale might still cause unnecessary complexity.

Assertions within functions are also part of the interface, but indirectly. You could consider an in-function assertion to be an *interface* for each statement, available for use by subsequent sequential statements.

What this means is that `contract_assert` placed on an empty statement (or between statements) should interact strongly with compiler data flow analysis, since that analysis must also link together adjacent statements and try to determine overall behaviour intended by the programmer. It seems reasonable that the condition is available as additional information in the data flow analysis and optimization passes within the compiler, and to help with inlining. However, it's not obvious that the boolean conditions are the proper tool or additional information that programmers should be indicating to the compiler to control or fine-tune how compiler data flow analysis should occur.

The semantics of multiple assertions on the same statements requires some theoretical thought. If there are two assertions with the same condition, it should be possible to combine them. This is discussed in detail below.

To describe semantics in detail I'm modelling C++ extended with assertions with category theory. The full description of the category theoretical model includes a category whose objects are C++ types and whose morphisms are C++ functions. I'm only giving compact description of it.

The following description is based on the "Elementary Theory of the Category of Sets" (ETCS) model of set theory [12].

The category of C++ types and C++ functions has terminal objects, binary products, finite products, equalizers, pullbacks, initial object, zero morphisms, coproducts, quotient objects, universal and existential quantification, coequalizers and pushouts, and therefore finite limits and finite colimits. Terminal object is a singleton C++ object instance. Binary products are given by `std::pair`. Finite products are given by `std::tuple`. Monomorphisms and equalizers are given by invariants on variables. Pullbacks are given by invariants on `std::pair`, but also by `std::map` and `std::multimap` when the first field can be ordered. In addition, there is a generalization of the pullback to multiple fields of a tuple, that implements invariants on `std::tuple`. Initial object describes (type of) undefined behaviour, and various zero morphisms exists that correspond to functions that produce undefined behaviour. Coproducts are given by `std::variant`. Coequalizers are given by class invariants. Pushouts describe how class invariants interact with `std::variant`. Universal quantification (of type variables) is given by template parameters. Existential quantification is given by (pointers to) objects with private and protected access control, as well as virtual functions.

C++ classes are modelled as special C++ types, which would be a combination of tuples and existential quantification. Here no distinction is made between a `struct` with only public data fields and `tuple`, even though of course syntactic difference exists for how fields are accessed. Category theory describes semantic distinctions clearly, but unifies distinct syntactic constructs for the same semantic notion when those produce isomorphic results. Objects are modelled

as special "history" morphisms from the terminal object that construct the instance and transition its state to current state. Since "past computation" doesn't need to be re-done, such history morphisms do not have much structure apart from the data of the object instance stored in memory. Member functions would be modelled as functions whose first parameter is the 'this' instance, as an action of the member function on the history morphisms. Data members would be modelled by special projection morphisms from the type describing the class to the type of the data member where composing a projection with a history morphism produces a history morphism. Private and protected access control would be modelled by existential quantification. Inheritance would be modelled by special "derived-to-base" conversion functions and its inverse which models `dynamic_cast`. Abstract classes would be modelled by existential quantification. Virtual functions would be modelled by substitution of concrete functions to existential slots (of the virtual function table).

C++ templates would be modelled by covariant endofunctors (and universal quantification), as well as natural transformations for function templates. Template instantiations, specializations and partial specializations are modelled by substitution of types for universally quantified type variables, that is, substitution of types for template arguments. Templates extend the category of C++ types and C++ functions into a 2-category.

Unconditional control jumps would be modelled as morphisms to initial object, where executing the unreachable code after the jump is considered to produce undefined behaviour.

The terminal object represents a run-time environment that is empty except for an (unnamed) singleton instance. This singleton instance supports for example, `sizeof`, `typeid` and `decltype` operations, because they can be used for expressions in all types. The terminal object has properties that make it useful for describing primitives that are available in all C++ types.

C++ types are described as objects in the category of C++ types and C++ functions. Category theory has a notion of object, which is different than C++ objects, so I try to distinguish "C++ object" and "type", where "type" or "C++ type" means the object in the category of C++ types and C++ functions, and "C++ object" or a run-time instance of type X is modelled as an element, that is a C++ function $o : 1 \to X$, where 1 is the terminal object. I'm still going to use notation $o : X$ for that. This is different than (set theory) notion of subset membership, which would be shown as $x \in e$, where e is an injective function.

I don't use constructs where I would need to model time, apart from the previously described history morphisms. Time could be modelled as special object, say T, such that an instance that changes in time might be modelled as morphism that takes time coordinate $t : T$ as input, say $f : T \to X$. This could be used to model time-dependent operations and stateful changes. However, this complicates the theoretical model to some extent, and I try to keep it simple, so I'm not describing state in the model. This restriction amounts to treating object instances as `const`, and not modelling pointers, references or iterators in detail. Possibly pointers to instances of type X could be modelled as pointed types, $1/X$. However there are many other possible models of time, say, methods performing stateful operations could be modelled as monoid actions, i.e. special endomorphisms on the global state, or as a monad of I/O actions, or as special homomorphisms from a type describing time coordinate. When I avoid describing time-dependent operations in detail, I don't need to choose the

model used. That said, C++ does have stateful operations, and the model still needs to acknowledge that. So I just assume that all functions could in principle perform stateful operations.

The theoretical basis for assertions in a category theory context is "monomorphisms", "equalizers" and "pullbacks". These express, using the Curry-Howard correspondence (a.k.a. propositions-as-types correspondence), "constraints on types", "equations" and "constrained tuples".

A simple way to explain these concepts is as follows: A monomorphism, a.k.a. injective function, is a way to model subsets. A constrained C++ type models variable declarations with a constraint, and acts like a restricted subset of objects: $\{o|p(o)\} \subseteq X$, where p is a (run-time) predicate, that is, a function or method returning a boolean, and o is a C++ object. X is the type of the variable, which is seen as the collection of possible values for the variable.

A type equation is modelled with equalizer, that is a type constraint of the form $E = \{o|f(o) = g(o)\} \subseteq X$, where f and g are functions whose codomain supports equality comparison. An equalizer for $f : X \to Y$ and $g : X \to Y$ is the smallest injective $e : E \to X$ such that $o : E \Leftrightarrow o \in e \Leftrightarrow f(o) = g(o)$. Thus for $o : X$, we can say that $o : E \Leftrightarrow f(o) = g(o) \Leftrightarrow \exists(z : E)o = e \circ z$. This means that membership in E for $o : X$ requires that the equation $f(o) = g(o)$ is true, and the injective function $e : E \to X$ is the proof of that membership, where $E \subseteq X$ is the subset of X that satisfies the equation.

A constrained tuple is modelled as a pullback, that is, a type constraint of the form $E = X_0 \times_Y ... \times_Y X_i = \{\textbf{make\_tuple}(a, b, ...)|f_0(a) = f_1(b) = ...\} \subseteq X_0 \times ... \times X_i$, where $f_i : X_i \to Y$. Conceptually a pullback means a table of data whose fields satisfy equations. This would be complicated, except that the concept is already implemented for `std::pair` (i.e. where $i \in \{0, 1\}$) by `std::map` and `std::multimap`, and more generally by associations in UML class diagrams. Anyway, the constrained tuple is a generalization to more than two fields, and that makes it more like a database table than a map. Anyway, the concept is well-known and commonly used everywhere.

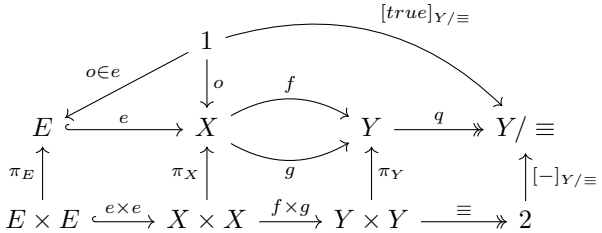
The category theory model of these concepts has a twist where each of these concepts have a dual, which is similar, but applies to variants, not to tuples, and where operations change direction. The dual of a "monomorphism" is an "epimorphism", the dual of "equalizer" is a "coequalizer" and a dual of "pullback" is a "pushout" [12]. Those produce operations on equivalence classes. Equivalence classes and C++ classes have some similarities, which could be utilized for additional features for C++. In particular, an equivalence class of *values* looks like how a C++ class is connected to its *object instances*. An epimorphism $p : X \to Y$, or surjective function, in this context describes partitioning "all instances" of the domain type X to "equivalence classes" described by the codomain type Y. In C++ context X would describe all object instances and Y would describe all classes whose instances are described by objects of type X.

An equivalence class in C++ classes would mean the access control difference between "class internal view" and "class external view", where internally to a class, the class has access to private members of the class and protected members of a base classes whereas from external point of view, the private and protected members are abstracted out as inaccessible. To be a real class invariant, the class invariant links only private and protected members, since then the abstraction created by the class ensures that the constraints can be modified. But this is a

design issue and it's definitely possible to consider cases where class invariants are used to connect external resources with class internal member variables. However, constraints in object oriented design typically only limit private and protected members.

A "coequalizer" in this context describes a class with class invariant, and in particular how the class invariant expressed as equation impacts the instances. Since only some of the instances satisfy the equation, the class is split to two, defined by the class of those instances that do satisfy the class invariant, and the class of those instances that do not satisfy it. So an equation of a class invariant in a C++ class is a partition of the class to two parts. A coequalizer $q : Y \to Y/\equiv$ for an equivalence relation $f(o) \equiv g(o)$ is defined by two functions $f : X \to Y$ and $g : X \to Y$ to be a class which unifies those instances $o \in X$ that satisfy the equation $f(o) = g(o)$.

I want the coequalizer to distinguish the erroneous cases from success cases, e.g. those cases where the class invariant produces different results, so that success cases are mapped to membership in the class with the class invariant, and failure cases are mapped to the class with inverse class invariant. So if we have an equation `class_invariant(x*x + y*y == 10*10)` as a class invariant, I choose the equivalence relation to be $f(o) \equiv g(o) \Leftrightarrow o.x{*}o.x{+}o.y{*}o.y == 10{*}10$ by setting $f(o) = o.x * o.x + o.y * o.y$ and $g(o) = 10 * 10$ where $o.x$ and $o.y$ are the fields of the object o. Then we compute the coequalizer $q : Y \to Y/\equiv$ and compose with either of the f or g. The coequalizer is the smallest epimorphic q such that it guarantees that $q \circ f = q \circ g : X \to Y/\equiv$, and thus the equation will be true for $o \in X$ iff $(q \circ f)(o) = [true]_{Y/\equiv}$, because construction of q guarantees that in that situation $f(o) = g(o)$. Here $[true]_{Y/\equiv}$ is a chosen representative for an equivalence class that corresponds to the equation being true, which is a fiber in X. The concept of fiber describes the equivalence class itself, which we identify with the C++ class being described. In the example, the equivalence class is the set of all points $o \in X$ in the circle. Interpretation of $Y/\equiv$ is the collection of equivalence classes where the data fields described by data in Y obtained from object in X are classified according to the chosen equivalence relation, here typically to two parts. The two parts can be computed using equalizer, which can be done in same situations than the computation of the coequalizer. This produces $e : E \to X$. Putting all of this together, we have following commuting diagram:



What this means is that the objects $o : X$ are classified by the equation to two groups based on whether $o \in e$. And this means that private data of the objects o are classified by the equation $f(o) \equiv g(o)$ to equivalence classes using the coequalizer. In the example, the equivalence class describes whether the point described by the coordinates in the object private data is on the circle of radius 10, so $[true]_{Y/\equiv}$ chooses the circle equivalence class. In particular, if the epimorphism $q : Y \to Y/\equiv$ has an inverse (it has many. Term "section"

is sometimes used for those), then $q^{-1} \circ [true]_{Y/\equiv}$ describes squared distance from origin (because f computes it) and this must be 10^2 (since g computes it), and because an equivalence class from the coequalizer q is chosen to satisfy the equation, so its inverse only produces data that satisfies the equation.

Further, if there is a common inverse $u : Y \to X$ for f and g (there can be many), then we can describe an object $o = u \circ q^{-1} \circ [true]_{Y/\equiv}$, and we can see that such objects are selected in such way that only points on the circle can be members. However, typically constructors for X would allow coordinates of both coordinates to be set individually, and by default nothing guarantees that such input satisfies the equation. So checking the constraint condition may be needed.

Therefore, for those objects, the equation will be true, so $o \in e$ proves that $o \in E$. And e is an equalizer for f and g, so if we have inverse arrows for q and e, and a common inverse u for f and g, we have computed the equalizer and proved with $e^{-1} \circ o = e^{-1} \circ u \circ q^{-1} \circ [true]_{Y/\equiv}$ that the instance o satisfies the equation. Therefore, computation of both the coequalizer and the equalizer will allow such equations to be used. The coequalizer describes "internal view" of the object by describing how the defining equation is computed and how this gives rise to an equivalence class. And the equalizer describes "external view" by describing membership of individual object instances in the equation. The computation of the three inverse functions works like `dynamic_cast`, each might not exist and it may be necessary to account for error cases.

Notice that we could instead choose $[false]_{Y/\equiv}$ as the chosen equivalence class by inverting the equation condition. However that case is less well behaved since we do not know that the corresponding apartness relation is an equivalence relation.

So in principle, each class with a class invariant has an inverse which consists of those instances that failed to satisfy the class invariant. However that case maps to instances whose private member variables contain corrupted data, so is a case we would like to avoid. If there was a need to process such corrupted data during error handling, then classes with negated class invariants might be useful. However that case would be out of scope of this paper.

The stateful nature of object instances in C++ is the part where I didn't model the object instances in detail in the theoretical model. The reason is the model of memory in the theoretical model would be too complicated and would produce a very low-level point of view to the theoretical model. The history morphisms that were used to model object instances are not adequate to describe the full complexity of the state changes in the object, and in general the model would need to account for multiprocessing and concurrency, including race conditions. However, when class invariant restricts membership of class instances in the class, the stateful changes should be accounted for. In particular, the theoretical model that uses the history morphisms considers two snapshots of same object instance at different times to be different history morphisms. This means that if a state change causes class invariant to fail for a snapshot of an instance described by such history morphism, suddenly the object stored in the instance no longer satisfies the requirements for the class, and somehow the code has left the scope of the class. If this happens for the storage of the instance, then the data stored in the instance is considered corrupted. Such scenario can happen temporarily within a critical section, but should be very quickly corrected.

The concept of "pushout" describes how variants interact with class invariants. When the variant alternative chooses a particular branch during `std::visit` operation, the class invariants associated with the branch selected are applicable. This might impact what data pattern matching on the variant alternatives can utilize based on the class invariant. Anyway, once it's known that the variant alternative is (an instance of) a class with a class invariant, the pattern matching branch that discovers an instance of such class can assume that the class invariant associated to that class is true. And once that branch has been checked, the subsequent branches might be able to infer that the class invariant is false, just because the previous branch already accounted for that case.

Notice however that variants are not consided an object-oriented construct, and visit operation on variants in particular interacts with the inheritance hierarchy. In particular, adding a class invariant to a class produces a subclass with fewer instances than the class that was started with. In particular, there should be a way to ignore the class invariant with an automatic derived-to-base conversion from a class declared with an invariant into a variable that doesn't define the invariant, but is otherwise similar. This conversion would correspond to ignoring the invariant, skipping the failures that would occur if the contract checks fail. Similarly, the corresponding opposite operation, that is `dynamic_cast` from base class to derived class where the derived class adds a new constraint should validate the class invariant, and fail the cast if the invariant is not valid! This is the answer to the question presented as to where the class invariant condition should be checked, such check belongs to an operation similar to `dynamic_cast`, at least when the constraints are natively supported in the language. The `static_cast` would instead simply assume the constraint without checking. If not supported by core language, this could be defined in a library.

I think this ability should be available if the class invariant would be defined in a derived class. A derived class that just adds an invariant seems to be a useful tool, because then the inheritance hierarchy would automatically provide the necessary functionality.

In addition, the theory contains a notion of a Turing category and Restriction categories that express a notion of constrained datatypes in partial functions by mapping failures of contract checks to nontermination [3]. Thus contract checks are modelled as restriction of domain of the operations to a subset where the contract checks succeed. However, in context of C++, I view nontermination as too drastic way of handling contract violations. The domain of functions would definitely be impacted by the restriction, so the constrained datatypes would need to be supported in function parameters and in library for example in `std::function`.

In general, if the assertion checking is forced to be performed at compile time, the resulting type system must have dependent types, because "constrained tuples" could mix compile-time evaluated functions with run-time evaluated functions.

There is an extensive theory on all these concepts, which is mostly outside the scope of this paper.

7.1 Loop invariants and conditional statements

Certain control flow constructs already check for a condition at run-time. Such conditions typically imply that the condition itself is true or false at specific points of the program execution.

```
int count = 10;
while (--count > 0) {
  contract_assert( count > 0 );
  // ...
}
contract_assert( !(count > 0) );

if (x > 13) {
  contract_assert( x > 13 );
  // ...
} else {
  contract_assert( !(x > 13) );
  // ...
}

int count = 10;
do {
  // ...
} while (--count > 0);
contract_assert( ! (count > 0) );
```

However, since these are so obvious, usually programmers are not writing these explicitly unless the logic is very complicated.

Notice that in these cases, the assertion is attached between two statements (or to an empty statement), and implication is that previous statements were able to establish the contract (or the corresponding check fails) so therefore, the next statements can rely on the assertion to be true, at least until the variables referred to in the assertion are again modified.

7.2 Constrained types

In principle, variable declarations should have possibility for expressing invariants [9].

```
int i{0}  invariant( i > -3 && i < 10 );
```

This would mean that, when the variable is initialized, it's first necessary to check that the invariant on the initialized value is valid. And when the variable is assigned or modified, it should again satisfy the invariant.

This is possible in current C++ as follows:

```
class ConstrainedInt
{
public:
  ConstrainedInt(int i) : value(i)
  {
     if (not (i > -3 && i < 10)) { throw assertion_failed(); }
```

```
  }
private:
  int value;
};
```

However, writing the above kind of class for every constraint seems excessive, and that's even before I've considered assignment or copy constructors.

Notice that presence of invariant may actually require the variable to be initialized, since uninitialized variables are unlikely to satisfy the invariant.

So `i = 3;` would be valid, but `i = 11;` would fail the invariant. So the assignments and initialization for a variable with an invariant could be re-written to include a check for the invariant.

```
i = 3; if (i > -3 && i < 10) {} else { throw assertion_failed(); }
```

From design point of view, these invariants are either considered to be associated with the variable itself, or to the assignment operation. However typically lots of variables are written, and it would be lots of work to write separate classes with assignment operations for every one of them (but with templates, maybe that's possible).

However, this approach suffers from a concurrency problem. It's not guaranteed that the assignment and the corresponding check are atomic. So typical approach for performing the check is that the *value* to be assigned is checked before assignment.

```
int temp = 3;
if (temp > -3 && temp < 10) {} else { throw assertion_failed(); }
i = temp;
```

However, the previous code was checking the condition in a constructor. And if this transformation is performed for the constructor, the check is then clearly before the initialization of member variables in a constructor. For constructor, checking the value after initialization is ok, since exceptions thrown from constructor will destroy the object, and therefore already-performed initializations have no effect, except for perfoming unnecessary computation. Unless of course they can have side effects.

The assignment case is very different. In assignment, the order of the validity check and the change of the member variable contents is important. The constraint should be checked before any assignments are done to avoid breaking the class invariant. Ideally whole "transaction" for modification should be checked before any changes made by it are performed.

If there are more than one variable that has such invariant, there is no guarantee that checking the invariant can be performed in atomic way. But using a correct ordering of the constraint check and the assignments helps. So for

```
struct Structure {
  int i invariant ( i > -3 && i < 10 );
  int j invariant ( j < 0 );
};
```

When initializing the fields, it's possible to first perform all validation:

```
Structure s{3,-5};
```

becomes:

```
const int si = 3;
const int sj = -5;
if (si > -3 && si < 10) {} else { throw assertion_failed("i > -3 && i < 10"); }
if (sj < 0) {} else { throw assertion_failed("j < 0"); }
Structure s{si,sj};
```

Sometimes, the individual field constraints depend on the value of another field, so cannot be described independently.

Usually such invariants should be attached to a class as a class invariant.

When modifying structure containing multiple fields from multiple threads, there are two approaches that can be used. Either the fields should be marked atomic, or mutexes should be used.

However, this choice has important consequences for constraint checking. two atomic variables that are linked by a constraint cannot really be checked atomically.

```
// Bad example:
struct Structure {
  std::atomic<int> field_1;
  std::atomic<int> field_2;
  class_invariant( field_1.load() > field_2.load() );
};
```

This is clearly unreasonable. There is no guarantee that other threads do not modify the atomic variables between the two loads. Of course, in a concurrent scenario, it may not matter which version of the value is used in the check. And so the check may be ok. However it doesn't make the overall structure indivisible/atomic as an object, as described in object-oriented paradigm, since concurrent changes may split the object (some field values checked are based on the new modification, and some values are from previous modifications).

```
struct Structure {
  bool invariant() const {
    std::lock_guard<std::mutex> g(m);
    return field_1 > field_2;
  }
  class_invariant( this->invariant() );
  mutable std::mutex m;
  int field_1, field_2;
};
```

With mutex, the situation is better and concurrent modification granularity is larger, because the code can reserve the mutex before attempting to evaluate the invariant. However, typical operations on individual fields still prevent satisfying the invariant:

```
Structure s; // Error, cannot default-initialize.
Structure s2{{},10,3}; // OK
s2.field_1 = 1; // OOPS, invariant broken after this statement.
s2.field_2 = -3; // Would be ok after this, if assignment doesn't check invariant.
```

Due to this problem, typically object-oriented approach is used where fields are declared private, the methods control how modifications to the fields can be done, and the mutex is taken in methods can be used. Or transactional memory TS could be used.

```
class Cls {
    Cls(int f1, int f2)
     post( class_invariant() )
    : field_1{f1}, field_2{f2} {}

    Cls &operator=(const Cls &o)
    {
      if (this != &o) {
        std::lock_guard<std::mutex> g(m);
        contract_assert( class_invariant() && o.class_invariant() );
        field_1 = o.field_1;
        field_2 = o.field_2;
        contract_assert( class_invariant() && o.class_invariant() );
      }
    }
    ~Cls() pre( class_invariant() ) {}
  private:
    bool class_invariant() const noexcept {
      return field_1 > field_2;
    }
    mutable std::mutex m;
    int field_1;
    int field_2;
};
```

The `class_invariant` method must be declared private, since it must be called within the lock.

Instead I could pass the lock to the class invariant operation.

```
class Cls {
  // ...

  Cls &operator=(const Cls &o)
  {
    if (this != &o) {
      std::lock_guard<std::mutex> g(m);
      std::lock_guard<std::mutex> g2(o.m);
      contract_assert( class_invariant_condition(g)
                  && o.class_invariant_condition(g2) );
      field_1 = o.field_1;
      field_2 = o.field_2;
```

```
      contract_assert( class_invariant_condition(g) );
    }
    return *this;
  }
  bool class_invariant_condition(std::lock_guard<std::mutex> &g) const noexcept {
    return field_1 > field_2;
  }
private:
  mutable std::mutex m;
};
```

Of course, it's not strictly necessary to take the argument, I might use recursive locks, which is better in the sense that I wouldn't need to account for whether the mutex is already locked. However, when using non-recursive locks it is useful in declaring that the class invariant should only be evaluated when the mutex is locked. The effectiveness of the mutex of course depends on which mutex is locked by the guard. So actually such code would want to ensure that the mutex of `this->m` must already be locked. Maybe that can somehow be expressed as a contract precondition? However `std::mutex` doesn't have a test for that, and neither has `std::lock_guard<>`. However, `std::unique_lock<std::mutex>::owns_lock()` can be used for that when `std::unique_lock<>` is used for locking. Unique lock also has operation to check which mutex the lock is locking.

```
  bool class_invariant_condition(std::unique_lock<std::mutex> &lock) const noexcept
    pre (lock.owns_lock() and lock.mutex() == &this->m)
  {
     return field_1 > field_2;
  }
```

This case is actually pretty compelling use for preferring `std::unique_lock` over `std::lock_guard`.

7.3 Assertions and optional

Assertions should interact in a specific way with `optional`. Suppose I construct an instance of `optional<int>`, then assert that the optional contains a value.

```
std::optional<int> value{1} invariant( value.has_value() );
```

By assuming that the invariant is satisfied, I could instead write more efficiently:

```
int value{1};
```

However I don't hope for the compiler optimizations to perform this deduction.

Now suppose I could write this:

```
template <class T>
using mandatory = std::optional<T>  invariant( has_value() );
```

I think associating invariants to type aliases seems difficult from compiler point of view, so this is not really a proposal for extension.

There are alternative mechanisms for obtaining same results, for example:

```
template <class T> class mandatory : public optional<T>
{
public:
  mandatory() = delete;
  constexpr mandatory(const T& x) : optional<T>{x} {}
  constexpr bool has_value() const { return true; }
};
```

7.4 Combining contracts

```
[[assert: cond]] [[assert: cond]] == [[assert: cond]];
```

I would ideally like to always optimize the assertions so the same condition is only checked once. This property that same condition, applied twice, means same as the condition checked once, is called "idempotence" of the assertions. However it's not always possible.

I'd interpret this condition in another way in context of assertions. I require all assertion conditions to be idempotent. This is a formal expression of one interpretation for thea idea that the condition doesn't perform side effects. Of course, if there is a side effect that would cause problems during contract checking for the execution of the condition, this idempotence cannot be true, because performing same side effect twice is usually *not* same as performing it once.

There is a huge difference in checking the same condition once and not checking it at all.

However certain side effects can be idempotent. For example, assigning the same value to a variable twice is same as assigning it only once, assuming no other operations occur in between (generally requires a single-threaded environment, a critical section established using a mutex, or thread local storage).

I consider idempotence of the assertion conditions the distinguishing criterion when the assertion can be part of an interface. If the condition is not idempotent, then it should not be declared as part of the interface. Interface always has two sides which have independent reasons for checking the contract conditions.

Idempotence doesn't mean that assertions must be checked more than once. However in practice, different sides of interface can be deployed in different places, can be separately compiled, etc. So it's difficult to ensure in compilation of interface anything that's not compiled in the same translation unit because it's in the other side of the interface.

Interface describes those things that **must** be common for the different participants to an interface for communicate and interact. So defining the contract condition in an interface *means* that both sides must agree what the contract is. The question is, does the interface then also describe which side checks for the contract? Both can do it, if that's agreed in the interface.

Of course, the idempotence rule for assertions derives from the corresponding idempotence rule of the boolean binary "and" operation:

```
x && x == x
```

Because if the condition is different, then you must combine them with "and".

```
[[assert: cond1]] [[assert: cond2]] == [[assert: cond1 && cond2]];
```

If it happens that the two conditions are the same, cond1 == cond2, then the idempotence of rule for the boolean binary and operation can be used to deduce the rule that same assertion, checked twice, is same as the assertion, checked once.

Similarly, if there are two preconditions, or two postconditions, they can be combined with an and operation.

```
[[pre: cond1]] [[pre: cond2]] == [[pre: cond1 && cond2]]

[[post: cond1]] [[post: cond2]] == [[post: cond1 && cond2]]
```

What about class invariants. Two class invariants can be combined:

```
class A
{
private:
  int i, j;
  class_invariant( i > 0 );
  class_invariant( j + i > 3 );
};
==
class A
{
private:
   int i,j;
   class_invariant( i > 0 && j + i > 3 );
};
```

7.5 Concurrency of class invariants

However, atomic processing of member variables should be considered.

```
  class A
  {
  public:
    A() : i(1), j(3) {}
    void f() {
       std::lock_guard<std::mutex> g(m);
       ++j;
    }
    ~A() { }
  private:
    bool class_invariant() const noexcept {
      return i > 0 && j + i > 3;
    }
    std::mutex m;
    int i,j;
  };
```

But there is a problem, should I take the mutex in the `class_invariant` operation? If I do, and then invoke the operation from asserts in the methods, I take the same mutex twice. With recursive mutex, this can be handled, but with normal mutex, I need to be more careful. If I don't take the mutex in `class_invariant`, then I must be careful not to invoke it without taking the mutex.

With preconditions and postconditions, I would have to write as follows:

```
class A
{
public:
  A(int i_, int j_)
      post( class_invariant() )
    : i(i_), j(j_)
  {
  }
  void f() {
     std::lock_guard<std::mutex> g(m);
     contract_assert( class_invariant() );
     ++j;
     contract_assert( class_invariant() );
  }
  ~A()
    pre( class_invariant() );
  {
    // ...
  }
private:
  bool class_invariant() const noexcept {
    return i > 0 && j + i > 3;
  }
  std::mutex m;
  int i,j;
};
```

Notice here I must place the pre and post conditions generated from class invariant inside the mutex lock, because if the checking is performed, I am reading the data of the object. In a constructor, the class invariant expands to a postcondition. In a destructor, the class invariant expands to a precondition.

Constructor must be started when there is only one possible user for the instance being created, and destructor should finish in situation where there is only one possible user. Otherwise I get a race condition that should be handled by some sort of mutual exclusion, e.g. mutex or condition variables. Therefore constructors and destructors do not need to obtain mutex unless they manipulate global state.

For implicitly existing operations such as assignments, copy constructors, move constructors etc, usually the class invariant of the source object transfers without trouble to the target object. But in principle those would be subject to contract checking as well. The default constructor is an exception, since default-constructed instances are unlikely to satisfy the class invariant, unless that has been explicitly accounted for.

Normally, if I declare a mutex in a class, I usually must take it in methods and release it when the method execution is done. The class invariant must be checked both when entering the method, and when leaving it.

However, in context of a class, if I'm doing proper locking, and the data members accessed are private, it is guaranteed by the method postconditions that contain the class invariant that the "next method called on the object" still satisfies the class invariant when the method is entered, because the data members are private so cannot be changed before the next method call. If the class invariant would depend on global (public) state or the data members would be public, the situation would be dramatically different.

All this changes if the locking of the mutex is left out, because then any invocations to the methods from multiple threads will cause race conditions that can impact the invariant. But if I get calls from only one thread, then I could remove the synchronization.

Now, since checks for the invariant properties are always *read* operations, in theory only *shared lock* is needed during checking of the invariant. But it could happen that the method itself whose precondition is being checked also performs writes. I don't expect that compiler optimization can really utilize this information though.

References

[1] Linux manual page, shutdown. https://www.man7.org/linux/man-pages/man2/shutdown.2.html.

[2] Abelson, Sussman, and Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1984.

[3] John Baez. Turing categories. https://golem.ph.utexas.edu/category/2019/08/turing_categories.html, 2019.

[4] Lawrence Crowl. Abandoning a process. https://open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2440.htm, 2007.

[5] Edgar Dijkstra. Goto statement considered harmful. *Communications of the ACM*, 11(3), March 1968.

[6] Oracle Inc. Java object serialization. https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html, 2024.

[7] Alexander Beels John Lakos, Alexei Zakharov. Centralized defensive-programming support for narrow contracts. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4075.pdf, 2014.

[8] Timur Doumler Joshua Berne and Andrzej Krzemieński. Contracts for c++. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/p2900r7.pdf, 2024.

[9] Andrzej Krzemieński. Value constraints. https://open-std.org/JTC1/SC22/WG21/docs/papers/2014/n4160.html, 2014.

[10] Andrzej Krzemieński. Contract violation handlers. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2339r0.html, 2021.

[11] Gašper Ažman Krzemieński Andrzej. Abort-only contract support. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2388r0.html, 2021.

[12] F. William Lawvere and Robert Rosebrugh. *Sets for Mathematics*. Cambridge University Press, 2003.

[13] Bertnard Meyer. *Object-oriented software construction*. Prentice Hall PTR, 2nd edition, 1997.

[14] Bjarne Stroustrup. *Design and Evolution of C++*. Addison-Wesley Publishing, 1995.