**Document Number**: P0166R0
**Date**: 2015-11-06
**Reply to**: J. Daniel Garcia
**e-mail**: josedaniel.garcia@uc3m.es
**Audience**: EWG

# Three interesting questions about contracts

J. Daniel Garcia
Computer Science and Engineering Department
University Carlos III of Madrid

## 1 Introduction

During the 2015 Kona meeting, Herb Sutter made three very interesting questions on contracts in order to facilitate discussions and make progress on a contracts proposal.

The basic idea was to explore how could a proposal be used to eventually write a bounds check contract on the subscript operation. That is, given the a potential `myvector` class:

```
template <class T>
class myvector {
  //...
  T & operator[](size_t pos);
  // ...
private:
  size_t  size;
  T * buf;
};

template <class T>
T & myvector::operator[](size_t pos)
{
  return buf[pos];
}
```

We want to answer the following questions:

1. How do I express a range check that `0 <= pos < this->size` ?
2. How do I decide whether the function should be noexcept?
3. How do I select the whole program violation handling?

This paper explores possible answers to this questions using the proposed example to explore all of them. Examples are expressed following the lines of syntax from [1].

In particular, I pay attention to the tension between different uses of the contracts facility in different domains and its relationship to exceptions management.

## 2 Effects on contract violation

Multiple effects have been suggested when a contract is violated. It is extremely important to remark that the set of valid effects is highly dependent on the type of applications being used.

In general the fact that a contract has been violated is a symptom that the program has a defect. That is, a contract is not violated because of any exogenous circumstance, but because the program is incorrectly using a function.

### 2.1 Fail-fast and termination

For some systems, once a defect has been detected the only thing that can be done is terminating the program. This is specially true for some safety critical applications and some real time systems.

A minor difference here is whether the system can eventually perform some final operations in the form of a `terminate()` handler.

### 2.2 Handle and continue use case

There are use cases where what the application developer wants is to handle is some way the contract violation and the proceed with the program. A use case that has bee repeatedly reported by Bloomberg is the case where upon contract violation you may want to log the contract violation and proceed with the program execution.

### 2.3 Recovering from a software defect

There are also systems where after a contract violation the system may recover and continue. One typical example is when a program as a structure where plug-ins may be added to the application. In those cases, a defect in a certain plug-in should not stop the whole application but the specific faulty plug-in.

Another example are applications that after a contract violation, still may want to cleanly save their state to a file and perform a regular cleanup.

### 2.4 Contracts and testing frameworks

Several unit testing frameworks use exceptions to notify that a test case has failed. This has been used sometimes to argue in favor of exceptions use to notify contract violation. However, a test case failure is different from a contract violation.

In general, with some given testing approaches it is relevant to be able to throw an exception when a contract violation happens. This would allow to test the contracts themselves, invoking functions with unexpected inputs to check that the contract is able to correctly detect the violation.

### 2.5 Summary of options

Given this plethora of different use cases, the following options seem to be needed by some subset of users:

- Be able to perform a fail-fast. Probably this could be achieved by just invoking `abort()`.
- Be able to terminate the program invoking the corresponding `terminate_handler()`.
- Be able to invoke a handler function and the resume execution at the calling site.
- Be able to report the violation by throwing an application wide exception (e.g. `contract_violation`).
- Be able to report the violation by throwing a specified exception (e.g. `out_of_bounds`).

This different behaviors could be specified by a build option. However, library writers still need the ability to specify two families of contracts:

1. **Non-throwing contracts**: Contracts that independently of the build mode used will never throw. Those contracts are essential to be able to specify preconditions and postconditions for operations that still want to provide the `noexcept` guarantee. The only valid effects for the violation of such contracts are some form of termination.
2. **Potentially throwing contracts**: Contracts that depending on the build mode may throw an exception. For these contracts if the build mode is to report throw exceptions the effect is throwing an exception. However, if the build mode is termination, they just terminate without throwing the exception.

## 3 Expressing contracts

### 3.1 Existing practice

I start by examining `vector::operator[]` in some popular implementation of the standard library. The code is roughly the following:

```
template <typename T>
T & vector::operator[](size_t pos) noexcept
{
  return *(buf + pos);
}
```

Firstly, the operation is marked as `noexcept`. Even, if the standard does not require the operation to be marked as `noexcept`, implementations are given the freedom to add this specifier. Secondly, the operation never throws. In fact, the source code for the debug version is slightly different and it lives in a different source file (as the the debug version of the data structure is also different).

```
template <typename T>
T & vector::operator[](size_t pos) noexcept
{
  check_subscript(pos);
  return *(buf + pos);
}
```

## 3.2 A simple contract

With contracts a class my express its expectations through a precondition.

```
template <typename T>
T & myvector::operator[](int pos) noexcept
[[expects: 0 <= pos && pos < size()]]
{
  return buf[pos];
}
```

In this example the contract is part of the function declaration and is visible to its callers. A key derived property is that the precondition is available to the caller and consequently to any static analysis tool even if the implementation code is not available. In this specific case, as the code is a `template` the source code is always available. However, there might be cases for non-templated code where the implementation is only distributed in binary format.

### 3.2.1 Effects on contract violation

When the build mode is terminating (fail-fast or terminate) the precondition violation implies program termination.

For a resuming handler build mode, the effect in this case would be rather catastrophic as the operation would then be returning a dangling reference.

Finally, in a the case of a throwing build mode, the effect would be equally termination as the operation has been marked as `noexcept`.

## 3.3 Implementation contracts

A different strategy would be using contracts in function bodies. There might be multiple reasons for a developer to do so. Firstly, some contracts might be expressed with a lower complexity as the algorithm proceeds. Secondly, an implementation contract may make use of implementation details not available in the declaration.

```
template <typename T>
T & myvector::operator[](int pos) noexcept
{
  [[check: 0 <= pos && pos < size]]
  return buf[pos];
}
```

As it has been stated earlier this approach would limit the static analysis in the case of non-inlined non-templated code.

### 3.3.1 Effects on contract violation

The effects would be here essentially the same that for the previous case of contracts in the declaration. The key point is that operation is `noexcept`.

### 3.4 A throwing index operator

Another option to be considered is that I might want to have an index operator which throws when the index is out of bounds.

```
template <typename T>
T & myvector::operator[](int pos)
{
  if (0 > pos || pos >= size) throw bounds_error{};
  return buf[pos];
}
```

In this case, the operation is not marked to be `noexcept` (i.e., it is `noexcept(false)`). Obviously, this approach has drawbacks. The check is now unconditional (which is asking to be controlled by some macro).

## 4 Being able to throw

As previously stated being able to throw on contract violation is essential for some approaches, while cannot be afforded in some other use cases.

### 4.1 Throwing contract

A possible solution is to be able to express a throwing contract. Following our example:

```
template <typename T>
T & myvector::operator[](int pos)
[[expects(bounds_error{}): 0<=pos && pos <=size()]]
{
  return buf[pos];
}
```

While this was the syntax originally suggested during the 2015 Kona meeting, a more generalized one can be achieved by adding an `effect` attribute.

```
template <typename T>
T & myvector::operator[](int pos)
[[expects: 0<=pos && pos <=size(), effect:throw bounds_error{pos,size}]]
{
  return buf[pos];
}
```

Regardless the syntax, whenever there is a precondition violation the exception is thrown. In this case the operation is not marked as `noexcept`.

#### 4.1.1 Effects on contract violation

When the build mode is terminating (fail-fast or terminate), the precondition violation still implies program termination. Although this might seen as counter-intuitive it is essential to support those application domains that cannot afford to make use of exceptions.

When the build mode is a throwing build mode an exception will be thrown. In the case of specified exception mode, the specific `bounds_error` exception is thrown. If the application is built with an application wide exception mode, then that exception takes precedence over the specific one.

If the application is built with the resuming handler mode, then the handler is executed and the exception is thrown, preventing the application to really resume.

## 5 Summary

As a summary, I explore different options to programmer in source code and the effects with each build mode.

### 5.1 Contract options

A programmer may express a contract as one of the following:

- **Non-throwing contract**: A contract that guarantees not to throw. A violation of such contract always results in some sort of program termination.
- **Potentially throwing contract**: A contract that might throw when the application is built in the appropriate mode.
- **Throwing contract**: A contract where the programmer specifically requests to throw. The fact that an exception is thrown or not is still dependent on the specific build mode used.

#### 5.1.1 Non-throwing contract

```
template <typename T>
T & myvector::operator[](int pos) noexcept
[[expects: 0<=pos && pos < size()]]
{
  return buf[pos];
}
```

#### 5.1.2 Potentially throwing contract

```
template <typename T>
T & myvector::operator[](int pos)
[[expects: 0<=pos && pos < size()]]
{
  return buf[pos];
}
```

#### 5.1.3 Throwing contract

```
template <typename T>
T & myvector::operator[](int pos)
[[expects: 0<=pos && pos < size(), effect: throw bounds_error{pos,size()}]]
{
  return buf[pos];
}
```

### 5.2 Build modes

The following modes are considered:

- **failfast**. The application terminates without further cleanup (e.g. calling `abort()`).
- **terminate**. The application terminates invoking the corresponding `terminate_handler`.
- **resume**. The application invokes a system wide handler function which then may return to resume executing despite the contract violation.
- **throw**. The application throws a system wide exception. This exception could be supplied to the build system or just a global predefined one (i.e. `broken_contract`).
- **throw-specific**. The application throws the exception specified in source code.

#### 5.2.1 Terminating modes

Modes *failfast* and *terminate* are essentially the same and, consequently, are studied here together. The only difference is that for *terminate* mode, the termination includes the execution of a user supplied handler and proper cleanup.

- Non-throwing contracts: The program terminates.
- Potentially throwing contracts: The program terminates without invoking any exception.
- Throwing contracts: The program terminates without invoking the user supplied exception.

### 5.2.2 Resume mode

Mode *resume* results in the execution of a user provided handler. The handler is supplied to the build system, and it is global for the whole application.

- Non-throwing contracts: The program runs the resume handler but does not resume. Instead it invokes the `terminate` function.
- Potentially throwing contracts: The program runs the resume handler and after that, it proceeds execution.
- Throwing contracts: The program runs the resume handler and after that, it throw the exception supplied by the contract.

### 5.2.3 Throwing mode

Mode *throw* results in throwing a global exception. This might be a predefined exception (e.g. `broken_contract`) or an exception supplied to the build system. In any case, the exception would be the same one for the whole application.

- Non-throwing contracts: The program ignores the global exception and terminates by invoking the `terminate` function.
- Potentially throwing contracts: The global exception is thrown.
- Throwing contracts: The program ignores the global exception and the exception specified by the throwing contract is thrown.

### 5.2.4 Throwing specific mode

Mode *throw-specific* results in throwing the exception specified by the contract if any.

- Non-throwing contracts: The program terminates by invoking the `terminate` function.
- Potentially throwing contracts: A global exception is thrown.
- Throwing contracts: The exception specified by the throwing contract is thrown.

Consequently, the modes *throw* and *throw-specific* can be merged into a single mode.

## Acknowledgements

Thanks to Peter Sommerlad for reviewing an earlier draft of this paper.

## References

[1] Gabriel Dos Reis, J. Daniel Garcia, Francesco Logozzo, manuel Fahdrich, and Shuvendu Lahri. Simple Contracts for C++. Working paper N4415, ISO/IEC JTC1/SC22/WG21, May 2015.