# A more predictable unchecked semantic

Document #: P3316R0
Date: 2024-05-22
Project: Programming Language C++
Audience: SG21 Contracts Working Group
Reply-to: Jonas Persson
<jonas.persson@iar.com>

## Contents

## 1 Introduction

In [P2900R5] there are Ignore, Observe, Enforce and Quick_enforce contract semantic. If your company is delivering libraries (static) this means you have to ship four times as many. This may be costly, so it is tempting to ship a single version and let the linker strip away the unused elements.
Those elements are the check, the handler and the abort. Keeping everything leaves us with Enforce, removing the handler leaves us with Quick_enforce, removing the abort leaves us with Observe and removing everything leaves us with Ignore. Almost. We are still left with a compiled library where the optimizer saw that failed checks would abort and hence optimize based on that.
That will not be the same had the library been compiled with Ignore semantics.

So what semantics is it then? It is not Ignore, but it is also not assume. Assume would have allowed the optimizer to optimize as if the check was always true. In this case we have a semantic where a program not fulfilling the contract is not expected to progress past the location of the check. This paper will call this semantic Promise, where passing this location establish the condition as a truth.

## 2 Naming

In this paper Promise is used as a placeholder name since it relies on the programmer promising to follow the contracts. The attribute is named establish as it makes something true. But I would appreciate some input from native speakers on better words to use. Other suggestions are presume, barrier.

## 3 The disappearing observer problem

The current Observer semantic has a weakness in that potential UB in a function that would have been avoided by an Enforced check is now subject to removal by the optimizer if the observer branch can be identified as not occurring in a valid program.
Consider a contract_assert and how it's expanded pseudocode for different contract semantics may look like after the optimizer pass:

**Source**

```
bool f(int* p, int* q)
{
  contract_assert(p != nullptr);




  return *p == *q;
}
```

**Enforce**

```
bool f(int* p, int* q)
{
  if(not (p != nullptr)) {
    invoke_handler();
    die();
  }
  return *p == *q;
}
```

#### 3.0.0.1 The unchecked variants then look like:

Remember that the dereference of pointers is undefined if the pointer is nullptr. This allows the compiler to backtrack and remove any branch where that is true.
For an Observe based on Ignore this means the whole handler branch may be removed. For Observers based on Promise from this proposal or the observable barrier from [P1494R2], the optimization is stopped from going back and remove earlier branches.

**Observe/Ignore**

```
bool f(int* p, int* q)
{




  return *p == *q;
}
```

**Observe/Promise**

```
bool f(int* p, int* q)
{
  if(not (p != nullptr)) {
    invoke_handler();
    [[establish(p != nullptr)]]
  }
  return *p == *q;
}
```

**P1494**

```
bool f(int* p, int* q)
{
  if(not (p != nullptr)) {
    invoke_handler();
    std::observable();
  }
  return *p == *q;
}
```

To see difference between Promise and std::observable we need to add another branch to the example. Since dereferencing q gives that q is not null, the first branch will never be taken in a valid program. In Enforce mode it may be removed.

**Source**

```
bool f(int* p, int* q)
{
  if(q != nullptr);
    puts("got null");

  contract_assert(p != nullptr);


  return *p == *q;
}
```

**Enforce**

```
bool f(int* p, int* q)
{



  if(not (p != nullptr)) {
    invoke_handler();
    die();
  }
  return *p == *q;
}
```

The same goes for Observe and Promise. But for std::observable all optimizations is blocked and the branch must be kept.

**Observe/Ignore**

```
bool f(int* p, int* q)
{






  return *p == *q;
}
```

**Observe/Promise**

```
bool f(int* p, int* q)
{



  if(not (p != nullptr)) {
    invoke_handler();
    [[establish(p != nullptr)]]
  }

  return *p == *q;
}
```

**P1494/std::observable**

```
bool f(int* p, int* q)
{
  if(q != nullptr);
    puts("got null");

  if(not (p != nullptr)) {
    invoke_handler();
    std::observable();
  }

  return *p == *q;
}
```

The same kind of effects also happens with pure Ignore checks and for pre and post conditions.

## 4 Ignore considered harmful

When writing code in the presence of contracts, one major benefit is that code outside contracts need not be correct.

**Defensive style**

```
int& deref(int* p)
    pre (p != nullptr)
{
  if(!p) throw err;
  return *p;
}
```

**Contract style**

```
int& deref(int* p)
    pre (p != nullptr)
{

  return *p;
}
```

This will put us in one of two situations

1) Code is written in contract style. Compiling in Ignore mode because we are unsure if everyone follows the contracts will cause lots of unpredictable UB when it happens.
2) Code written in defensive style. Lots of unnecessary defensive code will need to be written in case the code is ever compiled with Ignore, and a lot of extra tests to make sure it works.

If the default unchecked mode was Promise, all code could be written in Contract style.

Ignore is still useful, but code around it must be in costly defensive style. Only code prepared for it should have its contracts ignored.
New contracts on probation could be made as `pre ignorable` to make it clear in the code it is unreliable. Further details on this is for another paper.

And as seen in previous examples Promise generates code identical to Enforce while Ignore behave differently in many situations. Running tests with Enforce and then Ignore when releasing is unreliable.

## 5 Mixed mode applications

With mixed mode we mean an application linked using modules with different contract semantics.

When some modules is Ignore and some are Enforced, do we trust cross module calls? Why was that module compiled with Ignore. With Ignore we do not know what to expect. Contracts may or may not be respected. With Promise there is a promise that they will be followed even if no runtime check is done. Such mixed mode applications can be considered safe.

## 6 Proposals

Promise semantics is generally more well behaved and let developers safely code in a cleaner contract style

**Poll**: Replace Ignore with Promise as the recommended unchecked mode.

An Observe with promise semantics will not risk having the handler optimized away.

**Poll**: Use Observe based on Promise as the recommended observe mode.

For completeness we want a new attribute similar to [[assume]]

**Poll**: Add a new attribute [[establish( expression )]] that declares expression to be true after but not necessarily before

## 7 Conclusion

Ignore is unreliable and dangerous and should be replaced with the more reliable Promise semantics. It will behave more in line with Enforce and it will make expectations on contract style code more clear. With this change mixed mode builds will be well behaved as long when composed of modules using the four standardised contract semantics. It will also allow contract mode selection to be defer to the linker with the same semantics as if selected at compilation.

## 8 References

[P1494R2] S. Davis Herring. 2021-11-13. Partial program correctness. https://wg21.link/p1494r2

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