Document Number: P1290R0
Date: 2018-11-26
Reply to: J. Daniel Garcia
e-mail: josedaniel.garcia@uc3m.es
Audience: EWG

Avoiding undefined behavior in contracts

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

Abstract

The current wording for contracts opens some opportunities for undefined behavior. Those opportunities derive from the freedom for assuming a contract even when the program is built in a mode where the contract has not been checked. This paper proposes a minimal set of changes to solve the issue by requiring that only contracts that have been checked can be assumed. Additionally, this paper proposes an additional build option to decide whether axioms are assumed or not. Although this is the current status, there is a concerted effort to evolve from this status to a proposal that can be supported by all authors of the original design and specification of contracts.

1 Introduction

Consider the following code:

```
int f(int x)
  [[expects: x>0]]
{
  return 1/x;
}

int g(int x)
  [[expects audit: x==2 || x==3]]
{
  return f(x);
}
```

Under the current wording [1], contracts that are not checked can be assumed to be true. Consequently, we might have the following behaviors:

- When build mode is **audit**, both checks are intended to be evaluated and consequently assumed. As a consequence `f()`'s precondition can be elided even its checking is on as it can be assumed to always be satisfied.

- When build mode is **default**, only the check in `f()` is intended to be evaluated. However, assuming the precondition in `g()` implies that the precondition inf `f()` is always satisfied and so the check can be elided even though it is on. Consequently, invoking `g()` with a value of `0` would lead to undefined behavior.

- When build mode is **off**, no check evaluated. However, they are both assumed leading again to undefined behavior.

Additionally, when the continuation mode is **on** (continue after violation) the assumption brings in additional undefined behavior as now we are assuming conditions that might be false (because the failed and we returned after running the violation handler).

This original intent of allowing assumptions was to provide an ample margin for optimization. However, the above example is an illustration on how assumptions may lead to undefined behavior.

The interactions of contracts and undefined behavior have been explained in detail in [2]. However, it should be noted that the main goal of contracts is allowing to write more correct software by helping to detect more programming errors. Introducing new undefined behavior was an unintentional effect that needs to be avoided.

Of course, a secondary effect of contracts is giving compilers leeway to perform optimizations. The aim is to satisfy this goal only in the cases that the first goal is not sacrificed.

2 Potential for undefined behavior

In this section we analyze some examples of possible undefined behavior.

2.1 Example 1

In [2] an example provided by Herb Sutter and prototyped by Peter Dimov (see at `https://godbolt.org/g/7TP7Mt`) with simulated contracts is presented.

Essentially this example translated into contracts syntax would be:

```
void f(int x)  [[expects audit: x==2]]
{
  printf("%d\n", x);
}

void g(int x)  [[expects: x>=0 && x<3]]
{
  extern int a[3];
  a[x] = 42;
}

void foo();
void bar();
void baz();

void h(int x)  [[expects: x>=1 && x<=3]]
{
  switch(x) {
    case 1: foo(); break;
    case 2: bar(); break;
```

```
    case 3: baz(); break;
  }
}

void test()
{
  int val = std::rand();

  try { f(val); /*...*/ } catch(...) { /*...*/ }
  try { g(val); /*...*/ } catch(...) { /*...*/ }
  try { h(val); /*...*/ } catch(...) { /*...*/ }
}
```

2.1.1 Current status

With the current definition of contracts, compiling the code with the build mode set to **audit** is not problematic. The precondition at `f()` is checked and it can be assumed to be true in next calls. Then, calls to `g()` and `h()` can optimize out the contracts under the assumption that `x` is `2`.

However, if the build mode is set to **default**, the precondition at `f()` would not be checked, but still assumed. Consequently, after the call to `f(val)` the compiler would be allowed to assume that `val` is `2` and the preconditions of `g()` and `h()` would be assumed to be correct and optimized out. This would lead to undefined behavior. For calls to `h()` it might be the case that we got the surprising effect that no function is called. But even worse, if the switch is implemented as a jump table and the compiler assumes the contract and elides the jump table bounds check, then a wild branch would arise. The generated code would attempt to read out-of-bounds at `__jmptbl[val]`, reinterpret whatever bytes it finds there as an address of executable code, and jump there to continue execution. This would result in random code execution and a very serious security issue.

2.1.2 Avoiding unchecked assumptions with disabled continuation

If we change the situation to require that no assumption of unchecked contract can be made when the continuation is disabled, the outcome is quite different.

Compiling the code with the build mode set to **audit** would not be problematic and would lead to the same outcome than with the current wording.

When the build mode is set to **default**, the precondition at `f()` would not be checked and would not be assumed. Consequently, after the call to `f(val)` no assumption can be made on the value of `val`. The preconditions of `g()` and `h()` would not be optimized out and the checks would be performed. No undefined behavior happens.

2.1.3 Avoiding all assumptions with enabled continuation

If we avoid all assumptions when continuation is enabled, we also avoid the possible undefined behavior derived from assuming a failed contract.

2.2 Example 2

This example is a variation of previous example, which is also discussed in [2]. In this variation the precondition at function `f()` is now moved the be an axiom.

```
void f(int x)  [[expects axiom: x==2]]
{
  printf("%d\n", x);
}
```

With the current definition of contracts, axioms are always assumed.

When the build mode is set to **default**, the precondition at `f()` would not be checked (as it is an axiom), but still assumed. In this case, the contract elimination is considered to be intentional as an axiom is considered to be always true.

When the build mode is set to **off**, no precondition is checked, but `val==2` is still assumed.

However, in some cases it might be interesting to be able to remove assumptions introduced by axioms. That would be the case, in a debug version where the developer wants to remove all possible assumptions. On the other hand, there are cases where axioms are desired to be used as assumptions. We consider this aspect orthogonal to the checking level induced by the build mode.

2.3 Example 3

Consider now this simple example:

```
void f(int * p)  [[expects axiom: p!=nullptr]]
{
  if (p) g();
  else h();
}
```

When axioms are assumed `f()` would be optimized to always call `g()`. That is not always desirable. Again the ability to control independently whether axioms are assumed or not gives us what we need.

3 Assumptions and continuation mode

3.1 Basic example

Consider now the following code

```
void f(int * p)  [[expects: p!= nullptr]]
{
  if (p) g();
}
```

3.1.1 Disabled continuation and check default contracts

If we compile with continuation mode set to **off** (the handler never returns) and the checking level is set to **default**, the compiler can use the information from the precondition. The generated code would be essentially the following:

```
void f(int * p) {
  if (p==nullptr) {
    __invoke_handler(); // Never return
  }
  else {
    g();
```

```
  }
}
```

The assumption that `p` is not `nullptr` is derived from the structure of the generated code and no special provision is needed to state that the contract is assumed.

3.1.2 Enabled continuation mode and check default contracts

If we compile with continuation mode set to **on** (the handler might return) and checking level set to **default**, the compiler would generate a different code structure that would be essentially:

```
void f(int * p) {
  if (p==nullptr) {
    __invoke_handler (); // May return
  }
  g();
}
```

In this case, the contract (`p!=nullptr`)) is checked, but if it fails is not assumed. Again, no special provisions are needed, the behavior is the consequence of the structure of generated code.

3.2 Another example

Let's try another example:

```
void f(int i) {
  [[ assert : i==0]]
  [[ assert : i>=0]]
  g();
}
```

3.2.1 Disabled continuation and check default contracts

In this case, the generated code would be equivalent to:

```
void f(int i) {
  if (i!=0) __invoke_handler (); // does not return
  else {
    if (i<0) {             // Always false as i==0 is always true
      __invoke_handler (); // does not return
    }
    g();
  }
}
```

The second check can only be called if the first one was successful. But if the first was successful `i` must be `0` and the second one will be optimized out. Note, that again this is a consequence of the generated code structure, and the resulting code would be similar to:

```
void f(int i) {
  if (i!=0) __invoke_handler (); // does not return
  else {
    g();
  }
}
```

3.2.2 Enabled continuation and check default contracts

In this case, the generated code would be equivalent to:

```
void f(int i) {
  if (i!=0) __invoke_handler(); // may return
  if (i<0) __invoke_handler(); // may return. Never optimized out
  g();
}
```

Now, both checks are independent and no one can be elided.

3.3 Consequences

As it has benn shown through examples, no special provision in the standard is needed to state our goal. Under disabled continuation mode a contract check implies its assumption. Under enabled continuation mode a contract check does not imply any assumption at all.

4 Proposed solutions

4.1 Avoiding the undefined behavior

This paper proposes to avoid the undefined behavior by clarifying the semantics of every build mode in regards of both evaluation of conditions and assumption of those conditions. For assumption of conditions the clarification needs to address the case where continuation is disabled and when the continuation is enables.

To define such semantics, the following simple principles are proposed to be followed:

- A contract that, in a given build mode, is not evaluated cannot be used for any kind of assumption. This leads to modes where only contracts that have been checked are used for assumptions and avoiding in this way the identified paths towards undefined behavior.
- Moreover, contracts that have been checked can only be assumed if the continuation mode has been disabled. Otherwise, such assumptions cannot be made. Note, that no special provision is needed as the application of general rules of conditional statements would derive this behavior as illustrated in previous sections.
- Axiom contract are considered as if they had been evaluated when they are enabled. Otherwise, they are ignored. Note that in any case, they need to have a valid syntax.

Below, the exact semantics of each build level are identified when they are applied to each contract level.

4.1.1 Build mode

The build mode can be any of the following three: **off**, **default**, **audit**. This build level affeccts which checks are evaluated at run-time.

contract-level	Build mode		
	off	**default**	**audit**
`audit`	no	no	yes
`default`	no	yes	yes

Note, that only `audit` and `default` checks are affected by the build mode.

4.1.2 Continuation mode

The continuation mode can be either **off** or **on**. Note that the generated code for contracts check might be different depending on the continuation mode:

- When the continuation mode is `off` an implementation is allowed to assume that an invocation to the violation handler will not return an generate code accordingly.
- When the continuation mode is `on` an implementation is not allowed to perform such assumption, which will result in different code generation.

4.2 Allowing assumption of axioms

This paper proposes that an axiom mode is added. The axiom mode can be either **off** or **on**.

- When the axiom mode is **off** an implementation is not allowed to make any assumption.
- When the axiom mode is **on** an implementation is allowed to assume the axiom.

Note that when axiom mode is **on**, an axiom condition establishes an assumption barrier. That assumption barrier prevents that the condition is not backwards propagated.

4.3 Summary of semantics

In this section a summary of the build options and their semantics is presented.

The translation is controlled by the following options:

- Build mode: `off`, `default`, and `audit`.
- Axiom mode: `off`, `on`.
- Continuation mode: `off`, `on`.

1. Build-mode=`off`, Axiom-mode=`off`, continuation=`off|on`.
   - No check is performed.
   - No assumption is made.
2. Build-mode=`off`, Axiom-mode=`on`, continuation=`off|on`.
   - No check is performed.
   - Checks with `axiom` level are assumed.
3. Build-level=`default`, Axiom-mode=`off`, continuation=`off`
   - Checks with `default` level are evaluated and assumed.
   - Code for contract checks assumes that the violation handler does not return.
   - No assumption is made.
4. Build-level=`default`, Axiom-mode=`on`, continuation=`off`
   - Checks with `default` level are evaluated and assumed.
   - Code for contract checks assumes that the violation handler does not return.

- Checks with `axiom` level are assumed.

5. Build-level=`default`, Axiom-mode=`off`, continuation=`on`
   - Checks with `default` level are evaluated but not assumed.
   - Code for contract checks assumes that the violation handler may return.
   - No assumption is made.
6. Build-level=`default`, Axiom-mode=`on`, continuation=`on`
   - Checks with `default` level are evaluated but not assumed.
   - Code for contract checks assumes that the violation handler may return.
   - Checks with `axiom` level are assumed.
7. Build-level=`audit`, Axiom-mode=`off`, continuation=`off`
   - Checks with `audit` level are evaluated and assumed.
   - Checks with `default` level are evaluated and assumed.
   - Code for contract checks assumes that the violation handler does not return.
   - No assumption is made.
8. Build-level=`audit`, Axiom-mode=`on`, continuation=`off`
   - Checks with `audit` level are evaluated and assumed.
   - Checks with `default` level are evaluated and assumed.
   - Code for contract checks assumes that the violation handler does not return.
   - Checks with `axiom` level are assumed.
9. Build-level=`audit`, Axiom-mode=`off`, continuation=`on`
   - Checks with `audit` level are evaluated but not assumed.
   - Checks with `default` level are evaluated but not assumed.
   - Code for contract checks assumes that the violation handler may return.
   - No assumption is made.
10. Build-level=`audit`, Axiom-mode=`on`, continuation=`on`
    - Checks with `audit` level are evaluated but not assumed.
    - Checks with `default` level are evaluated but not assumed.
    - Code for contract checks assumes that the violation handler may return.
    - Checks with `axiom` level are assumed.

5 Some questions

Why do we add the axiom build mode? Contracts with levels `default` and `audit` are assumed only if they have been checked. That is pure consequence of their evaluation. However, for contracts with level axiom some systems may want to take the option to still assume them when other checks are disabled (new proposed axiom mode on) while in other systems the policy may be to avoid assuming any axiom when other checks are enabled (new proposed axiom mode off).

Would it make sense not to assume axioms when other checks are enabled? An axiom is expected to be used for checks that are always true and do not need to be checked. Usual practice should be enable assumptions on axioms. However, in some debug modes developers might want to avoid those assumptions. That is obtained by disabling axioms with axiom mode set to off.

Why not controlling individual semantics for each contract level? That would lead to some combinations that may be problematic or with surprising behavior. Consider for example, enabling checking for `audit` contracts, but disabling checks for `default` contracts. In other cases, the combination of choices might even lead again to the undefined behavior that we are trying hard to avoid. Even worse we might end up with the practice of needing to duplicate a predicate in more that one assertion to guarantee it in multiple build modes.

Why not more build modes? The proposed modes seem useful for a variety of use cases. They also seem enough to gain experience with the feature in C++20. After that, if needed, the catalog of build modes might be extended in C++23.

Has this been implemented? The prototype implementation at `https://github.com/arcosuc3m/clang-contracts` implements currently the new proposed semantics for `audit`, `default` and `off` as there is no specific assumption enforcement.

How should axioms be taught? Axioms should be used only for predicates that are never wrong. In fact, that is the mathematical notion of a logical axiom (*a predicate that is universally true*). Axioms are not really preconditions, postconditions, or assertions but a portable way of spelling assumptions. Note that an axiom should never be wrong because the consequence is to inject undefined behavior.

6 Proposed wording

In this section a (probably incomplete) wording is presented. This will be refined before the Kona meeting.

6.1 Part I: Avoiding undefined behavior

In section **[dcl.attr.contract.check]/4**, edit as follows:

> 4. ~~During constant expression evaluation (7.7), only predicates of checked contracts are evaluated. In other contexts, it is unspecified whether the predicate for a contract that is not checked under the current build level is evaluated; if the predicate of such a contract would evaluate to false, the behavior is undefined.~~ Only predicates of checked contracts under current build level are evaluated.

6.2 Part II: A new axiom mode

After [**dcl.attr.contract.check**]/**4**, add a new paragraph:

> 5. A translation may be performed with an *axiom mode* which can be either *off* or *on*. Unless the *axiom level* is *off* it is unspecified whether the predicate for an *axiom* contract is evaluated; if the predicate of such a contract would evaluate to false, the behavior is undefined.

Acknowledgments

I would like to express our thanks to John Lakos, Gabriel Dos Reis, Bjarne Stroustrup, Ville Voutilainen for all the feedback provided and interesting discussions with alternate points of view. Herb Sutter helped to improve explanations on the effects of undefined behavior Richard Smith and Jens Maurer provided initial clarifications on the effect of existing wording as well as additiona feedback.

The work from J. Daniel Garcia is partially funded by the Spanish Ministry of Economy and Competitiveness through project grant TIN2016-79637-P (BigHPC – Towards Unification of HPC and Big Data Paradigms) and the European Commission through grant No. 801091 (ASPIDE – Exascale programmIng models for extreme data processing).

References

[1] Gabriel dos Reis, J. Daniel Garcia, John Lakos, Alisdair Meredith, Nathan Myers, and Bjarne Stroustrup. Support for contract based programming in C++. Working paper p0542r5, ISO/IEC JTC1/SC22/WG21, June 2018.

[2] Ville Voutilainen. UB in contract violations. Working paper p1321r0, ISO/IEC JTC1/SC22/WG21, October 2018.