Number P1680R0
Date 2019-06-17
Audience EWG
Authors Andrew Sutton
(asutton@lock3software.com)
Jeff Chapman
(jchapman@lock3software.com)

Implementing Contracts in GCC

1 Overview

This paper reports on our implementation experience with Contracts as specified in the current Working Draft and various extensions (i.e., P1332R0, P1429R0, P1290R1). Our implementation takes P1332R0 (Contract Checking in C++: A (long-term) Road Map) as the basis for associating semantics with contracts because it provides the most general framework for doing so; this gives us the ability to experiment with fine-grain assignment of checking and assumption semantics for contracts.

We also relaxed all restrictions on the placement of contracts on function and member function declarations. This allows contracts to be added to a declaration at any point in a program, with some exceptions, which we explain below.

The implementation is hosted online at https://gitlab.com/lock3/gcc-new. The implementation lives in the branch `contracts`. This compiler can also be used online through Compiler Explorer (https://godbolt.org/) as x86-64 gcc (contracts).

2 Build modes

The model prescribed by P1332R0 provides a rich facility for configuring the behavior of contracts in a translation unit. However—and importantly—the configuration of the build mode can be controlled using only the two options specified in the current WD: the build level and continuation mode using the options:

```
-fcontract-build-level=<off|default|audit>
-fcontinuation-mode=<on|off>
```

We expect the former to be the most commonly used option for control and the latter to be used in test suites that include checks against contracts. These are described in more detail in subsequent sections. The remainder of this section describes the implementation of P1332R0, which we use as the basis for supporting those (and many other) configurations.

2.1 Contract mode

In the P1332R0 model, the configuration or mode of each contract is determined by either:

- a literal specification of behavior, or
- a combination of *contract level* and its *contract role*.

The literal semantics of a contract are one of:

- *ignore* – The contract does not affect the behavior of the program.
- *assume* – The condition may be used for optimization.
- *never continue* – The program terminates if the contract is not satisfied.
- *maybe continue* – A user-defined violation handler may terminate the program.
- *always continue* – The program continues even if the contract is not satisfied.

A contract level specifies whether the contract will be checked at runtime and constant expression evaluation time. The level generally denotes the computational cost of evaluating the condition of a contract. Contract levels are *default* (for cheaply computed conditions), *audit* (for costly conditions), and *axiom* (for undecidable conditions).

The *contract role* is a named mapping of each contract level to concrete semantics. This allows contracts in a translation unit to be nominally related so their semantics can be controlled as a group, rather than individually or globally. By default, contracts are assigned to the *default* role. For example, our default mapping for the default role is:

```
default=never, audit=ignore, axiom=ignore
```

However, in the current WD, the default role be:

```
default=never, audit=ignore, axiom=assume
```

In this configuration axioms can be used for optimization since their non-satisfaction may lead to undefined behavior.

2.1.1 Implementation

Conceptually, the model defines a table that maps role names to the assignment of semantics for a level. The columns of the table determine the concrete semantics for each of the different contract levels. The rows of the table define specific roles. Note that in this model, the build levels off, default, and audit can also be considered to be roles. This is shown in Table 1.

ROLES/LEVELS	DEFAULT	AUDIT	AXIOM
OFF	ignore	ignore	ignore
DEFAULT	never	ignore	ignore
AUDIT	never	never	ignore
REVIEW	never	ignore	ignore

Table 1. Concrete semantics for a specific configuration.

When a contract is encountered in source, we compute its concrete semantics, either from the literal specification or via lookup in the role table.

In practice, we only maintain a single role for the basic behavior of contracts; the first three rows are collapsed into a single `default` role that can be controlled and modified by global configuration options for the build level, continuation mode, and assumption model (P1290R0).

2.1.2 Configuring roles

New roles can be added to this table using the option:

```
-fcontract-role=<id>:<mapping>
```

where `<id>` is the name of the role and `<mapping>` is a comma-delimited `<semantic>`s (as above). Each semantic in the list corresponds to the levels `default`, `audit`, and `axiom` in that order. For example, the review role described in P1332R0 can be specified as:

```
-fcontract-role=review:never,ignore,ignore
```

If an unknown role is found in a program, we assign it semantics from the default role. We are planning an opt-in warning to diagnose unknown roles, but this will be disabled by default.

Behaviors of the default role can also be explicitly changed using the option:

```
-fcontract-semantic=<level>:<semantic>
```

where `<level>` is one of `default`, `audit`, or `axiom` and `<semantic>` is one of the (abbreviated) literal semantics `ignore`, `assume`, `never`, `maybe`, or `always`. For example, changing our default role to the WD role requires this compiler option:

```
-fcontract-semantic=axiom:assume
```

We added this model to support the a more granular method of configuring semantics.

2.1.3 Build level

The build-level option controls the semantics of the default role.

```
-fcontract-build-level=<off|default|audit>
```

Conceptually, this model selects the row in Table 1 to use as the default role, as if the default were simply a pointer to that role in the table. In practice, we simply overwrite the concrete semantics of the role as follows:

- `off` is equivalent to `-fcontract-role=default:ignore,ignore,ignore`
- default is equivalent to `-fcontract-role=default:never,ignore,ignore`
- audit is equivalent to `-fcontract-role=default:never,never,ignore`

2.1.4 Continuation mode

The continuation mode is controlled by the option:

```
-fcontract-continuation-mode=<on|off>
```

When set to on, this modifies the default role by overwriting “never continue” semantics with “maybe continue” semantics. For example: using `-fcontract-build-level=audit` with `-fcontract-continuation-mode=on` is equivalent to `-fcontract-role=default:maybe,maybe,ignore`.

2.1.5 Assumption mode

The assumption mode is controlled by the option:

```
-fcontract-assumption-mode=<on|off>
```

When set to on, this modifies the default role's axiom level by overwriting "ignore" semantics with "assume" semantics. For example: using `-fcontract-build-level=audit` with `-fcontract-assumption-mode=on` is equivalent to `-fcontract-role=default:never,never,assume`.

This was added to support the experimental ideas presented during the San Diego meeting.

2.1.6 Contract mode

All contracts can be disabled using the option:

```
-fcontract-mode=<on|off>
```

This is on by default. When set to off, all semantics are conceptually overwritten with "ignore" semantics.

3 Contract semantics

This section explains the implementation of concrete semantics. In general, when a contract is encountered, we build a statement to test the condition. For example, suppose we encounter this condition:

```
[[assert: p != 0]];
```

We generate a checking statement with the following form:

```
if (!(p != 0))
  __on_contract_violation(<args>);
```

Here, `__on_contract_violation` is one of a set of functions that implement the concrete semantics required above (i.e., it's a placeholder here). The <args> accepted by this function include all of the information necessary to build the contract violation object and include the source location of the contract, the contract level, continuation mode, a comment, etc.

3.1 Ignored contracts

Ignored contracts are not truly ignored; they are parsed and analyzed. The condition is not emitted for code generation.

We are considering parsing ignored conditions as unevaluated operands in order to support declared-but-not-defined functions used in axioms.

3.2 Assumed contracts

We generate code for assumed contracts to make them available to the optimizer. For example, given a simple assertion:

```
[[assert assume: p != nullptr]];
```

We generate, this:

```
if (!(p != nullptr))
  __builtin_unreachable();
```

This statement is turned into an assumption, which can then be used by the optimizer.

Note that GCC has an internal assertion statement node that appears to be usable directly as an assumption. We are currently investigating that usage over this approach.

We are considering parsing ignored conditions as unevaluated operands in order to support declared-but-not-defined functions used in axioms. This should not affect static analysis, since those conditions can be interpreted symbolically.

3.3 Never continuing contracts

A contract whose semantic is "never continue" is guaranteed to terminate when the contract violation handler is invoked. Given a simple assertion:

```
[[assert never_continue: p != nullptr]];
```

The corresponding runtime code is:

```
if (!(p != nullptr))
  __on_contract_violation_never_fn(<args>)
```

The function `__on_contract_violation_never_fn` is responsible for constructing the `contract_violation` object, invoking the handler, and (finally) calling `std::terminate`. The `__on_contract_violation_never_fn` is declared `[[noreturn]]`.

Any exceptions thrown by custom violation handler will also cause the program to terminate.

3.4 Maybe continuing contracts

A contract whose semantic is "maybe continue" may terminate when the contract violation handler is invoked. Given a simple assertion:

```
[[assert maybe_continue: p != nullptr]];
```

The corresponding runtime code is:

```
if (!(p != nullptr))
  __on_contract_violation_fn(<args>)
```

The function `__on_contract_violation_fn` is responsible for constructing the `contract_violation` object and invoking the handler. Whether the function program terminates is determined by violation handler; the default handler simply returns.

Any exceptions thrown by custom violation handler will be rethrown.

3.5 Always continuing contracts

A contract whose semantic is "always continue" is guaranteed to continue when the contract violation handler is invoked. Given a simple assertion:

```
[[assert always_continue: p != nullptr]];
```

The corresponding runtime code is:

```
if (!(p != nullptr))
  __on_contract_violation_always_fn(<args>)
```

The function `__on_contract_violation_always_fn` is responsible for constructing the `contract_violation` object and invoking the handler. The function is declared as

`__attribute__((pure))` as a means of indicating that it has no side effects and also `noexcept`. For the default handler, this is true. However, a user-defined handler may subvert that guarantee and cause the program to `terminate`.

Any exceptions thrown by custom violation handler will lead to undefined behavior.

4 Contract conditions

Preconditions and postconditions impose interesting requirements on the implementation. Ideally, we'd like to be able to lift preconditions and postconditions from inside the function definition to the call site. To support this (as future work), our implementation creates two versions of each guarded function (having preconditions or postconditions): a *checked* function which calls an *unchecked* function.

For example, consider the following function:

```
int f(int n)
  [[pre: n >= 0]]
  [[post r: n == r]]
{
  return n;
}
```

Internally, we generate the moral equivalent of the following:

```
int f_unchecked(int n)
{
  return n;
}
inline int f(int n) // checked function
{
  [[assert: n >= 0]];
  int __result = f_unchecked(n);
  [[assert: ({int r = __result; n == r;})]];
  return __result;
}
```

The checked function definition is generated at the point the guarded function is defined. The checked and unchecked functions have the same properties as the original guarded function, except that the checked function definition is also inline.

The unchecked function is not available for lookup.

Note that preconditions and postconditions are not actually translated into assertions; they are lowered directly to their corresponding conditions. We show these as assertions here for the sake of brevity.

For checking postconditions, we capture a canonical result value, and then bind result identifiers to that value. We currently use the return type of the function to build these variables, although we may later choose to declare them all as `auto&&`-typed variables. The

canonical result simplifies the generation of postconditions that do not refer to the return value. For example, this function:

```
int f(int n)
  [[post r: n == r]]
  [[post: some_global == true]];
```

has the following checked function definition:

```
inline int f(int n)
{
  int __result == f_unchecked(n);
  [[assert: ({int r = __result; n == r;})]];
  [[assert: some_global == true]];
  return __result;
}
```

The scope of identifiers in postconditions is unspecified in the standard. We limit the scope of such identifiers to the end of the attribute. This allows multiple postconditions to use the same name. For example:

```
int f(int n)
  [[post r: n == r]] // #1
  [[post r: r >= 0]]; // Ok, this r is distinct from #1's r
```

Preconditions and postconditions are not always parsed at the point they appear in the program. In particular:

- Contract conditions of member functions are late-parsed, just like inline member function definitions and default member initializers.
- Parsing of postconditions is deferred until the closing brace of the function definition. This makes it possible to define postconditions on functions with deduced return types.

Note that we also split function templates in the same manner, producing two function templates. We are planning a change to our implementation where we only fork non-template code and do so as late as possible (i.e., just prior to lowering).

The implementation of postconditions is ongoing. Preconditions are reasonably well supported.

5 Generalized redeclaration

We find it desirable to relax the "first declaration must be guarded" rule in the current WD, primarily because there is not always a canonical first declaration of a function. This also enables users to insulate clients from changes to contract conditions. For generality, we also extend this rule for classes, allowing contract conditions to be added to member function declarations and definitions outside a class body (in most cases). Our general model is that contracts can be added to a function at any point before or with its definition, and that redeclarations are valid with the same contracts or without contracts. This model makes the following translation well-formed:

```
// f.hpp
```

```
int f(int n); // #1

int f(int n) [[pre: n >= 0]]; // #2

// f.cpp
int f(int n) // #3
{
  return n;
}
```

Our implementation of this generalization is not complicated and works today. When we encounter #3, we a) declare the unchecked function if required, b) define #3 as the unchecked function, and c) define #1 as the checked function.

In other words, we end up with a program that looks like this:

```
// f.hpp
int f(int n); // #1

int f(int n) [[pre: n >= 0]]; // #2

// f.cpp
int f_unchecked(int n) // #3a
{
  return n;
}

inline int f(int n) // #3b
{
  [[assert: n >= 0]];
  int __result == f_unchecked(n);
  return __result;
}
```

Here, #3b is the synthesized checked function definition, #2 is the same declaration as before; it is a redeclaration of #1. The definition in #3a is adjusted to define the unchecked function.

Note that we could also add preconditions to #3 in the original example and the translation would be the same.

Note that any use of an unguarded declaration before a guarded declaration is encountered is still well-formed. Consider the following program:

```
int f(int n); // #1

int g(int n) {
  return f(n); // #2
}
```

```
int f(int n) [[pre: n >= 0]] { // #3
  return n;
}
```

At the point of the call to f at #2, we insert a call to the function declared in #1. The definition of that function is not required at the point of the call – it is only required at link time. The definition of `f` is supplied by #3. In other words, the program works as expected. The only drawback here is that we could never inline preconditions or postconditions at the call site #2; we haven't seen them yet. However, those contracts will still be checked at runtime.

Contract conditions cannot be added to a function once it has been defined.

5.1.1 Insulated contracts

In the example above, the contracts are visible to all clients of `f.hpp`, because the guarded redeclaration appears in the same header file as the unguarded first declaration. However, our approach *also* allows the guarded declaration (and/or definition) to appear only in the `f.cpp` file. This allows users to fully insulate clients from changes to preconditions or postconditions.

For example, if we start with this:

```
 // f.hpp
int f(int n); // #1

// f.cpp
int f(int n) [[pre: n >= 0]]; // #2

int f(int n) // #3
{
  return n;
}
```

The equivalent program would be similar to the one above.

```
// f.hpp
int f_unchecked(int n); // #0

// f.cpp
int f(int n) [[pre: n >= 0]]; // #2

int f_unchecked(int n) // #3
{
  return n;
}

inline int f(int n) // #4
{
  [[assert: n >= 0]];
  int __result == f_unchecked(n);
```

```
  return __result;
}
```

For clients of f.hpp, the story is slightly different:

```
// g.cpp
#include “f.hpp”

int g(int n) {
  return f(n); // #1
}
```

When `f.hpp` is processed, it provides only an unguarded declaration of `f`. However, this is still a valid redeclaration of the (non-locally) guarded `f`, and the call at #1 will call the checked version defined in the translation unit for `f.cpp`. Note that changes to the contracts in `f.cpp` do not require recompilation of `g.cpp`.

5.1.2 Member functions

This feature also works for member functions:

```
struct S {
  int f(int n);
};

int S::f(int n) [[pre: n >= 0]];

int S::f(int n) {
  return n;
}
```

Note that the out-of-class redeclaration of `S::f` is a new feature. This is currently prohibited in the current WD.

The redeclaration rules produce the following translation.

```
struct S {
  int f_unchecked(int n);
  int f(int n);
};

int S::f(int n) [[pre: n >= 0]];

int S::f_unchecked(int n) {
  return n;
}

int S::f(int n) {
  [[assert: n>= 0]];
  int __result = f_unchecked(n);
```

```
  return __result;
}
```

5.1.3 Virtual functions

We currently require contracts on the first declaration of virtual functions since the signatures of overrides must match. Deferring the annotation of contracts can lead to conditions where we would have to retroactively "patch" multiple class definitions in order to propagate those contracts. For example, consider:

```
// b.hpp
struct S {
  virtual int f(int n); // #1
};

// b.cpp
int S::f(int n) [[pre: n >= 0]]; // #2

int S::f(int n) { // #3
  return n;
}

// c.hpp
struct C : S {
  virtual int f(int n); // #4
};

// c.cpp
int C::f(int n) { // #5
  return n + 1;
}
```

Any calls that pass directly through the base type call the checked function defined inside `b.cpp`. The definition that occurs at #5 has not seen any contracts and subsequently does not emit any contract checks.

6 Conclusion

This paper summarizes our implementation experience with Contracts as specified in the current Working Draft and various extensions (i.e., P1332R0, P1429R0, P1290R1). We've found that the mapping of contract modes to specific and concrete semantics simplifies the implementation while providing a sound framework for the addition of new contract-related features (e.g., explicit assumptions).