Doc. No.: P2737R0
Project: Programming Language - C++ (WG21)
Audience: SG21 Contracts
Author: Andrew Tomazos <andrewtomazos@gmail.com>
Date: 2022-12-04

# Proposal of Condition-centric Contracts Syntax

## Introduction

We propose and justify a concrete syntax for MVP contracts we call **condition-centric** syntax. Our proposed syntax differs from the two syntaxes (**attribute-like** and **closure-based**) outlined in the contracts working paper P2521R2.

## Example

| Attribute-like | Closure-based | **Condition-centric (this proposal)** |
|---|---|---|
| `int select(int i, int j)`<br>`  [[pre: i >= 0]]`<br>`  [[pre: j >= 0]]`<br>`  [[post r: r >= 0]]`<br>`{`<br>`  [[assert: _state >= 0]];`<br><br>`  if (_state == 0)`<br>`  return i;`<br>`  else`<br>`  return j;`<br>`}`<br>`int pre; // OK`<br>`int assert; // OK`<br>`int post; // OK` | `int select(int i, int j)`<br>`  pre{i >= 0}`<br>`  pre{j >= 0}`<br>`  post(r){r >= 0}`<br>`{`<br>`  assert{_state >= 0};`<br><br>`  if (_state == 0)`<br>`  return i;`<br>`  else`<br>`  return j;`<br>`}`<br>`int pre; // OK`<br>`int assert; // ???`<br>`int post; // OK` | `int select(int i, int j)`<br>`  precond(i >= 0)`<br>`  precond(j >= 0)`<br>`  postcond(result >= 0)`<br>`{`<br>`  incond(_state >= 0);`<br><br>`  if (_state == 0)`<br>`  return i;`<br>`  else`<br>`  return j;`<br>`}`<br>`int precond; // ERROR`<br>`int incond; // ERROR`<br>`int postcond; // ERROR` |

## Summary

We propose the following four changes to the MVP contract specification and syntax:

1. Change the term in the specification for a contract “assertion” to an “incondition”. (This differs from other proposals which refer to them as “assertions”.)

2. Full keywords `precond`, `incond` and `postcond` are reserved. (This differs from other proposals which use the spellings `pre`, `post` and `assert`, and do not reserve them as full keywords.)

3. The syntax for a contract uses parentheses to delimit the condition: `precond(expr)`, `incond(expr)`, `postcond(expr)`. (This differs from other proposals that use either double square-brackets, a colon and/or braces to delimit the contract and/or the expression).

4. A postcondition-local predefined variable `result` is introduced into the scope of a postcondition to refer to the return value of the function. (This differs from other proposals that use a “binding syntax” where a new and potentially-unique name for the return value is declared in every postcondition)

## Inconditions

We propose coining a new term called an “incondition”. If a precondition is a condition that must be true at the entry of a function and a postcondition is a condition that must be true at the exit of a function, an incondition is a condition that must be true at a point with**in** (or **in**side) a function. The linguistic precedent for this “triplet” is demonstrated by this table:

| | order | condition |
|---|---|---|
| pre | preorder | precondition |
| in | inorder | incondition |
| post | postorder | postcondition |

Much the same as postcondition, incondition is not an English word and does not appear in the dictionary. Incondition is a new term we have developed in much the same way as postcondition was developed, by joining an existing prefix (“in-”) with an existing word (“condition”) to create a new term that matches our desired meaning.

The motivation for replacing the term “assertion” is three fold:

1. The term “assertion” is inconsistent in style with precondition and postcondition. They are not apparently related to one another. For example, conditions are said to be violated whereas assertions are said to fail. (It is important that it is apparent that the three things are related so that information about the language feature will be known by the user to appertain to all three things. For example, it is expected that there will be multiple translation modes that impact

contract semantics, so it will be necessary to understand which things this translation mode selection impacts.)

2. We already have a heavily-used “assertion” in C++. That is C assert (<cassert>). Also `static_assert`, and there are numerous user libraries that offer a facility they call an assertion. If a contract assertion is called an assertion, then it will be ambiguous in discussions with an assertion created with C assert (or with static_assert, or user-library assertions), without further qualification of which kind of C++ assertions is meant.

3. assert is not claimable as a keyword, or at least it very nearly conflicts with the function-like macro of the same name in the C library.

The term incondition addresses these three points in the following fashion:

1. The term “incondition” has a consistent style with precondition and postcondition. Because all three are “something-conditions”, it becomes apparent that they are part of the same language feature, and creates a correct expectation that they have the same syntax and semantics. That they are three kinds of the same thing. That they are special cases of the same general feature.

2. The term “incondition” cannot be confused with an “assertion” created with C assert, static_assert or an assertion from a user-library, and does not require further qualification to disambiguate.

3. `incond` is claimable as a full keyword, and has little to no existing use as an identifier or macro.

## Keywords

We propose reserving three full keywords for the contracts feature spelled **`precond`**, **`incond`** and **`postcond`** to represent preconditions, inconditions and postconditions, respectively.

The spelling “cond” is a very common shortening of the word “condition” used by programmers.

These spellings have on average only a thousand hits in ACTCD19 (codesearch.isocpp.org) so these spellings are available to claim as full keywords.

## Function Result Syntax

For referring to the result of the function in a postcondition we propose introducing a postcondition-local predefined variable into the scope of postconditions spelled **`result`**. This method of introducing a non-keyword name to represent something has a precedent in existing function-local predefined variables, more generally the notion of introducing an implicit spelling

for something has a precedent in the `this` keyword to represent the implicit object parameter of a member function.

## Function-like Keyword Syntax

For the general form of a contracts syntax we propose `keyword ( expression )`. That is:

```
precond ( expression )
incond ( expression )
postcond ( expression )
```

This kind of function-like use of a keyword to introduce a parenthesized sequence of tokens has numerous precedents in the language:

```
alignas ( … )
alignof ( … )
decltype ( … )
for ( … )
if ( … )
switch ( … )
sizeof ( … )
static_assert ( … )
while ( … )
```

Although there are also precedents for `keyword { … }` in use, we feel that:

1. The precedents of if ( expression ), while ( expression ) and static_assert ( expression ) are particularly strong, given they also delimit condition expressions (that are even converted to bool) specifically.
2. It should also be noted that in real-world code parenthesis are more commonly used to delimit expressions than statements, whereas braces are more commonly used to delimit statements than expressions.
3. A parenthesized expression is a primary expression, taken from the mathematical notation to group operations together.

## Grammar / Specification

Add **precond, incond** and **postcond** to C++ keywords list.

```
precondition:
    precond ( expression )

incondition:
```

```
    incond ( expression )

postcondition:
    postcond ( expression )

For each value returning function F, a postcondition-local predefined
variable named result is introduced into the scope of the expression of each
of the postconditions of F.  It holds the return value of the function.

(Postcondition-local predefined variables will be specified in a similar
fashion to function-local predefined variables. The remainder of the grammar
and specification is the same as the other proposals.)
```

# FAQ

**Why do you think implicitly introducing a name for the result of the function in a postcondition, is better than the binding syntax of previous contracts proposals?**

The design decision here is between:

A) a syntax that requires explicitly declaring a name for the result in each postcondition
B) a syntax that implicitly introduces a (common) name for the return value in each postcondition

We feel that A creates an unnecessary DRY violation, whereas B does not.  We see no significant benefits to A over B.  We therefore propose B.

**What about structured binding?  Without a "binding syntax" how do you destructure the return value?**

1. We think contracts are viable without the ability to easily apply a structured binding to the return value of the function in a postcondition, so regardless of syntax, structured binding does not belong in the MVP.

2. With any of the syntaxes you can always write an auxiliary bool-returning function (or a lambda) to be used in the condition expression.  Within that function, you can use a structured binding or any of the other kinds of statements or declarations.

3. A possible forward-compatible future extension of our syntax is to add an init-statement to the condition, with the same syntax as an if statement today.  Within that init-statement you can create a structured binding of the return value.  We think this is a

better approach than providing a bespoke binding syntax, because it reuses existing syntax and user knowledge, keeping the language overall simpler (less syntax to learn).

**Why do you think a non-keyword name is better than a full keyword for the result of the function?**

We do not. All things being equal, we would have preferred a keyword to a non-keyword name. The three alternatives we considered were:

A) A full keyword `return`
B) A full keyword `resval`, short for "**res**ult **val**ue".
C) A non-keyword name `result`

Other alternatives were either inviable or so clearly worse than either A, B or C they are not worth mentioning.

Our analysis concluded:

1. An ambiguity can arise with A between an expression involving the return value and a return statement (perhaps part of a lambda expression in a postcondition). This alternative would require a disambiguation rule. B and C do not require a disambiguation rule. While this ambiguity is unlikely to come up in practice, it is still a factor.
2. The spelling of B is significantly worse than A and C readability-wise. A and C are short and complete English words, B is not.
3. Full keywords (A and B) are preferable to a non-keyword (C) because full keywords are simpler, conceptually and implementation-wise. (paraphrasing Bjarne)

Upon weighing these three tradeoffs we came to the conclusion that C was the best of the three alternatives.

**What about if the name `result` is used in an outer scope?**

If result is used in an outer scope it can be referred to with a qualified name:

```
int result;
namespace N { int result; }
class C {
    int result;
    int f()
        postcond (result + ::result + N::result + this->result
          == 42);
```

If result is used as a function parameter it can be changed (parameter names are not significant):

```
int f(int result_in)
   postcond(result == result_in);
```

**Why are you proposing full keywords rather than context-sensitive keywords?**

The design decision on the spellings precond, incond and postcond was made first, prior to the design decision of whether they are full keywords or context-sensitive keywords.

We prefer the spellings precond, incond and postcond to pre, post and assert because:

1. The former are more readable. For example it's much easier to intuit that precond is short for precondition than it is to intuit that pre is short for precondition (because cond is a common shortening of condition). pre could be short for something else, like prepare or preview, or any other pre words. post is a commonly used identifier to mean the English word "post" as in "post this message". The desire to move away from assert has already been motivated.

2. Because all three spellings are "something-cond" it is clear that they are part of the same language feature, with the same motivation given in point 2 of the Inconditions section.

Once the spellings were selected we then made the design decision of whether to make them full keywords or context-sensitive keywords. Paraphrasing Bjarne again, keywords are simpler conceptually and implementation-wise. Further to this thought, we claim that if a spelling is available as a full keyword it should be claimed as such. Full keywords (if available) should be preferred to context-sensitive keywords.

Full keywords have another added benefit to syntax extensibility. That is, it is easier to extend a syntax that uses full keywords, because full keywords are not ambiguous with a non-keyword name (identifier) of the same spelling in the same position - whereas context-sensitive keywords are.

**If we added a feature post-MVP that allowed for capturing of program state at entry of the function for use in a postcondition (such as non-const function parameters), how would you extend the condition-centric contracts syntax to accommodate?**

First we should observe that the existence or semantics of such an extension are not yet known. We may cover that use case in some way that we haven't thought of yet, or perhaps even not at all. We think it's therefore premature to propose a concrete syntax for anything other than checking for forward-compatibility and extensibility.

What we would point out is that as the contracts are introduced by full keywords in the proposed condition-centric syntax and terminated by a parenthese-delimited expression, the syntax is

wide open for extension and would be forward compatible with almost any added sequence of tokens between the keyword and the opening parenthesis of the condition.  For this reason we can prove that it is at least as extensible as the other two syntaxes (attribute-like and closure-based) in this regard.

Further as precond, incond and postcond are full keywords, we could also develop additional declarations and/or statements that used these keywords to introduce new statements, in the event that we wanted to extend the syntax with additional namespace-scope, class-scope or function-scope constructs.

## References

P2521R2 Contract support — Working Paper
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/p2521r2.html