Doc. No.: D2737R0
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. Our syntax competes with the two syntaxes outlined in the contracts working paper P2521R2.

Inconditions

The first move is that we coin 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

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

1. The term “assertion” is inconsistent in style with precondition and postcondition.

2. We already have a heavily-used “assertion” in C++. That is C assert (<cassert>). Also `static_assert`.

3. assert is not claimable as a keyword.

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

1. The term “incondition” has a consistent style with precondition and postcondition. 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.

2. The term “incondition” cannot be confused with an assertion created with C assert or static_assert.

3. See below.

Keywords

Next, 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

Next, for referring to the result of the function in a postcondition we propose introducing a non-keyword name into the scope of postconditions spelled **`result`**. This method of introducing a non-keyword name to represent something has a precedent in `` `__func__` ``, 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 shape of a contracts syntax we propose `keyword ( condition )`. That is:

```
precond ( condition )
incond ( condition )
postcond ( condition )
```

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 ( condition )`, `while ( condition )` and `static_assert ( condition )` are particularly strong, given they also delimit conditions 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.

Example

Putting it all together we have:

```
int select(int i, int j)
  precond(i >= 0)
  precond(j >= 0)
  postcond(result >= 0)
{
  incond(_state >= 0);

  if (_state == 0) return i;
  else             return j;
}
```

Grammar / Specification

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

```
precondition:
    precond ( condition )

incondition:
    incond ( condition )

postcondition:
```

```
    postcond ( condition )
```

For a value returning function, the name `result` is introduced into the scope of the condition of a postcondition. It refers to the return value of the function.

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.

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. A creates an ambiguity with a return statement, requiring a disambiguation rule. B and C do not. 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. A and B are short and complete English words, C 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);
```

References

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