TECHNICAL SPECIFICATION

ISO/IEC TS 22277

First edition
2017-11

Technical Specification — C++ Extensions for Coroutines

Langages de programmation — Extensions C++ pour les Coroutines

Reference number
ISO/IEC TS 22277:2017(E)

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see the following URL: www.iso.org/iso/foreword.html.

This document was prepared by Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Technical Specification — C++ Extensions for Coroutines

1 Scope [intro.scope]

1 This document describes extensions to the C++ Programming Language (Clause 2) that enable definition of coroutines. These extensions include new syntactic forms and modifications to existing language semantics.

2 The International Standard, ISO/IEC 14882:2014, provides important context and specification for this document. This document is written as a set of changes against that specification. Instructions to modify or add paragraphs are written as explicit instructions. Modifications made directly to existing text from the International Standard use underlining to represent added text and ~~strikethrough~~ to represent deleted text.

2 Normative references [intro.refs]

1 The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

(1.1) — ISO/IEC 14882:2014, *Programming Languages – C++*

ISO/IEC 14882:2014 is hereafter called the *C++ Standard*. Beginning with Clause 5, all clause and subclause numbers, titles, and symbolic references in [brackets] refer to the corresponding elements of the C++ Standard. Clauses 1 through 4 of this document are unrelated to the similarly-numbered clauses and subclauses of the C++ Standard.

3 Terms and definitions [intro.defs]

No terms and definitions are listed in this document. ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at http://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

4 General [intro]

4.1 Implementation compliance [intro.compliance]

Conformance requirements for this specification shall be the same as those defined in subclause 1.4 of the C++ Standard. [*Note:* Conformance is defined in terms of the behavior of programs. — *end note*]

4.2 Feature testing [intro.features]

An implementation that provides support for this document shall define the feature test macro in Table 1.

Table 1 — Feature-test macro

Name	Value	Header
`__cpp_coroutines`	201707	*predeclared*

4.3 Program execution [intro.execution]

In subclause 1.9 of the C++ Standard modify paragraph 7 to read:

7 An instance of each object with automatic storage duration (3.7.3) is associated with each entry into its block. Such an object exists and retains its last-stored value during the execution of the block and while the block is suspended (by a call of a function, suspension of a coroutine (5.3.8), or receipt of a signal).

4.4 Lexical conventions [lex]

In subclause 2.12 of the C++ Standard add the keywords `co_await`, `co_yield`, and `co_return` to Table 4 "Keywords".

4.5 Basic concepts [basic]

In subclause 3.6.1 of the C++ Standard add underlined text to paragraph 3.

3 The function `main` shall not be used within a program. The linkage (3.5) of `main` is implementation-defined. A program that defines `main` as deleted or that declares `main` to be `inline, static`, or `constexpr` is ill-formed. The function `main` shall not be a coroutine (8.4.4). The name `main` is not otherwise reserved. [*Example:* member functions, classes, and enumerations can be called `main`, as can entities in other namespaces. — *end example*]

4.6 Dynamic storage duration [basic.stc.dynamic]

In subclause 3.7.4.1 of the C++ Standard modify paragraph 4 as follows:

4 A global allocation function is only called as the result of a new expression (5.3.4), ~~or~~ called directly using the function call syntax (5.2.2), called indirectly to allocate storage for a coroutine frame (8.4.4), or called indirectly through calls to the functions in the C++ standard library. [*Note:* In particular, a global allocation function is not called to allocate storage for objects with static storage duration (3.7.1), for objects or references with thread storage duration (3.7.2), for objects of type `std::type_info` (5.2.8), or for an exception object (15.1). — *end note*]

5 Expressions [expr]

5.3 Unary expressions [expr.unary]

Add *await-expression* to the grammar production *unary-expression*:

unary-expression:
postfix-expression
`++` *cast-expression*
`--` *cast-expression*
await-expression
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof (` *type-id* `)`
`sizeof ... (` *identifier* `)`
`alignof (` *type-id* `)`
noexcept-expression
new-expression
delete-expression

5.3.8 Await [expr.await]

Add this subclause to 5.3:

1 The `co_await` expression is used to suspend evaluation of a coroutine (8.4.4) while awaiting completion of the computation represented by the operand expression.

await-expression:
`co_await` *cast-expression*

2 An *await-expression* shall appear only in a potentially-evaluated expression within the *compound-statement* of a *function-body* outside of a *handler* (Clause 15). In a *declaration-statement* or in the *simple-declaration* (if any) of a *for-init-statement*, an *await-expression* shall appear only in an *initializer* of that *declaration-statement* or *simple-declaration*. An *await-expression* shall not appear in a default argument (8.3.6). A context within a function where an *await-expression* can appear is called a *suspension context* of the function.

3 Evaluation of an *await-expression* involves the following auxiliary types, expressions, and objects:

(3.1) — p is an lvalue naming the promise object (8.4.4) of the enclosing coroutine and P is the type of that object.

(3.2) — a is the *cast-expression* if the *await-expression* was implicitly produced by a *yield-expression* (5.20), an initial suspend point, or a final suspend point (8.4.4). Otherwise, the *unqualified-id* `await_transform` is looked up within the scope of P by class member access lookup (3.4.5), and if this lookup finds at least one declaration, then a is `p.await_transform(`*cast-expression*`);` otherwise, a is the *cast-expression*.

(3.3) — o is determined by enumerating the applicable `operator co_await` functions for an argument a (13.3.1.2), and choosing the best one through overload resolution (13.3). If overload resolution is ambiguous, the program is ill-formed. If no viable functions are found, o is a. Otherwise, o is a call to the selected function.

(3.4) — e is a temporary object copy-initialized from o if o is a prvalue; otherwise e is an lvalue referring to the result of evaluating o.

(3.5) — h is an object of type `std::experimental::coroutine_handle<`P`>` referring to the enclosing coroutine.

(3.6) — *await-ready* is the expression e`.await_ready()`, contextually converted to `bool`.

(3.7) — *await-suspend* is the expression e`.await_suspend(`h`)`, which shall be a prvalue of type `void` or `bool`.

(3.8) — *await-resume* is the expression e`.await_resume()`.

4 The *await-expression* has the same type and value category as the *await-resume* expression.

5 The *await-expression* evaluates the *await-ready* expression, then:

(5.1) — If the result is `false`, the coroutine is considered suspended. Then, the *await-suspend* expression is evaluated. If that expression has type `bool` and evaluates to `false`, the coroutine is resumed. If that expression exits via an exception, the exception is caught, the coroutine is resumed, and the exception is immediately re-thrown (15.1). Otherwise, control flow returns to the current caller or resumer (8.4.4) without exiting any scopes (6.6).

(5.2) — If the result is `true`, or when the coroutine is resumed, the *await-resume* expression is evaluated, and its result is the result of the *await-expression*.

6 [*Example:*

```
template <typename T>
struct my_future {
  ...
  bool await_ready();
  void await_suspend(std::experimental::coroutine_handle<>);
  T await_resume();
};

template <class Rep, class Period>
auto operator co_await(std::chrono::duration<Rep, Period> d) {
  struct awaiter {
    std::chrono::system_clock::duration duration;
    ...
    awaiter(std::chrono::system_clock::duration d) : duration(d){}
    bool await_ready() const { return duration.count() <= 0; }
    void await_resume() {}
    void await_suspend(std::experimental::coroutine_handle<> h){...}
  };
  return awaiter{d};
}

using namespace std::chrono;

my_future<int> h();

my_future<void> g() {
  std::cout << "just about go to sleep...\n";
  co_await 10ms;
  std::cout << "resumed\n";
  co_await h();
}

auto f(int x = co_await h()); // error: await-expression outside of function suspension context
int a[] = { co_await h() }; // error: await-expression outside of function suspension context
```

— *end example*]

5.17 Assignment and compound assignment operators [expr.ass]

Add *yield-expression* to the grammar production *assignment-expression*.

assignment-expression:
conditional-expression
logical-or-expression assignment-operator initializer-clause
throw-expression
yield-expression

5.19 Constant expressions [expr.const]

Add bullets prohibiting *await-expression* and *yield-expression* to paragraph 2.

— an *await-expression* (5.3.8);
— a *yield-expression* (5.20);

5.20 Yield [expr.yield]

Add a new subclause to Clause 5.

yield-expression:
`co_yield` *assignment-expression*
`co_yield` *braced-init-list*

1 A *yield-expression* shall appear only within a suspension context of a function (5.3.8). Let e be the operand of the *yield-expression* and p be an lvalue naming the promise object of the enclosing coroutine (8.4.4), then the *yield-expression* is equivalent to the expression `co_await` p`.yield_value(`e`)`.

[*Example:*

```
template <typename T>
struct my_generator {
  struct promise_type {
    T current_value;
    ...
    auto yield_value(T v) {
      current_value = std::move(v);
      return std::experimental::suspend_always{};
    }
  };
  struct iterator { ... };
  iterator begin();
  iterator end();
};

my_generator<pair<int,int>> g1() {
  for (int i = i; i < 10; ++i) co_yield {i,i};
}
my_generator<pair<int,int>> g2() {
  for (int i = i; i < 10; ++i) co_yield make_pair(i,i);
}

auto f(int x = co_yield 5); // error: yield-expression outside of function suspension context
int a[] = { co_yield 1 }; // error: yield-expression outside of function suspension context

int main() {
  auto r1 = g1();
```

```
  auto r2 = g2();
  assert(std::equal(r1.begin(), r1.end(), r2.begin(), r2.end()));
}
```

— *end example*]

6 Statements [stmt.stmt]

6.5 Iteration statements [stmt.iter]

Add the underlined text to paragraph 1.

1 Iteration statements specify looping.

iteration-statement:
`while` `(` *condition* `)` *statement*
`do` *statement* `while` `(` *expression* `)` `;`
`for` `(` *for-init-statement* *condition*$_{opt}$`;` *expression*$_{opt}$`)` *statement*
`for` `co_await`$_{opt}$ `(` *for-range-declaration* `:` *for-range-initializer* `)` *statement*

6.5.4 The range-based `for` statement [stmt.ranged]

Add the underlined text to paragraph 1.

1 For a range-based `for` statement of the form

`for` `co_await`$_{opt}$ `(` *for-range-declaration* `:` *expression* `)` *statement*

let *range-init* be equivalent to the *expression* surrounded by parentheses[1]

`(` *expression* `)`

and for a range-based `for` statement of the form

`for` `co_await`$_{opt}$ `(` *for-range-declaration* `:` *braced-init-list* `)` *statement*

let *range-init* be equivalent to the *braced-init-list*. In each case, a range-based `for` statement is equivalent to

```
{
  auto && __range = range-init;
  for ( auto __begin =  co_await_opt begin-expr,
  __end = end-expr;
  __begin != __end;
   co_await_opt ++__begin ) {
    for-range-declaration = *__begin;
    statement
  }
}
```

where `co_await` is present if and only if it appears immediately after the `for` keyword, and `__range`, `__begin`, and `__end` are variables defined for exposition only, and `_RangeT` is the type of the expression, and *begin-expr* and *end-expr* are determined as follows: ...

1) this ensures that a top-level comma operator cannot be reinterpreted as a delimiter between *init-declarator*s in the declaration of `__range`.