Resumable Functions

This document is directly related to N3970, the working draft for a Technical Specification focusing on concurrency. It proposes a number of additions to N3970.

In relation to the superseded document N3858, this paper drops all the background information and focuses on proposed edits to the Concurrency TS. For background information, the reader is referred to the superseded document.

It is hereby proposed that the following modifications to the standard be incorporated into the Concurrency Technical Specification, N3970. Section and paragraph numbers refer to the C++ standard working draft.

1 resumable

In 1.9 paragraph 15, add the underlined text:

Every evaluation in the calling function (including other function calls to non-resumable functions, as defined in 8.3.5/15) that is not otherwise specifically sequenced before or after the execution of the body of the called function is indeterminately sequenced with respect to the execution of the called function. The execution of a resumable function may appear to interleave with the calling function. When a resumable function is suspended at the await keyword, a placeholder of the eventual result is returned and the calling function continues its execution. After suspending, a resumable function may be resumed and will eventually complete its logic, at which point it executes a return statement filling in the value of the placeholder.

To footnote 9, add the underlined text:

In other words, function executions do not interleave with each other, with the exception of resumable functions, which may interleave with their caller.

In 2.11 paragraph 2, add to Table 3 – Identifiers with special meaning:

resumable

In 5.1.2 paragraph 1, add the underlined:
In 5.1.2 paragraph 4, add the underlined text:

If a lambda-expression does not include a lambda-declarator, it is as if the lambda-declarator were (). The lambda return type is auto, which is replaced by the trailing-return-type if provided and/or deduced from return statements as described in 7.1.6.4. If the lambda has the resumable specifier and no trailing-return-type is provided, the return type is future<T>, where T is the type deduced from return statements. [Example:

Auto x1 = [](int i){ return i; }; // OK: return type is int
Auto x2 = [](i, 2); // Error: deducing return type from braced-init-list
Int j;
Auto x3 = []{ return j; }; // OK: return type is int
Auto x4 = []() resumable {
    Int i = await read_stream();
    Return i; }; // OK: return type is future<int>
—end example]

In 8.0 paragraph 4, add the underlined text:

Parameters-and-qualifiers:

( parameter-declaration-clause ) attribute-specifier-seq, cv-qualifier-seq, ref-qualifier, resumable-specification, exception-specification

In 8.3.5 paragraph 1, add the underlined text:

D1 ( parameter-declaration-clause ) cv-qualifier-seq

ref-qualifier, resumable-specification, exception-specification

In 8.3.5 paragraph 2, add the underlined text:

D1 ( parameter-declaration-clause ) cv-qualifier-seq

ref-qualifier, resumable-specification, exception-specification, attribute-specifier-seq

At the end of 8.3.5 paragraph 2, add the following:

The resumable-specification is not a part of the function type.

In 8.4.1 paragraph 2, add the underlined text:

D1 ( parameter-declaration-clause ) cv-qualifier-seq

ref-qualifier, resumable-specification, exception-specification, attribute-specifier-seq, trailing-return-type

In 8.3.5, add a paragraph 15

15. The function specified with a resumable specifier is a resumable function:

resumable-specification:
resumable

- A resumable function is a function that returns a placeholder for an eventually available value.
- It may be observed by its caller to return without filling the placeholder with a value.
- If the resumable function terminates, it will fill the placeholder with either a value or an exception.
- Some side-effects of the resumable function may be delayed until after its return.
- The caller of this function can resume its work without waiting for the resumable function to finish.

[Example:]

```cpp
int work1(int value);
void f(int value) {
    future<int> f = g(value);
    work2();
}

future<int> g(value) resumable {
    result = await std::async([=] {return work1(value)); //
    return result;
}
```

- end example]

- If a declaration contains a resumable-specification then every subsequent redeclaration shall also contain a resumable-specification.
- A resumable-specification shall not appear in a typedef declaration or alias-declaration.
- A function declaration with the resumable specifier must return future<T> or shared_future<T>.
- The parameter-declaration-clause of a resumable function shall not terminate with an ellipsis.
- The result returned from a function when it first suspends is a placeholder for the eventual result: i.e. a future<T> representing the return value of a function that eventually computes a value of type T.
- The parameter-declaration-clause may terminate with a function parameter pack.
- The resumable keyword is only a reserved keyword in the resumable-specification position of a function’s declaration. It has no special meaning if used elsewhere.

2 await

In 2.12, add to Table 4 – Keywords:

```cpp
await
```

In 5.3 paragraph 1, add the underlined text:

Expressions with unary operators group right-to-left.
unary-expression:
    postfix-expression
    ++ cast-expression
    -- cast-expression
    unary-operator cast-expression
    sizeof unary-expression
    sizeof ( type-id )
    sizeof ... ( identifier )
    alignof ( type-id )
    noexcept-expression
    new-expression
    delete-expression

unary-operator: one of
    * & + - ! ~ await

To 5.3 ‘Unary expressions,’ add a subsection 5.3.8:

5.3.8 await unary operator

A unary operator expression of the form:

    await cast-expression

1. The await operator is only valid within resumable functions [8.3.5] and decltype() expressions.
2. When the await operator is applied to an operand in a resumable function, the execution of the resumable function is suspended until the operand completes.
3. The cast-expression shall be of class type future<T> or shared_future<T> or shall be implicitly convertible to future<T> or shared_future<T>.
4. await is globally reserved but meaningful only within the body of a function or within the argument of a decltype() expression.
5. The await operator shall not be invoked if there is an exception being handled (15.3)\(^1\).
6. The await operator shall not be executed while a lock [30.4.2] is being held.
7. The result of await is of type T, where T is the return type of the get function of the future or shared_future object. If T is void, then the await expression cannot be the operand of another expression.

3 return

In 6.6.3, add a paragraph 4:

4 Within a resumable function declared to return future<T> or shared_future<T>, where T may be void, any return statement shall be treated as if the function were declared to return a value of type T.

---

\(^1\) The motivation for this is to avoid interfering with existing exception propagation mechanisms, as they may be significantly (and negatively so) impacted should await be allowed to occur within exception handlers.