Doc No: N3650
Supersedes: N3564
Date: 2013-05-02
Reply to: Niklas Gustafsson < niklas.gustafsson@microsoft.com>
Deon Brewis < deonb@microsoft.com>
Herb Sutter <hsutter@microsoft.com>
Sana Mithani <sanam@microsoft.com>

Resumable Functions

While presenting a proposal that can be adopted or rejected in isolation, this document is related to N3634. The reader is advised to read both as a unit and to consider how the two build on each other for synergy. Reading them in their assigned numeric order is strongly advised.

1. The Problem

The direct motivation for introducing resumable functions is the increasing importance of efficiently handling I/O in its various forms and the complexities programmers are faced with using existing language features and existing libraries.

The motivation for a standard representation of asynchronous operations is outlined in N3558 and won't be repeated here. The need for language support for resumable functions arises from the inherent limitations of the library-based solution described in that proposal.

Taking a purely library-based approach to composition of asynchronous operations means sacrificing usability and versatility: the development of an asynchronous algorithm usually starts with a synchronous, imperative expression of it, which is then manually translated into an asynchronous equivalent. This process is quite complex, akin to the reformulation of an imperative algorithm in a pure functional language, and the resulting code may be difficult to read.

A pure library-based approach leads to object lifetime management complexities and thus a different way of designing the objects that are to be used by and with asynchronous operation. An even bigger problem with solutions that avoid language support for asynchrony is the lack of ability to compose asynchronous operations using the rich variety of traditional control-flow primitives.

Consider this example, using the modified version of std::future<T> introduced in N3558:

```
future<int> f(shared_ptr<stream> str)
{
    shared_ptr<vector<char>> buf = ...;
    return str->read(512, buf)
      .then([](future<int> op)              // lambda 1
            {
              return op.get() + 11;
            });
}

future<void> g()
{
    shared_ptr<stream> s = ...;
    return f(s).then([s](future<int> op)  // lambda 2
          {
              s->close();
          });
}
```

When g() is activated, it creates its stream object and passes it to f, which calls the read() function, attaches a continuation (lambda 1) to its result, and then returns the result of the then() member function call. After the call to f() returns, g attaches a continuation (lambda 2) to the result, after which it returns to its caller.

When the read operation finishes, lambda 1 will be invoked from some context and its logic executed, resulting in the operation returned from f() completing. This, in turn, results in lambda 2 being invoked and its logic executed. If you were to set a breakpoint in either of the lambdas, you would get very little context or information on how you got there, and a debugger would be hard-pressed to make up for the lack of information.

To make matters worse, the above code does not consider that the futures returned by read() and f() may already be completed, making the attachment of a continuation lambda unnecessary and expensive. To squeeze out all the performance possible, the code will wind up being quite complex.

In order to properly account for the fact that the lifetime of local objects passed to asynchronous operations (s, buf) is different from the scope in which they are declared the programmer has to allocate them on the heap and find some means of managing them (using a shared_ptr<> is often sufficient).

Contrast this with how the same algorithm, just as efficient and asynchronous, would look when relying on resumable functions:

```
    future<int> f(stream str) async
    {
        shared_ptr<vector<char>> buf = ...;
        int count = await str.read(512, buf);
        return count + 11;
    }

    future<void> g() async
    {
        stream s = ...;
        int pls11 = await f(s);
        s.close();
    }
```

Not only is this simpler, it is more or less identical to a synchronous formulation of the same algorithm. Note, in particular, that there is no need to manage the lifetime of locally declared objects by allocating them on the heap: the compiler takes care of the lifetimes, allowing the programmer to write code that looks almost identical to synchronous code.

The library-based approach gets even more complicated when our example includes control-flow such as conditional evaluation and/or loops. The language-based approach allows control-flow to remain identical to the synchronous formulation, including the use of try-catch blocks and non-reducible constructs such as goto and break.

The following example illustrates this.

While iterative composition is not covered in N3558, it is nevertheless expected that libraries will provide higher-order compositional constructs to mimic the behavior of such things as loops and conditional expressions/statements. With the help of a do_while() construct (not described in N3558), we get this code:

```
       auto write =
         [&buf](future<int> size) -> future<bool>
         {
           return streamW.write(size.get(), buf).then(
             [](future<int> op){ return op.get() > 0; });
         };

       auto flse = [](future<int> op){ return 
future::make_ready_future(false);};

       auto copy = do_while(
         [&buf]() -> future<bool>
         {
           return streamR.read(512, buf)
             .choice(
                [](future<int> op){ return op.get() > 0; }, write, flse);
         });
```

With resumable functions, the same code snippet would be:

```
int cnt = 0;

do
{
    cnt = await streamR.read(512, buf);

    if ( cnt == 0 ) break;

    cnt = await streamW.write(cnt, buf);

} while (cnt > 0);
```

It is not necessarily a lot shorter, but undoubtedly easier to comprehend, more or less identical to a synchronous formulation of the same algorithm. Further, no special attention needs to be paid to object lifetimes.

Resumable functions are motivated by the need to adequately address asynchronous operations, but are not actually tied to the proposal for a standard representation of such operations. The definition in this proposal can be used with any types that fit the described patterns. For example, resumable functions may be used to implementing a system for fully synchronous co-routines.

That said, throughout this document, future<T> will be used as the primary example of usage and implementation.

2. The Proposal

2.1 Terminology

A resumable function is a function that is capable of split-phase execution, meaning that the function may be observed to return from an invocation without producing its final logical result or all of its side-effects. This act is defined as the function suspending its execution. 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.

After suspending, a resumable function may be resumed by the scheduling logic of the runtime and will eventually complete its logic, at which point it executes a return statement (explicit or implicit) and sets the function's result value in the placeholder.

It should thus be noted that there is an asymmetry between the function's observed behavior from the outside (caller) and the inside: the outside perspective is that function returns a value of type future<T> at the first suspension point, while the inside perspective is that the function returns a value of type T via a return statement, functions returning future<void>/shared_future<void> behaving somewhat different still.

Within the function, there are zero or more suspension points. A resumable function *may* pause when it reaches a suspension point. Given control-flow, it may or may not be the case that a resumable function actually reaches a suspension point before producing a value (of type T); conversely, a given suspension point may be reached many times during the execution of a function, again depending on its control-flow.

A resumable function *may* continue execution on another thread after resuming following a suspension of its execution.

2.2 Declaration and Definition

Resumable functions are identified by decorating the function definition and declaration with the “async” keyword following the formal argument list. In the grammar productions for function definitions and lambda expressions, the async keyword is placed right before the exception-specification.

Any function or lambda that can legally return a future<T>/shared_future<T> or future<void> /shared_future<void> may be a resumable function, regardless of scope. [1]

2.3 Restrictions

Resumable functions cannot use a variable number of arguments. For situations where varargs are necessary, the argument unwrapping may be placed in a function that calls a resumable function after doing the unwrapping of arguments.

The return type of a resumable function must be future<T> or shared_future<T>. The restrictions on T are defined by std::future, not this proposal, but T must be a copyable or movable type, or ‘void.’ It must also be possible to construct a variable of T without an argument; that is, it has to have an accessible (implicit or explicit) default constructor if it is of a class type.

Await expressions may not appear within the body of an exception handler and should not be executed while a lock on any kind is being held by the executing thread.

2. 4 Suspension Points

Within a resumable function, its resumption points are uniquely identified by the presence of the ‘await’, which is treated as a keyword or reserved identifier within resumable functions. It is used as a unary operator, which pauses the function and resumes it when its operand is available. The expression denoted by the ‘await’ keyword is called an “await expression.”

The unary operator has the same precedence as unary operator ‘!’ boolean. Therefore, these statements are equivalent:

```
int x = (await expr) + 10;
int x = await expr + 10;
```

The operator may take an operand of any type that is future<U>/shared_future<U> or convertible to future<U>/ shared_future<U>.

[1] Resumable functions may return void as a possible extension. This design decision can be discussed separately at the Bristol meeting.

If U is not 'void,' it produces a value of type U by waiting for the future to be filled and returning the value returned by the future's 'get()' function. If U is 'void,' the await expression must be the term of an expression statement. That is, it cannot be the operand of another expression (since it yields 'void').

The U used in the operand of any given await expression in a function does not correspond to or have to be related to, the operands of other await expressions or the return type of the function itself: the types of what the function consumes (using await) and produces (using return) are independent of each other.

2.5 Return Statements

A resumable function produces its final value when executing a return statement or, only in the case of future<void>/shared_future<void> as the function return type, it reaches its end of the function without executing a return statement.

For example:

```
future<int> abs(future<int> i_op) async
{
    int i = await i_op;
    return (i < 0) ? -i : i;
}
```

Within a resumable function declared to return S<T>, where S is a future or shared_future and T is not 'void'*, a return statement should accept an operand of type T*. In the case of a resumable function declared to return future<void> or shared_future<void>, a return statement should be of the form "return;" or be omitted. In such a function, reaching the end of the function represents termination of the function and filling of the result.

3. Interactions and Implementability

3.1 Interactions

Keywords

The proposal uses two special identifiers as keywords to declare and control resumable functions. These should cause no conflict with existing, working, code.

In the case of 'async', it appears in a place where it is not currently allowed and should therefore not cause any ambiguity. Introducing the use of 'async' as an identifier with a special meaning only when it appears in that position is therefore not a breaking change.

In the case of 'await,' it is globally reserved but meaningful only within the body of a function or within the argument of a decltype() expression.

A possible conflict, but still not a breaking change, is that the identifiers may be in use by existing libraries. In the case of 'async,' the context should remove the possibility of conflict, but 'await' is more difficult. When used with a parenthesized operand expression, it will be indistinguishable from a call to a function ‘await’ with one argument.

A second possible non-breaking conflict is if there is a macro of the name ‘await,’ in which preprocessing will create problems.

A quick search of the header files for the Microsoft implementation of the standard C++ and C libraries, the Windows 7 SDK, as well as a subset of the Boost library header files show that there are no such conflicts lurking within those common and important source bases.

Overload Resolution

From the perspective of a caller, there is nothing special about a resumable function when considering overload resolutions.

Expression Evaluation Order / Operator Precedence

This proposal introduces a new unary operator, only valid within resumable functions and decltype() expressions. The precedence of the operator is the same as that of the ‘!’ operator, i.e. Boolean negation.

3.2 Implementation: Heap-allocated activation frames

The following implementation sketch is not intended to serve as a reference implementation. It is included for illustrative purposes only. It has already been shown that a more sophisticated approach is possible and quite feasible. However, this design has the advantage of being fairly simple and also similar to the implementation that C# relies on for its support of asynchronous methods.

3.2.1 Function Definition

The definition of a resumable function results in the definition of the locals frame structure and the added function, into which the body of the resumable method is moved before being transformed. The resumable method itself is more or less mechanically changed to allocate an instance of the frame structure, copy or move the parameters, and then call the added method.

It’s worth pointing out that the frame structure used in this example is an artifact of our attempt to represent the transformations using valid C++ code. A “real” implementation would allocate a suitably large byte array and use that for storage of local variables and parameters. It would also run constructors and destructors at the correct point in the function, something that our source-code implementation cannot.

The definition:

```
    future<int> f(future<double> g) async { return ceil(await g); }
```

results in:

```
    struct _frame_f
    {
        int _state;
        future<int> _resultF;
        promise<int> _resultP;
        _frame_f(future<double> g) : g(g), _state(0)
        {
            _resultF = future<int>(_resultP);
        }
        future<double> g;
    };

    future<int> f(double g)
    {
        auto frame = std::make_shared<_frame_f>(g);

        _res_f(frame);

        return frame->_resultF;
    }

    void _res_f(const std::shared_ptr<_frame_f> &frame)
    {
        return ceil(await frame->g);
    }
```

Note that the body of the _res_f() function represents artistic license, as it represents a transitional state of the original body. It still needs to be transformed into its final form, as described in the next section.

3.2.2 Function Body

There are four main transformations that are necessary, not necessarily performed in the order listed: a) space for local variables needs to be added to the frame structure definition, b) the function prolog needs to branch to the last resumption point, c) await expressions need to be hoisted and then transformed into pause/resumption logic, and d) return statements need to be transformed to modify the _result field of the frame.

3.2.3 Allocating Storage

All variables (and temporaries) with lifetimes that statically span one or more resumption points need to be provided space in the heap-allocated structure. In the hand-translated version, their lifetimes are extended to span the entire function execution, but a real, low-level implementation must treat the local variable storage in the frame as just storage and not alter the object lifetimes in any way.

The heap-allocated frame is reference-counted so that it can be automatically deleted when there are no longer any references to it. In this source-code implementation, we're using std::shared_ptr<> for reference counting. Something more tailored may be used by a real implementation.

An implementation that cannot easily perform the necessary lifetime analysis before allocating space in the frame should treat all local variables and formal parameters of a resumable function as if their lifetimes span a resumption point. Doing so will increase the size of the heap-allocated frame and decrease the stack-allocated frame.

3.2.4 Function Prolog

The _state field of the frame structure contains an integer defining the current state of the function. A function that has not yet been paused always has _state == 0. With the exception of the initial state, there is a one-to-one correspondence between state identities and resumption points. Except for the value 0, the actual numerical value assigned to states has no significance, as long as each identity uniquely identifies a resumption point.

Each state is associated with one label (branch target), and at the prolog of the function is placed the equivalent of a switch-statement:

```
    void _res_f(std::shared_ptr<_frame_f> frame)
    {
        switch(frame->_state)
        {
        case 1: goto L1;
        case 2: goto L2;
        case 3: goto L3;
        case 4: goto L4;
        }

    }
```

In the hand-coded version, special care has to be taken when a resumption point is located within a try-block; an extra branch is required for each try block nesting a resumption point: first, the code branches to just before the try-block, then we allow the code to enter the block normally, then we branch again:

```
    void _res_f(std::shared_ptr<_frame_f> frame)
    {
        switch(frame->_state)
        {
        case 1: goto L1_1;
        case 2: goto L2;
        case 3: goto L3;
        case 4: goto L4;
        }

L1_1:
        try
        {
            switch(frame->_state)
            {
            case 1: goto L1;
            }

L1:
            ...
        }
    }
```

Depending on the implementation of try-blocks, such a chaining of branches may not be necessary in a low-level expression of the transformation.

3.2.5 Hoisting ‘await’ Expressions

Before transformation, each resumption point needs to be in one of these two forms:

```
        x = await expr;
        await expr;
```

In other words, embedded await operators need to be hoisted and assigned to temporaries, or simply hoisted in the case of void being the result type. The operand ‘expr’ also needs to be evaluated into a temporary, as it will be used multiple times in the implementation, before and after the resumption point.

Note that await expressions may appear in conditional (ternary) expressions as well as in short-circuit expressions, which may affects their hoisting in some compiler implementations.

3.2.6 Implementing await expressions

In our hand-coded implementation, the hoisted expression “t = await g;” is transformed thus:

```
        if ( !frame->g.ready() )
        {
            frame->_state = 1;
            frame->g.then(
              [=](future<double> op)
              {
                  __res_f(frame);
              });
            return;
        }

L1:
        t = frame->g.get();
```

In the case of ‘await g’ being used as the expression of an expression statement, i.e. the value is thrown away, the compiler must emit a call to ‘wait()’ after the resumption. Calling wait()when the result is not used gives the runtime a chance to raise any propagated exceptions that may otherwise go unobserved.

3.2.7 Transforming ‘return’ Statements

Return statements are simply transformed into calls to set the value contained in the _result objects, or overwrite it with a new object:

```
// return ceil(await g);

if ( frame->_state == 0 )
    frame->_resultF = make_ready_future<int>(ceil(t));
else
    frame->_resultP.set(ceil(t));
```

The test for _state == 0 is done to establish whether the function has ever been paused or not. If it has not, it means that the caller will not yet have been passed back the result instance and it is therefore not too late to replace it with a more efficient value holder. *It is an optimization and an implementation is not required to test for this condition.*

3.3 Implementation: Resumable Side Stacks

With this prototype implementation each resumable function has its own side stack. A side stack is a stack, separate from the thread's main stack, allocated by the compiler when a resumable function is invoked. This side stack lives beyond the suspension points until logical completion.

Consider the following:

```
void foo()
{
    future<bool> t = bar();
}
future<bool> bar() async
{
    do_work();
    //possible suspension points and synchronous work here
    return true;
}
```

When foo (a non-resumable function) is invoked, the current thread's stack is allocated. Next, when the function bar() is executed, the async keyword is recognized and the compiler creates a side stack, allocating it appropriately. Next, the compiler switches from the thread's current stack, to the new side stack, this is the push side stack operation. Depending on the implementation, the side stack can run on the same thread or a new thread. do_work(), a synchronous operation executes on the resumable function's side stack.

Next consider the following resumable function which has a suspension point denoted by the await keyword:

```
void foo()
{
    future<bool> t = bar();
    do_work();
    bool b = t.get();
}
future<bool> bar() async
```

```
{
     do_work();
    await some_value;
    do_more_work();
    return true;
}
```

From above, we know that resumable function bar() is currently running on its own side stack. After completing the synchronous work on the call stack, the function reaches a possible suspension point which is indicated with ‘await’. If the value at that point is not ready, bar() is suspended and a future<T>, (where T is the type of the function’s return statement), is returned to the calling function. The side stack is popped and the compiler switches back to the thread’s main stack. Function foo() continues to proceed until it gets blocked waiting for the future’s state to be ready. After some time, when some_value is fulfilled, the function bar() resumes from where it left off (the suspension point) and bar()’s side stack is pushed. When the end of the resumable function is reached, the previously returned future’s state is set to ready and its value is set to the function’s return value. Once the resumable function bar() is completed, the side stack is popped off and deleted, and the compiler switches back to the main thread.