Doc No: WG21 N3780
Date: 2013-09-26
Reply to: Nicolai Josuttis (nico@josuttis.de)
Subgroup: SG1 – Concurrency
Prev. Version: none

Why Deprecating async() is the Worst of all Options

The concurrency working group has come to the conclusion to propose to deprecate async(), which we introduced as new high-level concurrency interface with C++11. In this paper I want to point out why this is the worst of all options we had to solve the underlying "problem" so that we can avoid making this mistake.

As a disclaimer, I am not neutral in respect to this subject and a little bit emotional. While I try to present facts and opinions in a fair way, please apologize just in case this sounds offending for you. Also, please respect that English is not my native language that that we Germans tend to have no problems to make statements without only using nice words.

History and Context

In C++11, we introduced async() as a new convenient abstraction especially to newcomers as a remarkable simplified way of spawning work to (perhaps as-if) other thread. That is, the goal is to give non-experts the ability to benefit from concurrent platforms while minimizing their effort and level of knowledge they have to have.

Provided func1() and func2 are independent, instead of

```
int result = func1() + func2()
```

you can program:

```
// start both tasks asynchronously
std::future<int> result1(std::async(func1));
std::future<int> result2(std::async(func2));
...
// use outcome of both tasks
int result = result1.get() + result2.get();
```

or:

```
// start both tasks asynchronously
auto result1(std::async(func1));
auto result2(std::async(func2));
...
// use outcome of both tasks
int result = result1.get() + result2.get();
```

which has the same effect with the potential optimization that func1() and func2() run in parallel (with main()). No other special handling is necessary. The user does **not** have to deal with states, exceptions, etc.

It is my experience that this feature widely considered as an important and commendable example of the improvements of C++11.

Initially it was difficult to understand from the standard what happens if get() is not called. During preparation of the 2nd edition of the C++ Standard Library Book, I raised this as an issue with the committee in order to clarify it. The result of this discussion was, that it is our understanding that the destructor of future will block if the future is created by async() having an async launch policy:

```
{
  auto f = async(task)  // start task asynchronously if possible
  ...
} // blocks until end of task if no get() was called
```

At least one implementation that did not follow this understanding was adjusted accordingly. Fortunately, clarifying words are proposed to become part of C++14, now.

The "Problem"

The core motivation to deprecate async() is to my best knowledge and understanding as follows:

Unfortunately, it turned out that the decision to let future destructors block when returned by async() turned out to be different from futures coming from other sources, such as promises or packaged tasks.

As a consequence, applications or frameworks that process future coming from external sources, don't know whether the destructor will block:

```
{
   future<...> f = user_provided_function()
 } // might or might not block
```

This in itself is not a problem. This is just the semantics futures got following to what we standardized. However, under circumstances where we **expect** future destructors not to block, this is a problem.

Thus, given that:

- std::future<> sometimes blocks and
- unnecessarily blocking is harmful in some contexts (e.g. responsive GUI programming)

there are two different conclusions that can be drawn from these two points:

- This is unfortunate, but it's not a fundamental problem: It just means that std::future is not usable in contexts where blocking is considered harmful.
- This is a fundamental problem for C++, because the class to be used in such contexts must be called future.

That is, instead of accepting the status quo of C++11 for better or worse, **the "problem" exists only because there is a "need" to bring other wished/expected behavior to an existing symbol instead of introducing a new symbol for it.**

This also can be considered as: There is nothing wrong with the existing async() and std::future feature except that some frameworks "misuse" std::futures in a non-conforming way and want to this misuse to become valid code.

Possible Solutions

The first and maybe obvious solution is: Take what we standardized with C++11 for better or worse and provide something different using different symbols, so that programmers can switch to the new interface when it is available and mature without the fear that existing code will be broken:

However, as written, this seems not to be an acceptable solution for those who consider the current state as a severe problem. So, we started to deal with several proposals that went into the requested direction. We had a couple of options discussed and rejected. For example:

- Let futures returned by async() not block. However, this is not only a API change, it would introduce subtle program bugs, which are likely to be exploitable as security holes (see N3679).

- We could add additional future types so that some future types might block while others don't (see N3637). However, this is an API breaking change.

- We could add a member function to detect whether a future will or might block. For example:

  ```
  future<...> f = ...
  if (f.returned_by_async()) {
    ...
  }
  ```

  However, this does little to solve the problem, since there's not really a good option of what to do when this function says that it blocks.

- We could add a member function to bring the future into a state not to block. For example:

  ```
  future<...> f = ...
  f.detach()  // ensure destructor won't block (might cause other problems though)
  ```

  However, this leads to a couple of problems such as how to deal with exceptions coming from detached threads/futures.

So, again, we could leave the status as it is for better or worse. But instead, the proposed solution is now to **deprecate** async() as a whole **without providing an alternative in C++14** for application programmers that (still) want to use async() or a similar interface. The goal is to minimize applications of async() as it is now and signaling that this might not be supported in C++17 (or later).

Note that there is the announcement/expectation/hope that there will be a TS soon providing a better alternative. However, this alternative does not exist yet (although the discussion about this "problem" runs for more than a year now).

Even worse, it seems not to be clear yet, how the upcoming alternative to async() will look like and where exactly it will break existing valid C++11 code.

> I might be wrong here, but so far nobody of the guys that want to deprecate async() and claiming that there will be an alternative soon, has answered my request to describe what exactly will be broken with the announced alternative solution for async(). The range of answers I got informally was from "nothing will be broken" up to "std::future will be broken".
>
> Such an information would be very helpful to come up with a better alternative to "deprecate" only those applications of async() that will be broken instead of throwing the bay out with the bathwater by deprecating the feature as a whole.

The Consequences of Deprecating async()

So, having described the "problem", and the proposed resolution we have now, let's look what this "solution" to deprecate async() now without providing an alternative means:

- **We no longer have a recommended high level interface**
  and have to teach the following about C++11/C++14:

  > "There is/was a working convenient high level interface, but because under some circumstances (using frameworks with futures coming from different sources and wrongly expecting future not to block in the destructor) this could cause a problem.
  >
  > Instead of forcing to use futures as they are standardized or introduce a separate alternative, we deprecated the interface as a whole.
  >
  > So, dear application programmer, even in the simplest example (as locally trying to run two tasks in parallel), we no longer recommend to use this interface because it might be removed. To write corresponding portable code, you now have to use the low level interface provided with std::thread, std::promise, std::exception_ptr, etc. instead, but we hope (and expect) to have something better soon."

- **But, we still have to support the deprecated high-level interface**.
  Code using a deprecated feature is still valid code. Thus, code that gets a std::future from an external source still has to deal with the fact that the future might be created

with async().

AND:

If we finally disable async() with C++17 or later, then we will have 6 or more years of support of async() that return futures that might block in the destructor.

So, a key question is:

- Are all library implementors and compiler vendors willing in C++17 or later to disable code that was valid for 6 or more years?

I can't imagine that (especially because again and again the argument for features that even were provided without being standardized is that we have to keep it valid).

- **We blame ourselves and C++ and send a terrible message for the reputation of C++**

  After announcing something working and cool, we disable that feature, (which is not broken, just because it is not perfect), without providing a better alternative.

  Claiming that we blame C++ is not theory, it already has been started. For example, when Stefanus tweeted that async will or might become deprecated one answer was the following:

  > @gpakosz: @stefanusdutoit
  > **The fact that top men standardized something already broken tells me I'm not smart enough to use C++11 or 14 in production**

  This will reduce confidence in C++11 (and C++ as a whole) and therefore compromise the positive momentum switch C++11 has given us.

Summary

So the summary of the current situation is in my opinion pretty simple:

We “solved” the “problem” with the worst solution I can imagine of:

- We disable useful and working code without providing any alternative (now).
- We don't solve the problem.
- We blame us and C++ and compromise the reputation of C++/C++11/C++14.

For this reason, I strongly urge you to vote against deprecating async().

I also strongly suggest to agree on the following guidelines on deprecations (extending what Howard wrote in “[c++std-lib-34588] Re: Comments on CH 9 (Remove deprecated strstreams from the standard)“:

- To deprecate a feature of C++, the following conditions must all hold:

- There must be something seriously broken, such that the application of the feature is a dangerous. A “not-prefect” design is not enough.
- At the time of deprecation we have to provide an alternative, which has proven to be at least as powerful as the existing solution and fixing the problem that caused the deprecation.
- If possible, the alternative should not break existing code that was valid before. Especially a deprecated feature should still be able to use when the alternative solution is available.
- You have to wait X years/standards to remove a deprecated feature.

What we can do now?

Unfortunately, in all discussions about this topic, all trials to solve this problem without throwing out the baby with the bathwater, were rejected. So now we have come into a situation that only a few options are left (provided we agree that deprecating async() is not an useful option).

So, I see only the following options left for C++14:

1. **Leave the situation as it is.** Accept this situation for better or worse. Provide something better if you can, which doesn’t break the existing standard.

   As a minor improvement we might help programmers to detect this problem with a minimum almost safe change, by adding a member function for futures that allows to detect whether the destructor might block (something like returned_by_async() as described above).

2. **Make a change NOW keeping the overall async() functionality.** Something like deprecating the return type of async() or changing the return type to something new or implementation defined, that behaves as future, but does not provide the ability to assign the return value of async to a future. This would mean that all programs using auto to declare the return type of async() remain valid while other programs have to be fixed based on an error detected by the compiler.

   In an informal conversion Herb told me that the request to deprecate async() as a whole is only driven by the fact that we have no way to deprecate parts of an interface. So we could introduce such an ability now instead of making a bad move.

The damage of async on the future type is done. If we cannot live with the backwards incompatibility of changing the return type of async (or deprecating the return type only), then, instead of trying to preemptively deprecate async(), we will need to choose a new name for a vocabulary type along the lines of future that does not have these issues.