
18.1. The High-Level Interface: `async()` and Futures

For novices, the best starting point to run your program with multiple threads is the high-level interface of the C++ standard library provided by `std::async()` and class `std::future<>` :

- `async()` provides an interface to let a piece of functionality, a *callable object* (see Section 4.4, page 54), run in the background as a separate thread, if possible.
- Class `future<>` allows you to wait for the thread to be finished and provides access to its outcome: return value or exception, if any.

This section introduces this high-level interface in detail, extended by an introduction to class `std::shared_future<>` , which allows you to wait for and process the outcome of a thread at multiple places.

18.1.1. A First Example Using `async()` and Futures

Suppose that we have to compute the sum of two operands returned by two function calls. The usual way to program that would be as follows:

```
func1() + func2()
```

This means that the processing of the operands happens sequentially. The program will first call `func1()` and then call `func2()` or the other way round (according to language rules, the order is undefined). In both cases, the overall processing takes the duration of `func1()` *plus* the duration of *func2()* plus computing the sum.

These days, using the multiprocessor hardware available almost everywhere, we can do better. We can at least try to run `func1()` and `func2()` in parallel so that the overall duration takes only the maximum of the duration of `func1()` and `func2()` plus processing the sum.

Here is a first program doing that:

Click here to view code image

```
// concurrency/async1.cpp

#include <future>
#include <thread>
#include <chrono>
#include <random>
#include <iostream>
#include <exception>
using namespace std;

int doSomething (char c)
{
    // random-number generator (use c as seed to get different sequences)
    std::default_random_engine dre(c);
    std::uniform_int_distribution<int> id(10,1000);

    // loop to print character after a random period of time
    for (int i=0; i<10; ++i) {
        this_thread::sleep_for(chrono::milliseconds(id(dre)));
        cout.put(c).flush();
    }

    return c;
}

int func1 ()
{
    return doSomething('.');
}

int func2 ()
{
     return doSomething('+');
}

int main()
{
    std::cout << "starting func1() in background"
```

```
                << " and func2() in foreground:" << std::endl;

    // start func1() asynchronously (now or later or never):
    std::future<int> result1(std::async(func1));

    int result2 = func2();      // call func2() synchronously (here and now)

    // print result (wait for func1() to finish and add its result to result2
    int result = result1.get() + result2;

    std::cout << "\nresult of func1()+func2(): " << result
              << std::endl;
}
```

To visualize what happens, we simulate the complex processings in `func1()` and `func2()` by calling `doSomething()`, which from time to time prints a character passed as argument[1] and finally returns the value of the passed character as `int`. "From time to time" is implemented using a random-number generator to specify intervals, which `std::this_thread::sleep_for()` uses as timeouts for the current thread (see Section 17.1, page 907, for details of random numbers, and Section 18.3.7, page 981, for details of `sleep_for()`). Note that we need a unique *seed* for the constructor of the random-number generator (here, we use the passed character `c`) to ensure that the generated random-number sequences differ.

[1] Output by concurrent threads is possible but might result in interleaved characters (see Section 4.5, page 56).

Instead of calling:

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

we call:

```
std::future<int> result1(std::async(func1));
int result2 = func2();
int result = result1.get() + result2;
```

So, first, we *try* to start `func1()` in the background, using `std::async()`, and assign the result to an object of class `std::future`:

```
std::future<int> result1(std::async(func1));
```

Here, `async()` *tries* to start the passed functionality immediately asynchronously in a separate thread. Thus, `func1()` ideally starts here without blocking the `main()` function. The returned *future* object is necessary for two reasons:

1. It allows access to the "future" outcome of the functionality passed to `async()`. This outcome might be either a return value or an exception. The future object has been specialized by the return type of the functionality started. If just a background task was started that returns nothing it has to be `std::future<void>`.
2. It is necessary to ensure that sooner or later, the passed functionality gets called. Note that I wrote that `async()` *tries* to start the passed functionality. If this didn't happen we need the future object to force a start when we need the result or want to ensure that the functionality was performed. Thus, you need the future object even if you are not interested in the outcome of a functionality started in the background.

To be able to exchange data between the place that starts and controls the functionality and the returned future object, both refer to a so-called *shared state* (see Section 18.3, page 973).

Of course, you can also, and usually will, use `auto` to declare the future (I explicitly wanted to demonstrate its type here):

```
auto result1(std::async(func1));
```

Second, we start `func2()` in the foreground. This is a normal synchronous function call so that the program blocks here:

```
int result2 = func2();
```

Thus, if `func1()` successfully was started by `async()` and didn't end already, we now have `func1()` and `func2()` running in parallel.

Third, we process the sum. This is the moment when we need the result of `func1()`. To get it, we call `get()` for the returned future:

```
int result = result1.get() + result2;
```

Here, with the call of `get()`, one of three things might happen:

1. If `func1()` was started with `async()` in a separate thread and has already finished, you immediately get its result.

2. If `func1()` was started but has not finished yet, `get()` blocks and waits for its end and yields the result.

3. If `func1()` was not started yet, it will be forced to start now and, like a synchronous function call, `get()` will block until it yields the result.

This behavior is important because it ensures that the program still works on a single-threaded environment or, if for any other reason, it was not possible for `async()` to start a new thread.

A call of `async()` does not guarantee that the passed functionality gets started and finished. If a thread is available, it will start, but if not — maybe your environment does not support multithreading or no more threads are available — the call will be deferred until you explicitly say that you need its outcome (calling `get()`) or just want the passed functionality to get done (calling `wait()` ; see Section 18.1.1, page 953).

Thus, the combination of

```
std::future<int> result1(std::async(func1));
```

and

```
result1.get()
```

allows you to optimize a program in a way that, if possible, `func1()` runs in parallel while the next statements in the main thread are processed. If it is not possible to run it in parallel, it will be called sequentially when `get()` gets called. This means that, in any case, it is guaranteed that after `get()` , `func1()` was called either asynchronously or synchronously.

Accordingly, two kinds of outputs are possible for this program. If `async()` could successfully start `func1()` , the output might be something like the following:

Click here to view code image

```
starting func1() in background and func2() in foreground:
++..++++.++.+.+.....
result of func1()+func2(): 89
```

If `async()` couldn't start `func1()` , it will run after `func2()` , when `get()` gets called, so that the program will have the following output:

Click here to view code image

```
starting func1() in background and func2() in foreground:
++++++++++..........
result of func1()+func2(): 89
```

So, based on this first example, we can define a general way to make a program faster: You can modify the program so that it might benefit from parallelization, if the underlying platform supports it, but still works as before on single-threaded environments. For this, you have to do the following:

- `#include <future>`
- Pass some functionality that could run on its own in parallel as a *callable object* to `std::async()`
- Assign the result to a `future<` *ReturnType* `>` object
- Call `get()` for the `future<>` object when you need the result or want to ensure that the functionality that was started has finished

Note, however, that this applies only when no *data race* occurs, which means that two threads concurrently use the same data resulting in undefined behavior (see Section 18.4.1, page 982).

Note that without calling `get()` , there is no guarantee that `func1()` will ever be called. As written, if `async()` couldn't start the passed functionality immediately, it will *defer* the call so that it gets called only when the outcome of the passed functionality explicitly is requested with `get()` (or `wait()` ; see page 953). But without such a request, the termination of `main()` will even terminate the program without ever calling the background thread.

Note also that you have to ensure that you ask for the result of a functionality started with `async()` no earlier than necessary. For example, the following "optimization" is probably not what you want:

Click here to view code image

```
std::future<int> result1(std::async(func1));
int result = func2() + result1.get();  // might call func2() after func1()
ends
```

Because the evaluation order on the right side of the second statement is unspecified, `result1.get()` might be called before `func2()` so that you have sequential processing again.

To have the best effect, in general, your goal should be to maximize the distance between calling `async()` and calling `get()` .

Or, to use the terms of [N3194:Futures]: *Call early* and *return late*.

If the operation passed to `async()` doesn't return anything, `async()` yields a `future<void>`, which is a partial specialization for `future<>`. In that case, `get()` returns nothing:

Click here to view code image

```
std::future<void> f(std::async(func));    // try to call func asynchronously
...
f.get();     // wait for func to be done (yields void)
```

Note, finally, that the object passed to `async()` may be any type of a *callable object*: function, member function, function object, or lambda (see Section 4.4, page 54). Thus, you can also pass the functionality that should run in its own thread inline as a lambda (see Section 3.1.10, page 28):

```
std::async([]{ ... })  // try to perform ... asynchronously
```

Using Launch Policies

You can force `async()` to not defer the passed functionality, by explicitly passing a *launch policy*[2] directing `async()` that it should definitely start the passed functionality asynchronously the moment it is called:

[2] The launch policy is a *scoped enumeration*, so you have to qualify the values (enumerators) with `std::launch` or `launch` (see Section 3.1.13, page 32).

Click here to view code image

```
// force func1() to start asynchronously now or throw std::system_error
std::future<long> result1= std::async(std::launch::async, func1);
```

If the asynchronous call is not possible here, the program will throw a `std::system_error` exception (see Section 4.3.1, page 43) with the error code `resource_unavailable_try_again`, which is equivalent to the POSIX `errno EAGAIN` (see Section 4.3.2, page 45).

With the `async` launch policy, you don't necessarily have to call `get()` anymore because, if the lifetime of the returned future ends, the program will wait for `func1()` to finish. Thus, if you don't call `get()`, leaving the scope of the future object (here the end of `main()`) will wait for the background task to end. Nevertheless, also calling `get()` here before a program ends makes the behavior clearer.

If you don't assign the result of `std::async(std::launch::async, ... )` anywhere, the caller will block until the passed functionality has finished, which would mean that this is nothing but a synchronous call.[3]

[3] Note that there was some controversial understanding and discussion in the standardization committee about how to interpret the current wording if the result of `async()` is not used. This was the result of the discussion and should be the behavior of all implementations.

Likewise, you can force a deferred execution by passing `std::launch:deferred` as launch policy to `async()`. In fact, with the following you defer `func1()` until `get()` is called for `f`:

Click here to view code image

```
std::future<...> f(std::async(std::launch::deferred,
                              func1));    // defer func1 until get()
```

Here, it is guaranteed that `func1()` never gets called without `get()` (or `wait()`; see page 953). This policy especially allows to program *lazy evaluation*. For example:[4]

[4] Thanks to Lawrence Crowl for pointing this out and providing an example.

Click here to view code image

```
auto f1 = std::async( std::launch::deferred, task1 );
auto f2 = std::async( std::launch::deferred, task2 );
...
auto val = thisOrThatIsTheCase() ? f1.get() : f2.get();
```

In addition, explicitly requesting a `deferred` launch policy might help to simulate the behavior of `async()` on a single-threaded environment or simplify debugging (unless race conditions are the problem).

Dealing with Exceptions

So far, we have discussed only the case when threads and background tasks run successfully. However, what happens when an exception occurs?

The good news is: Nothing special; `get()` for futures also handles exceptions. In fact, when `get()` is called and the background operation was or gets terminated by an exception, which was/is not handled inside the thread, this exception gets propagated again. As a result, to deal with exceptions of background operations, just do the same with `get()` as you would do when calling the operation synchronously.

For example, let's start a background task with an endless loop allocating memory to insert a new list element:[5]

[5] Trying to consume memory until an exception occurs is bad practice, of course, which on some operating systems might cause trouble. So beware before trying this example out.

Click here to view code image

```
// concurrency/async2.cpp

#include <future>
#include <list>
#include <iostream>
#include <exception>
using namespace std;

void task1()
{
   // endless insertion and memory allocation
   // - will sooner or later raise an exception
   // - BEWARE: this is bad practice
   list<int> v;
   while (true) {
       for (int i=0; i<1000000; ++i) {
           v.push_back(i);
       }
       cout.put('.').flush();
   }
}

int main()
{
    cout << "starting 2 tasks" << endl;
    cout << "- task1: process endless loop of memory consumption" <<
endl;
    cout << "- task2: wait for <return> and then for task1" << endl;

    auto f1 = async(task1);  // start task1() asynchronously (now or later or
never)

    cin.get();  // read a character (like getchar())

    cout << "\nwait for the end of task1: " << endl;
    try {
        f1.get();  // wait for task1() to finish (raises exception if any)
    }
    catch (const exception& e) {
        cerr << "EXCEPTION: " << e.what() << endl;
    }
}
```

Sooner or later, the endless loop will raise an exception (probably a `bad_alloc` exception; see Section 4.3.1, page 43). This exception will terminate the thread because it isn't caught. The future object will keep this state until `get()` is called. With `get()`, the exception gets further propagated inside `main()`.

Now we can summarize the interface of `async()` and futures as follows: `async()` gives a programming environment the chance to start in parallel some processing that is used later (where `get()` is called). In other words, if you have some independent functionality `f`, you can benefit from parallelization, if possible, by passing `f` to `async()` the moment you have all you need to call `f` and replacing the expression where you need the result or outcome of `f` by a `get()` for the future returned by `async()`. Thus, you have the same behavior but the chance of better performance because `f` might run in parallel before the outcome of `f` is needed.

Waiting and Polling

You can call `get()` for a `future<>` only once. After `get()`, the future is in an invalid state, which can be checked only by calling `valid()` for the future. Any call other than destruction will result in undefined behavior (see Section 18.3.2, page 975, for details).

But futures also provide an interface to wait for a background operation to finish without processing its outcome. This interface is callable more than once and might be combined with a duration or timepoint to limit the amount of waiting time.

Just calling `wait()` forces the start of a thread a future represents and waits for the termination of the background operation:

Click here to view code image

```
std::future<...> f(std::async(func));     // try to call func asynchronously
...
f.wait();          // wait for func to be done (might start background task)
```

Two other `wait()` functions exist for futures, but those functions do *not* force the thread to get started, if it hasn't started yet:

1. With `wait_for()`, you can wait for a limited time for an asynchronously running operation by passing a duration:

Click here to view code image

```
std::future<...> f(std::async(func));     // try to call func asynchronously
...
f.wait_for(std::chrono::seconds(10));      // wait at most 10 seconds for
func
```

2. With `wait_until()`, you can wait until a specific timepoint has reached:

Click here to view code image

```
std::future<...> f(std::async(func));     // try to call func asynchronously
...
f.wait_until(std::system_clock::now()+std::chrono::minutes(1));
```

Both `wait_for()` and `wait_until()` return one of the following:

- `std::future_status::deferred` if `async()` deferred the operation and no calls to `wait()` or `get()` have yet forced it to start (both functions return immediately in this case)
- `std::future_status::timeout` if the operation was started asynchronously but hasn't finished yet (if the waiting expired due to the passed timeout)
- `std::future_status::ready` if the operation has finished

Using `wait_for()` or `wait_until()` especially allows to program so-called *speculative execution*. For example, consider a scenario where we must have a usable result of a computation within a certain time, and it would be nice to have an accurate answer:[6]

[6] Thanks to Lawrence Crowl for pointing this out and providing an example.

Click here to view code image

```
int quickComputation();        // process result "quick and dirty"
int accurateComputation();     // process result "accurate but slow"
std::future<int> f;   // outside declared because lifetime of
accurateComputation()
                      // might exceed lifetime of bestResultInTime()
int bestResultInTime()
{
    // define time slot to get the answer:
    auto tp = std::chrono::system_clock::now() +
std::chrono::minutes(1);

    // start both a quick and an accurate computation:
    f = std::async (std::launch::async, accurateComputation);
    int guess = quickComputation();

    // give accurate computation the rest of the time slot:
    std::future_status s = f.wait_until(tp);

    // return the best computation result we have:
    if (s == std::future_status::ready) {
        return f.get();
    }
    else {
        return guess;     // accurateComputation() continues
    }
}
```

Note that the future `f` can't be a local object declared inside `bestResultInTime()` because when the time was too short to finish `accurateComputation()` the destructor of the future would block until that asynchronous task has finished.

By passing a zero duration or a timepoint that has passed, you can simply "poll" to see whether a background task has started and/or is (still) running:

Click here to view code image

```
future<...> f(async(task));      // try to call task asynchronously
...
// do something while task has not finished (might never happen!)
while (f.wait_for(chrono::seconds(0)) != future_status::ready) {
    ...
}
```

Note, however, that such a loop might never end, because, for example, on single-threaded environments, the call will be deferred until `get()` is called. So you either should call `async()` with the `std::launch::async` launch policy passed as first argument or check explicitly whether `wait_for()` returns `std::future_status::deferred` :

Click here to view code image

```
future<...> f(async(task));      // try to call task asynchronously
...
// check whether task was deferred:
if (f.wait_for(chrono::seconds(0)) != future_status::deferred) {
    // do something while task has not finished
    while (f.wait_for(chrono::seconds(0)) != future_status::ready) {
        ...
    }
}
...
auto r = f.get();     // force execution of task and wait for result (or exception)
```

Another reason for an endless loop here might be that the thread executing the loop has the processor and the other threads are not getting any time to make the future ready. This can reduce the speed of programs dramatically. The quickest fix is to call `yield()` (see Section 18.3.7, page 981) inside the loop:

Click here to view code image

```
std::this_thread::yield();     // hint to reschedule to the next thread
```

and/or sleep for a short period of time.

See Section 5.7, page 143, for details of durations and timepoints, which can be passed as arguments to `wait_for()` and `wait_until()` . Note that `wait_for()` and `wait_until()` usually will differ when dealing with system-time adjustments (see Section 5.7.5, page 160, for details).

18.1.2. An Example of Waiting for Two Tasks

This third program demonstrates a few of the abilities just mentioned:

Click here to view code image

```
// concurrency/async3.cpp

#include <future>
#include <thread>
#include <chrono>
#include <random>
#include <iostream>
#include <exception>
using namespace std;

void doSomething (char c)
{
    // random-number generator (use c as seed to get different sequences)
    default_random_engine dre(c);
    uniform_int_distribution<int> id(10,1000);

    // loop to print character after a random period of time
    for (int i=0; i<10; ++i) {
        this_thread::sleep_for(chrono::milliseconds(id(dre)));
        cout.put(c).flush();
    }
}

int main()
{
     cout << "starting 2 operations asynchronously" << endl;

     // start two loops in the background printing characters . or +
     auto f1 = async([]{ doSomething('.'); });
     auto f2 = async([]{ doSomething('+'); });
```

```
        // if at least one of the background tasks is running
        if (f1.wait_for(chrono::seconds(0)) != future_status::deferred ||
            f2.wait_for(chrono::seconds(0)) != future_status::deferred) {
            // poll until at least one of the loops finished
            while (f1.wait_for(chrono::seconds(0)) != future_status::ready
&&
                   f2.wait_for(chrono::seconds(0)) != future_status::ready)
{
                ...;
                this_thread::yield();     // hint to reschedule to the next thread
            }
        }
        cout.put('\n').flush();

        // wait for all loops to be finished and process any exception
        try {
            f1.get();
            f2.get();
        }
        catch (const exception& e) {
            cout << "\nEXCEPTION: " << e.what() << endl;
        }
        cout << "\ndone" << endl;
}
```

Again, we have an operation `doSomething()` that from time to time prints a character passed as argument (see Section 18.1.1, page 948).

Now, with `async()`, we start `doSomething()` twice in the background, printing two different characters using different delays generated by the corresponding random-number sequences:

```
auto f1 = std::async([]{ doSomething('.'); });
auto f2 = std::async([]{ doSomething('+'); });
```

Again, in multithreading environments, there would now be two operations simultaneously running that “from time to time” print different characters.

Next, we “poll” to see whether one of the two operations has finished:[7]

[7] Without doing something useful inside the loop, this would just be busy waiting, which means that the problem would be better solved with condition variables (see Section 18.6.1, page 1003).

Click here to view code image

```
while (f1.wait_for(chrono::seconds(0)) != future_status::ready &&
       f2.wait_for(chrono::seconds(0)) != future_status::ready) {
    ...
    this_thread::yield();  // hint to reschedule to the next thread
}
```

However, because this loop would never end if neither of the tasks were launched in the background when `async()` was called, we first have to check whether at least one operation was not deferred:

Click here to view code image

```
if (f1.wait_for(chrono::seconds(0)) != future_status::deferred ||
    f2.wait_for(chrono::seconds(0)) != future_status::deferred) {
        ...
}
```

Alternatively, we could call `async()` with the `std::launch:async` launch policy.

When at least one background operation has finished or none of them was started, we write a newline character and then wait for both loops to end:

```
f1.get();
f2.get();
```

We use `get()` here to process any exception that might have occurred.

In a multithreading environment, the program might, for example, have the following output:

```
starting 2 operations asynchronously
++.++..+.+..++.+.+
..
done
```

Note that regarding the order of all three characters `.`, `+`, and *newline*, nothing is guaranteed. It might be typical that the first character is a dot because this is the output from the first operation started — thus, started a little bit earlier — but as you can see here, a `+` might also come first. The characters `.` and `+` might be mixed, but this also is not guaranteed. In fact, if you remove the `sleep_for()` statement, which enforces the delay between each printing of the passed character, the first loop is done before the first context switch to the other thread, so the output might more likely become:

```
starting 2 operations asynchronously
..........
++++++++++
done
```

This output will also result if the environment doesn't support multithreading, because in that case, both calls of `doSomething()` will be called synchronously with the calls of `get()`.

Also, it is not clear when the newline character gets printed. This might happen before any other characters are written if the execution of both background tasks is deferred until `get()` is called. Then the deferred tasks will be called one after the other:

```
starting 2 operations asynchronously

..........++++++++++
done
```

The only thing we know is that newline won't be printed before one of the loops has finished. It is not even guaranteed that newline comes directly after the last character of one of the sequences, because it might take some time until the end of one of the loops is recorded in the corresponding future object and this recorded state is evaluated (note that this is not real-time processing). For this reason, you might have an output where a couple of `+` characters are written after the last dot and before the newline character:

```
starting 2 operations asynchronously
.+..+..+..+.+..++
+++
done
```

Passing Arguments

The previous example demonstrated one way to pass arguments to a background task: You simply use a lambda (see Section 3.1.10, page 28), which calls the background functionality:

```
auto f1 = std::async([]{ doSomething('.'); });
```

Of course, you can also pass arguments that existed before the `async()` statement. As usual, you can pass them by value or by reference:

Click here to view code image

```
char c = '@';
auto f = std::async([=]{      // =: can access objects in scope by value
                        doSomething(c);   // pass copy of c to doSomething()
                    });
```

By defining the *capture* as `[=]`, you pass a *copy* of `c` and all other visible objects to the lambda, so inside the lambda you can pass this copy of `c` to `doSomething()`.

However, there are other ways to pass arguments to `async()` because `async()` provides the usual interface for *callable objects* (see Section 4.4, page 54). For example, if you pass a function pointer as the first argument to `async()`, you can pass multiple additional arguments, which are passed as parameters to the function called:

Click here to view code image

```
char c = '@';
auto f = std::async(doSomething,c);    // call doSomething(c) asynchronously
```

You can also pass arguments by reference, but the risk of doing so is that the values passed become invalid before the background task even starts. This applies to both lambdas and functions directly called:

Click here to view code image

```
char c = '@';
auto f = std::async([&]{ doSomething(c); });        // risky!

char c = '@';
auto f = std::async(doSomething,std::ref(c));      // risky!
```

If you control the lifetime of the argument passed so that it exceeds the background task, you can do it. For example:

Click here to view code image

```
void doSomething (const char& c);      // pass character by reference
```

```
...
char c = '@';
auto f = std::async([&]{ doSomething(c); });      // pass c by reference
...
f.get();      // needs lifetime of c until here
```

But beware: If you pass arguments by reference to be able to modify them from a separate thread, you can easily run into undefined behavior. Consider the following example where after trying to start an output loop for printing a character in the background you switch the character printed:

Click here to view code image

```
void doSomething (const char& c);     // pass character by reference
...
char c = '@';
auto f = std::async([&]{ doSomething(c); });      // pass c by reference
...
c = '_';   // switch output of doSomething() to underscores, if it still runs
f.get();   // needs lifetime of c until here
```

First, the order of accessing `c` here and in `doSomething()` is undefined. Thus, the switch of the output character might come before, in the middle of, or after the output loop. Even worse, because we modify `c` in one thread and another thread reads `c`, this is a nonsynchronized concurrent access to the same object (a so-called *data race*, see Section 18.4.1, page 982), which results in undefined behavior unless you protect the concurrent access by using mutexes (see Section 18.5, page 989) or atomics (see Section 18.7, page 1012).

So, let me make clear: **If you start to use** `async()`**, you should pass *all* objects necessary to process the passed functionality *by value* so that** `async()` **uses only *local copies*.** If copying is too expensive, ensure that the objects are passed as constant reference and that `mutable` is not used. In any other case, read Section 18.4, page 982, and make sure that you understand the implications of your approach.

You can also pass a pointer to a member function to *async()*. In that case, the first argument after the member function has to be a reference or a pointer to the object for which the member function gets called:

Click here to view code image

```
class X
{
  public:
    void mem (int num);
    ...
};
...

X x;
auto a = std::async(&X::mem, x, 42);     // try to call x.mem(42)
asynchronously
...
```

18.1.3. Shared Futures

As we have seen, class `std::future` provides the ability to process the future outcome of a concurrent computation. However, you can process this outcome only once. A second call of `get()` results in undefined behavior (according to the C++ standard library, implementations are encouraged but not required to throw a `std::future_error`).

However, it sometimes makes sense to process the outcome of a concurrent computation more than once, especially when multiple other threads process this outcome. For this purpose, the C++ standard library provides class `std::shared_future` . Here, multiple `get()` calls are possible and yield the same result or throw the same exception.

Consider the following example:

Click here to view code image

```
// concurrency/sharedfuture1.cpp

#include <future>
#include <thread>
#include <iostream>
#include <exception>
#include <stdexcept>
using namespace std;

int queryNumber ()
{
    // read number
    cout << "read number: ";
    int num;
```

```
        cin >> num;

        // throw exception if none
        if (!cin) {
            throw runtime_error("no number read");
        }

        return num;
    }

    void doSomething (char c, shared_future<int> f)
    {
         try {
             // wait for number of characters to print
             int num = f.get();    // get result of queryNumber()

             for (int i=0; i<num; ++i) {
                 this_thread::sleep_for(chrono::milliseconds(100));
                 cout.put(c).flush();
             }
         }
         catch (const exception& e) {
             cerr << "EXCEPTION in thread " << this_thread::get_id()
                       << ": " << e.what() << endl;
         }
    }

    int main()
    {
        try {
            // start one thread to query a number
            shared_future<int> f = async(queryNumber);

            // start three threads each processing this number in a loop
            auto f1 = async(launch::async,doSomething,'.',f);
            auto f2 = async(launch::async,doSomething,'+',f);
            auto f3 = async(launch::async,doSomething,'*',f);

            // wait for all loops to be finished
            f1.get();
            f2.get();
            f3.get();
         }
         catch (const exception& e) {
             cout << "\nEXCEPTION: " << e.what() << endl;
         }
         cout << "\ndone" << endl;
    }
```

In this example, one thread calls `queryNumber()` to query an integral value, which is then used by other threads already running. To perform this task, the result of `std::async()` , which starts the query thread, gets assigned to a `shared_future` object, specialized for the return value:

```
shared_future<int> f = async(queryNumber);
```

Thus, a shared future can be initialized by an ordinary `future` , which moves the state from the future to the shared future. To be able to use `auto` for this declaration, you can, alternatively, use the `share()` member function:

```
auto f = async(queryNumber).share();
```

Internally, all shared future objects share the *shared state*, which `async()` creates to store the outcome of the passed functionality (and store the functionality itself if it is deferred).

The shared future is then passed to the other threads, starting `doSomething()` with the shared future as second argument:

```
auto f1 = async(launch::async,doSomething,'.',f);
auto f2 = async(launch::async,doSomething,'+',f);
auto f3 = async(launch::async,doSomething,'*',f);
```

Inside each call of `doSomething()` , we wait for and process the result of `queryNumber()` by calling `get()` for the shared future passed:

Click here to view code image

```
void doSomething (char c, shared_future<int> f)
{
    try {
```

```
            int num = f.get();      // get result of queryNumber()
            ...
        }
        catch (const exception& e) {
            cerr << "EXCEPTION in thread " << this_thread::get_id()
                 << ": " << e.what() << endl;
        }
    }
```

If `queryNumber()` throws an exception, which happens if no integral value could be read, each call of `doSomething()` will get this exception with `f.get()`, so that the corresponding exception handling will occur.

Thus, after reading the value `5` as input, the output might be:

```
read number: 5
*+.*+.*.+*+.*.+
done
```

But if typing `'x'` as input, the output might be:

```
read number: x
EXCEPTION in thread 3: no number read
EXCEPTION in thread 4: no number read
EXCEPTION in thread 2: no number read

done
```

Note that the order of the thread outputs and the ID values are undefined (see Section 18.2.1, page 967, for details about thread IDs).

Also note that there is a minor difference in the declaration of `get()` between `future` and `shared_future`:

• For class `future<>`, `get()` is provided as follows (`T` is the type of the returned value):

Click here to view code image

```
T future<T>::get();              // general get()
T& future<T&>::get();            // specialization for references
void future<void>::get();        // specialization for void
```

where the first form returns the moved result or a copy of the result.

• For class `shared_future<>`, `get()` is provided as follows:

Click here to view code image

```
const T& shared_future<T>::get();    // general get()
T& shared_future<T&>::get();         // specialization for references
void shared_future<void>::get();     // specialization for void
```

where the first form returns a reference to the result value stored in the shared *shared state*.

Or, as [*N3194:Futures*] states:

"The single-use value `get()` is move optimized (e.g., `std::vector<int> v = f.get()`). ... The const reference `get()` is access optimized (e.g., `int i = f.get()[3]`)."

This design introduces the risk of lifetime or data race issues if returned values are modified (see Section 18.3.3, page 977, for details).

You could also pass a shared future by reference (that is, declare it as reference and use `std::ref()` to pass it):

Click here to view code image

```
void doSomething (char c, const shared_future<int>& f)
auto f1 = async(launch::async,doSomething,'.',std::ref(f));
```

Now, instead of using multiple shared future objects all sharing the same *shared state*, you'd use one shared future object to perform multiple `get()`'s (one in each thread). However, this approach is more risky. As a programmer you have to ensure that the lifetime of `f` (yes, `f`, not the *shared state* it refers to) is not smaller than for the threads started. In addition, note that the member functions of shared futures do not synchronize with themselves, although the shared *shared state* is synchronized. So, if you do more than just read data, you might need external synchronization techniques (see Section 18.4, page 982) to avoid *data races*, which would result in undefined behavior. Or as Lawrence Crowl, one of the authors of the concurrency library, wrote in a private communication: "If the code stays tightly coordinated, passing by reference is fine. If the code may propagate into regions with an incomplete understanding of the purpose and restrictions, passing by value is better. Copying the shared future is expensive, but not so expensive as to justify a latent bug in a large system."

For further details of class `shared_future` see Section 18.3.3, page 976.