**High-Level Interface: async() and Futures**

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 run in the background as a separate thread, if possible.

**future<>** allows you to wait for the thread to be finished and provides access to its outcome: return value or exception, if any.

# Example Using async() and Futures

Suppose we want to compute the sum of two operands returned by two function calls.

result = func1() + func2()

The processing of the operands happens sequentially. The overall processing takes the duration of func1() plus the duration of func2() plus computing the sum.

Using concurrency, we can 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.

#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();

          }

          cout << endl;

          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));

// auto result1(std::async(func1));                     you can use auto also

          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;

          return 0;

}

Output:

starting func1() in background and func2() in foreground:

++++++++++

..........

result of func1()+func2(): 89

**Explanation**

Instead of calling:

          int result = func1() + func2();

we call:

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

          int result2 = func2();

          int result = result1.get() + result2;

1.   We try to start func1() in the background

2.   We start func2() in the foreground

3.   We process the sum

**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));

You can also use auto to declare the future:

auto 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.

2.   It is necessary to ensure that sooner or later, the passed functionality gets called. If passed functionality does not start 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.

**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. To get result, 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.

You can call get() for a future<> only once.

# Using Launch Policies

## std::launch::async

Used to not defer the passed functionality

You can force async() to definitely start the passed functionality asynchronously the moment it is called:

// 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 with the error code resource\_unavailable\_try\_again (equivalent to the POSIX errno EAGAIN)

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. 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.

std::future<long> result1 = std::async(std::launch::async, func1);

                                                                                      // asynchronous call

std::async(std::launch::async, func1);                            // synchronous call

## std::launch::deferred

Used to defer the passed functionality

You can force a deferred execution, until get() is called, by passing std::launch:deferred as launch policy to async().

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()). This policy especially allows to program lazy evaluation.

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

**What happens when an exception occurs?**

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.

#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;

          }

          return 0;

}

Output:

starting 2 tasks

- task1: process endless loop of memory consumption

- task2: wait for <return> and then for task1

wait for the end of task1:

...............EXCEPTION: std::bad\_alloc

# 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.

The futures interface might be combined with a duration or timepoint to limit the amount of waiting time.

## wait()

forces the start of a thread a future represents and waits for the termination of the background operation:

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:

## wait\_for()

wait for a limited time for an asynchronously running operation by passing a duration:

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

## wait\_until()

wait until a specific timepoint has reached:

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 function 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.

## 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:

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:

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)) {

...

}

**Drawback:** on single-threaded environments, the call will be deferred until get() is called.

**Solution:** call async() with the std::launch::async launch policy passed as first argument or check explicitly whether wait\_for() returns std::future\_status::deferred:

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)

# Shared Futures

With class std::future you can process the outcome only once. A second call of get() results in undefined behavior.

class std::shared\_future is provided to process the outcome of a concurrent computation more than once, especially when multiple other threads process this outcome. Here, multiple get() calls are possible and yield the same result or throw the same exception.

## Example

#include <future>

#include <thread>

#include <iostream>

#include <exception>

#include <stdexcept>

using namespace std;

int queryNumber () {

cout << "read number: "; // 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;

}

return;

}

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;

return 0;

}

Output - 1:

read number: 3

+-\*+-\*+-\*

done

Output - 2:

read number: EXCEPTION in thread 126270916757248 : no number read

EXCEPTION in thread 126270829778688 : no number read

EXCEPTION in thread 126270907434752 : no number read

done

A shared future can be initialized by an ordinary future, which moves the state from the future to the shared future.

shared\_future<int> f = async(queryNumber);

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).

There is a minor difference in the declaration of get() between future and shared\_future:

For class future<> (T is the type of the returned value):

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<> :

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.

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.

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

void doSomething (char c, const shared\_future<int>& f)

auto f1 = async(launch::async,doSomething,’.’,std::ref(f));

# Async in Detail

std::async() is a convenience function to start some functionality in its own thread if possible.

The exact behavior of async() is complex and highly depends on the launch policy, which can be passed as the first optional argument.

Three standardized forms of how async() can be called as described here:

1. future async (std::launch::async, F func, args...)
2. future async (std::launch::deferred, F func, args...)
3. future async (F func, args...)

## future async (std::launch::async, F func, args...)

* Tries to start func with args as an asynchronous task (parallel thread).
* If this is not possible, it throws an exception of type std::system\_error with the error code std::errc::resource\_unavailable\_try\_again.
* Unless the program aborts, the started thread is guaranteed to finish before the program ends.
* The thread will finish:
  + If get() or wait() is called for the returned future
  + If the last object that refers to the shared state represented by the returned future gets destructed
* This implies that the call of async() will block until func has finished if the return value of async() is not used.

## future async (std::launch::deferred, F func, args...)

* Passes func with args as a “deferred” task, which gets synchronously called when wait() or get() for the returned future gets called.
* If neither wait() nor get() is called, the task will never start.

## future async (F func, args...)

* Is a combination of calling async() with launch policies std::launch:async and std::launch::deferred. According to the current situation, one of the two forms gets chosen. Thus, async() will defer the call of func if an immediate call in async launch policy is not possible.
* Thus, if async() can start a new thread for func, it gets started. Otherwise, func is deferred until get() or wait() gets called for the returned future.
* The only guarantee this call gives is that after calling get() or wait() for the returned future, func will have been called and finished.
* Without calling get() or wait() for the returned future, func might never get called.
* This form of async() will not throw a system\_error exception if it can’t call func asynchronously (it might throw a system error for other reasons, though).

**Note**

1. For all these forms of async(), func might be a callable object (function, member function, function object, lambda).
2. Passing a launch policy of std::launch::async|std::launch::deferred to async() results in the same behavior as passing no launching policy.
3. Passing 0 as launch policy results in undefined behavior (this case is not covered by the C++ standard library, and different implementations behave differently).

# Futures in Detail

Class future<>, represents the outcome of an operation. It can be a return value or an exception **but not both**.

If the future was returned by async() and the associated task was deferred, get() or wait() will start it synchronously. Note that wait\_for() and wait\_until() do not start a deferred task.

The outcome can be retrieved only once. For this reason, a future might have a valid or invalid state: valid means that there is an associated operation for which the result or exception was not retrieved yet.

The operations available for class future<> are:

|  |  |
| --- | --- |
| Operation | Effect |
| future f | Default constructor; creates a future with an invalid state |
| future f(rv) | Move constructor; creates a new future, which gets the state of rv, and invalidates the state of rv |
| f.~future() | Destroys the state and destroys \*this |
| f = rv | Move assignment; destroys the old state of f, gets the state of rv, and invalidates the state of rv |
| f.valid() | Yields true if f has a valid state, so you can call the following member functions |
| f.get() | Blocks until the background operation is done (forcing a deferred associated functionality to start synchronously), yields the result (if any) or raises any exception that occurred, and invalidates its state |
| f.wait() | Blocks until the background operation is done (forcing a deferred associated functionality to start synchronously) |
| f.wait\_for(dur) | Blocks for duration dur or until the background operation is done (a deferred thread is not forced to start) |
| f.wait\_until(tp) | Blocks until timepoint tp or until the background operation is done (a deferred thread is not forced to start) |
| f.share() | Yields a shared\_future with the current state and invalidates the state of f |

The return value of get() depends on the type future<> is specialized with:

1. If it is void, get() also has type void and returns nothing.
2. If the future is parametrized with a reference type, get() returns a reference to the return value.
3. Otherwise, get() returns a copy or move assigns the return value, depending on whether the return type supports move assignment semantics.

You can call get() only once, because get() invalidates the future’s state.

For a future that has an invalid state, calling anything else but the destructor, the move assignment operator, or valid() results in undefined behavior. For this case, the standard recommends throwing an exception of type future\_error with the code std::future\_errc::no\_state, but this is not required.

If accessing a future (except destructor, move assignment operator or valid()) that has an invalid state, the standard recommends throwing an exception of type future\_error with the code std::future\_errc::no\_state.

Neither a copy constructor nor a copy assignment operator is provided, ensuring that no two objects can share the state of a background operation. You can move the state to another future object only by calling the move constructor or the move assignment operator.

If the destructor is called for a future that is the last owner of a shared state and the associated task has started but not finished yet, the destructor blocks until the end of the task.

# Shared Futures in Detail

Class shared\_future<> provides the same semantics and interface as class future with the following differences:

1. Multiple calls of get() are allowed. Thus, get() does not invalidate its state.
2. Copy semantics (copy constructor, copy assignment operator) are supported.
3. get() is a constant member function returning a const reference to the value stored in the shared state. For class std::future, get() is a nonconstant member function returning a move-assigned copy (or a copy if that’s not supported), unless the class is specialized by a reference type.
4. Member share() is not provided.

With reference as return of get() data races are possible. Data races occur with unclear order of conflicting actions on the same data, such as nonsynchronized reads and writes from multiple threads, and result in undefined behavior.

The same problem applies to exceptions. One example discussed during the standardization was when an exception was caught by reference and then modified:

try {

shared\_future<void> sp = async(f);

sp.get();

}

catch (E& e) {

e.modify(); // risk of undefined behavior due to a data race

}

This code introduces a data race if another thread processes the exception.

**Solution:** It was proposed to require that current\_exception() and rethrow\_exception(), which are used internally to pass exceptions between threads, create copies of the exceptions. However, the costs for this change were considered too high.

# END