## 18.2. The Low-Level Interface: Threads and Promises

Besides the high-level interface of `async()` and (shared) futures, the C++ standard library provides a low-level interface to start threads and deal with them.

### 18.2.1. Class `std::thread`

To start a thread, you simply have to declare an object of class `std::thread` and pass the desired task as initial argument, and then either wait for its end or *detach* it:

Click here to view code image

```
void doSomething();

std::thread t(doSomething);       // start doSomething() in the background
...
t.join();   // wait for t to finish (block until doSomething() ends)
```

As for `async()`, you can pass anything that's a *callable object* (function, member function, function object, lambda; see Section 4.4, page 54) together with possible additional arguments. However, note again that unless you really know what you are doing, you should pass *all* objects necessary to process the passed functionality *by value* so that the `thread` uses only *local copies* (see Section 18.4, page 982, for some of the problems that might occur otherwise).

In addition, this is a low-level interface, so the interesting thing is what this interface does *not* provide compared to `async()` (see Section 18.1, page 946):

- Class `thread` doesn't have a launch policy. The C++ standard library always tries to start the passed functionality in a new thread. If this isn't possible, it throws a `std::system_error` (see Section 4.3.1, page 43) with the error code `resource_unavailable_try_again` (see Section 4.3.2, page 45).
- You have no interface to process the result or outcome of the thread. The only thing you can get is a unique thread ID (see Section 18.2.1, page 967).
- If an exception occurs that is not caught inside the thread, the program immediately aborts, calling `std::terminate()` (see Section 5.8.2, page 162). To pass exceptions to a context outside the thread `exception_ptr` s (see Section 4.3.3, page 52) have to be used.
- You have to declare whether, as a caller, you want to wait for the end of the thread (calling `join()` ) or to *detach* from the thread started to let it run in the background without any control (calling `detach()` ). If you don't do this before the lifetime of the thread object ends or a move assignment to it happens, the program aborts, calling `std::terminate()` (see Section 5.8.2, page 162).
- If you let the thread run in the background and `main()` ends, all threads are terminated abruptly.

Here is a first complete example:

Click here to view code image

```
// concurrency/thread1.cpp

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

void doSomething (int num, char c)
{
    try {
        // random-number generator (use c as seed to get different sequences)
        default_random_engine dre(42*c);
        uniform_int_distribution<int> id(10,1000);
        for (int i=0; i<num; ++i) {
            this_thread::sleep_for(chrono::milliseconds(id(dre)));
            cout.put(c).flush();
            ...
        }
    }
```

```
        // make sure no exception leaves the thread and terminates the program
        catch (const exception& e) {
            cerr << "THREAD-EXCEPTION (thread "
                 << this_thread::get_id() << "): " << e.what() << endl;
        }
        catch (...) {
            cerr << "THREAD-EXCEPTION (thread "
                 << this_thread::get_id() << ")" << endl;
        }
    }

    int main()
    {
        try {
          thread t1(doSomething,5,'.');   // print five dots in separate
    thread
          cout << "- started fg thread " << t1.get_id() << endl;

          // print other characters in other background threads
          for (int i=0; i<5; ++i) {
              thread t(doSomething,10,'a'+i); // print 10 chars in separate thread
              cout << "- detach started bg thread " << t.get_id() << endl;
              t.detach();   // detach thread into the background
          }

          cin.get();      // wait for any input (return)
          cout << "- join fg thread " << t1.get_id() << endl;
          t1.join();    // wait for t1 to finish
        }
        catch (const exception& e) {
          cerr << "EXCEPTION: " << e.what() << endl;
        }
    }
```

Here, in `main()`, we start a couple of threads that perform the statements of `doSomething()`. Both `main()` and `doSomething()` have corresponding try-catch clauses for the following reasons:

- In `main()`, creating a thread might throw a `std::system_error` (see Section 4.3.1, page 43) with the error code `resource_unavailable_try_again`.
- In `doSomething()`, started as `std::thread`, any uncaught exception would cause the program to terminate.

For the first thread started in `main()`, we later wait for it to finish:

**Click here to view code image**

```
    thread  t1(doSomething,5,'.');   // print five dots in separate thread
    ...
    t1.join();                       // wait for t1 to finish
```

The other threads are detached after they were started, so they still might be running at the end of `main()`:

**Click here to view code image**

```
    for (int i=0; i<5; ++i) {
        thread t(doSomething,10,'a'+i);   // print 10 chars in separate thread
        t.detach();                       // detach thread into the background
    }
```

As a consequence, the program would immediately terminate all background threads when `main()` ends, which is the case when, due to `cin.get()`, some input could be read, *and* due to `t1.join()`, the fifth dot as last character of the thread performing `doSomething(5,'.')` was written. Because the waiting for the input and the printing of the dots run in parallel, it doesn't matter what happens first.

For example, the program might have the following output if I press *Return* after the second dot was printed:

```
    - started fg thread 1
    - detach started bg thread 2
    - detach started bg thread 3
    - detach started bg thread 4
    - detach started bg thread 5
    - detach started bg thread 6
    ecad.dbcebabd.a
    - join fg thread 1
    b.ceade.bbcadbe.
```

**Beware of Detached Threads**

Detached threads can easily become a problem if they use nonlocal resources. The problem is that you lose control of a detached thread and have no easy way to find out whether and how long it runs. Thus, make sure that a detached thread does not access any objects after their lifetime has ended. For this reason, passing variables and objects to a thread by reference is always a risk. Passing arguments by value is strongly recommended.

Note, however, that the lifetime problem also applies to global and static objects, because when the program exits, the detached thread might still run, which means that it might access global or static objects that are already destroyed or under destruction. Unfortunately, this would result in undefined behavior.[8]

[8] Thanks to Hans Boehm and Anthony Williams for pointing out this problem.

So, as a general rule for detached threads, take into account the following:

- Detached threads should prefer to access local copies only.
- If a detached thread uses a global or static object, you should do one of the following:
  - Ensure that these global/static objects are not destroyed before all detached threads accessing them are finished (or finished accessing them). One approach to ensure this is to use condition variables (see Section 18.6, page 1003), which the detached threads use to signal that they have finished. Before leaving `main()` or calling `exit()`, you'd have to set these condition variables then to signal that a destruction is possible.[9]

[9] Ideally, you should use `notify_all_at_thread_exit()` (see Section 18.6.4, page 1011) to set the condition variable to ensure that all thread local variables are destructed.

  - End the program by calling `quick_exit()`, which was introduced exactly for this reason to end a program without calling the destructors for global and static objects (see Section 5.8.2, page 162).

Because `std::cin`, `std::cout`, and `std::cerr` and the other global stream objects (see Section 15.2.2, page 751) according to the standard "are not destroyed during program execution," access to these objects in detached threads should introduce no undefined behavior. However, other problems, such as interleaved characters, might occur.

Nevertheless, as a rule of thumb keep in mind that the only safe way to terminate a detached thread is with one of the "... `at_thread_exit()`" functions, which force the main thread to wait for the detached thread to truly finish. Or you can just ignore this feature, according to a reviewer who wrote: "Detached threads is one of those things that should be moved into the chapter on dangerous features that almost no one needs."

#### Thread IDs

As you can see, the program prints thread IDs provided either by the thread object or inside a thread, using namespace `this_thread` (also provided by `<thread>`):

Click here to view code image

```
void doSomething (int num, char c)
{
    ...
    cerr << "THREAD-EXCEPTION (thread "
         << this_thread::get_id() << ")" << endl;
    ...
}

thread t(doSomething,5,'.'); // print five dots in separate thread
cout << "- started fg thread " << t1.get_id() << endl;
```

This ID is a special type `std::thread::id`, which is guaranteed to be unique for each thread. In addition, class `id` has a default constructor that yields a unique ID representing "no thread":

Click here to view code image

```
std::cout << "ID of \"no thread\": " << std::thread::id()
          << std::endl;
```

The only operations allowed for thread IDs are comparisons and calling the output operator for a stream. You should not make any further assumptions, such as that "no thread" has ID `0` or the main thread has ID `1`. In fact, an implementation might generate these IDs on the fly when they are requested, not when the threads are started, so the number of the main thread depends on the number of requests for thread IDs before. So, the following code:

Click here to view code image

```
std::thread t1(doSomething,5,'.');
std::thread t2(doSomething,5,'+');
std::thread t3(doSomething,5,'*');
std::cout << "t3 ID:       " << t3.get_id() << std::endl;
std::cout << "main ID:     " << std::this_thread::get_id() << std::endl;
std::cout << "nothread ID: " << std::thread::id() << std::endl;
```

might print:

```
t3 ID:       1
main ID:     4
```

```
nothread ID: 0
```

or:

```
t3 ID:       3
main ID:     4
nothread ID: 0
```

or:

```
t3 ID:       1
main ID:     2
nothread ID: 3
```

or even characters as thread IDs.

Thus, the only way to identify a thread, such as a master thread, is to compare it to its saved ID when it was started:

**Click here to view code image**

```
std::thread::id masterThreadID;
void doSomething()
{
    if (std::this_thread::get_id() == masterThreadID) {
        ...
    }
    ...
}

std::thread master(doSomething);
masterThreadID = master.get_id();
...
std::thread slave(doSomething);
...
```

Note that IDs of terminated threads might be reused again.

For further details of class `thread`, see Section 18.3.6, page 979.

### 18.2.2. Promises

Now the question arises as to how you can pass parameters and handle exceptions between threads (which also explains how a high-level interface, such as `async()`, is implemented). Of course, to pass values to a thread, you can simply pass them as arguments. And if you need a result, you can pass *return arguments* by reference, just as described for `async()` (see Section 18.1.2, page 958).

However, another general mechanism is provided to pass result values and exceptions as outcomes of a thread: class `std::promise`. A promise object is the counterpart of a *future* object. Both are able to temporarily hold a *shared state*, representing a (result) value or an exception. While the future object allows you to retrieve the data (using `get()`), the promise object enables you to provide the data (by using one of its `set_...()` functions). The following example demonstrates this:

**Click here to view code image**

```
// concurrency/promise1.cpp

#include <thread>
#include <future>
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>
#include <functional>
#include <utility>

void doSomething (std::promise<std::string>& p)
{
    try {
        // read character and throw exception if 'x'
        std::cout << "read char ('x' for exception): ";
        char c = std::cin.get();
        if (c == 'x') {
            throw std::runtime_error(std::string("char ")+c+" read");
        }
        ...
        std::string s = std::string("char ") + c + " processed";
        p.set_value(std::move(s));    // store result
    }
    catch (...) {
        p.set_exception(std::current_exception());    // store exception
    }
}

int main()
{
    try {
        // start thread using a promise to store the outcome
        std::promise<std::string> p;
        std::thread t(doSomething,std::ref(p));
        t.detach();
        ...

        // create a future to process the outcome
        std::future<std::string> f(p.get_future());

        // process the outcome
        std::cout << "result: " << f.get() << std::endl;
    }
    catch (const std::exception& e) {
        std::cerr << "EXCEPTION: " << e.what() << std::endl;
    }
    catch (...) {
        std::cerr << "EXCEPTION " << std::endl;
    }
}
```

After including `<future>`, where promises also are declared, you can declare a promise object, specialized for the value to hold or return (or `void` if none):

```
std::promise<std::string> p; // hold string result or exception
```

The promise internally creates a *shared state* (see Section 18.3, page 973), which can be used here to store a value of the corresponding type or an exception, and can be used in a future object to retrieve this data as the outcome of the thread.

This promise is then passed to a task running as a separate thread:

```
std::thread t(doSomething,std::ref(p));
```

By using `std::ref()` (see Section 5.4.3, page 132), we ensure that the promise is passed by reference so that we can manipulate its state (copying is not possible for promises).

Now, inside the thread, we can store either a value or an exception by calling `set_value()` or `set_exception()`, respectively:

**Click here to view code image**

```
void doSomething (std::promise<std::string>& p)
{
    try {
        ...
        p.set_value(std::move(s));    // store result
    }
    catch (...) {
        p.set_exception(std::current_exception());     // store exception
    }
}
```

To store an exception, the convenience function `std::current_exception()`, which is defined in `<exception>`, is used (see Section 4.3.3, page 52). It yields the currently handled exception as type `std::exception_ptr` or `nullptr` if we currently do not handle an exception. The promise object stores this exception internally.

The moment we store a value or an exception in a *shared state*, it becomes *ready*. Thus, you can now retrieve its value somewhere else. But for the retrieval, we need a corresponding future object sharing the same *shared state*. For this reason, inside `main()`, by calling `get_future()` for the promise object we create a future object, which has the usual semantics introduced in Section 18.1, page 946. We could also have created the future object before starting the thread:

```
std::future<std::string> f(p.get_future());
```

Now, with `get()`, we either get the stored result or the stored exception gets rethrown (internally, calling `std::rethrow_exception()` for the stored `exception_ptr`):

```
f.get(); // process the outcome of the thread
```

Note that `get()` blocks until the *shared state* is *ready*, which is exactly the case when `set_value()` or `set_exception()` was performed for the promise. It does *not* mean that the thread setting the promise has ended. The thread might still perform other statements, such as even store additional outcomes into other promises.

If you want the *shared state* to become *ready* when the thread really ends — to ensure the cleanup of thread local objects and other stuff before the result gets processed — you have to call `set_value_at_thread_exit()` or `set_exception_at_thread_exit()` instead:

Click here to view code image

```
void doSomething (std::promise<std::string>& p)
{
    try {
        ...
        p.set_value_at_thread_exit(std::move(s));
    }
    catch (...) {
        p.set_exception_at_thread_exit(std::current_exception());
    }
}
```

Note that using promises and futures is not limited to multithreading problems. Even in single-threaded applications, we could use a promise to hold a result/value or an exception that we want to process later by using a future.

Note also that we can't store both a value and an exception. Any attempt to do this would result in a `std::future_error` with the error code `std::future_errc::promise_already_satisfied` (see Section 4.3.1, page 43).

For further details of class `promise`, see Section 18.3.4, page 977.

### 18.2.3. Class packaged_task<>

`async()` gives you a handle to deal with the outcome of a task that you try to start immediately in the background. Sometimes, however, you need to process the outcome of a background task that you don't necessarily start immediately. For example, another instance, such as a thread pool, might control when and how many background tasks run simultaneously. In this case, instead of

Click here to view code image

```
double compute (int x, int y);

std::future<double> f = std::async(compute,7,5);   // try to start a background task
...
double res = f.get();   // wait for its end and process result/exception
```

you can program:

Click here to view code image

```
double compute (int x, int y);

std::packaged_task<double(int,int)> task(compute);   // create a task
std::future<double> f = task.get_future();           // get its future
...
task(7,5);               // start the task (typically in a separate thread)
...
double res = f.get();    // wait for its end and process result/exception
```

where the task itself is usually, but not necessarily, started in a separate thread.

Thus, class `std::packaged_task<>`, also defined in `<future>`, holds both the functionality to perform and its possible outcome (the so-called *shared state* of the functionality; see Section 18.3, page 973).

For further details of class `packaged_task`, see Section 18.3.5, page 977.