**Low-Level Interface: Threads and Promises**

The C++ standard library provides a low-level interface also to start threads and deal with them.

# Class std::thread

To start a thread, 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:

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) together with possible additional arguments.

Note: 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.

# std::async vs std::thread

This is a low-level interface, what this interface does not provide compared to async():

1. 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 with the error code resource\_unavailable\_try\_again.
2. There is no interface to process the result or outcome of the thread. The only thing you can get is a unique thread ID.
3. If an exception occurs that is not caught inside the thread, the program immediately aborts, calling std::terminate().
4. To pass exceptions to a context outside the thread exception\_ptrs have to be used.
5. To wait for the end of the thread call join() or to detach from the thread started to let it run in the background without any control call 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().
6. If you let the thread run in the background and main() ends, all threads are terminated abruptly.

# Example

#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 + 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

}

cout << "Enter a character : " << endl;

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;

}

return 0;

}

Output:

- started fg thread 108414260070144

- detach started bg thread 108414250735360

- detach started bg thread 108414241584896

- detach started bg thread 108414233163520

- detach started bg thread 108414224287488

- detach started bg thread 108414214948608

Enter a character :

+abcdebdb+aced+aebd++cacedcbaabcdeabcdbddaecadbecbceaeeT

- join fg thread 108414260070144

main() and doSomething() have corresponding try-catch clauses for the following reasons:

1. In main(), creating a thread might throw a std::system\_error with the error code resource\_unavailable\_try\_again.
2. In doSomething(), started as std::thread, any uncaught exception would cause the program to terminate.

# Beware of Detached Threads

Detached threads can easily become a problem if they use nonlocal resources.

**Problem:** 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.

**Solution:** 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.

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.

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

1. Detached threads should prefer to access local copies only.
2. 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 , 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.
* 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.

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.

Ideally, you should use notify\_all\_at\_thread\_exit() to set the condition variable to ensure that all thread local variables are destructed.

# Thread IDs

The program prints thread IDs provided either by the thread object or inside a thread, using namespace this\_thread (also provided by <thread>).

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".

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.

**Note:**

1. 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.
2. IDs of terminated threads might be reused again.

# Promises

class std::promise is provided to pass result values and exceptions as outcomes of a thread. 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).

**Example**

#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 exceptiopn 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 (...) {

std::cout << "Store EXCEPTION " << std::endl;

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::cout << "EXCEPTION: " << e.what() << std::endl;

}

catch (...) {

std::cout << "EXCEPTION " << std::endl;

}

return 0;

}

Output - 1:

A

result: char A processed

Output - 2:

x

Store EXCEPTION

EXCEPTION: char x read

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:

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**

1. Using promises and futures is not limited to multithreading problems. Even in singlethreaded applications, we could use a promise to hold a result/value or an exception that we want to process later by using a future.
2. Both a value and an exception cannot be stored. Any attempt to do this would result in a std::future\_error with the error code std::future\_errc::promise\_already\_satisfied.

# Class packaged\_task<>

Sometimes, 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 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:

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).

# std::thread in Detail

An object of class std::thread is provided to start and represent a thread.

Operations available for class thread are:

|  |  |
| --- | --- |
| Operation | Effect |
| thread t | Default constructor; creates a nonjoinable thread object |
| thread t(f, ...) | Creates a thread object, representing f started as thread (with additional args), or throws std::system\_error |
| thread t(rv) | Move constructor; creates a new thread object, which gets the state of rv, and makes rv nonjoinable |
| t.~thread() | Destroys \*this; calls std::terminate() if the object is joinable |
| t = rv | Move assignment; move assigns the state of rv to t or calls std::terminate() if t is joinable |
| t.joinable() | Yields true if t has an associated thread (is joinable) |
| t.join() | Waits for the associated thread to finish (throws std::system\_error if the thread is not joinable) and makes the object nonjoinable |
| t.detach() | Releases the association of t to its thread while the thread continues (throws std::system\_error if the thread is not joinable) and makes the object nonjoinable |
| t.get\_id() | Returns a unique std::thread::id if joinable or std::thread::id() if not |
| t.native\_handle() | Returns a platform-specific type native\_handle\_type for nonportable extensions |

If the thread object has an associated thread, it is said to be joinable. In that case, joinable() yields true, and get\_id() yields a thread ID that differs from std::thread::id().

Note that detached threads should not access objects whose lifetimes have ended. This implies the problem that when ending the program, you have to ensure that detached threads don’t access global/static objects).

In addition, class std::thread provides a static member function to query a hint for the possible number of parallel threads:

**unsigned int std::thread::hardware\_concurrency ()**

* Returns the number of possible threads.
* This value is just a hint and does not guarantee to be exact.
* Returns 0 if the number is not computable or well defined.

# std::promise in Detail

An object of class std::promise is provided to temporarily hold a (return) value or an exception.

In general, a promise can hold a shared state. The shared state is said to be ready if it holds a value or an exception.

Operations available for class promise are:

|  |  |
| --- | --- |
| Operation | Effect |
| promise p | Default constructor; creates a promise with shared state |
| promise p(allocator\_arg, alloc) | Creates a promise with shared state, which uses alloc as allocator |
| promise p(rv) | Move constructor; creates a new promise object, which gets the state of rv and removes the shared state from rv |
| p.~promise() | Releases the shared state and if it is not ready (no value or exception), stores a std::future\_error exception with condition broken\_promise |
| p = rv | Move assignment; move assigns the state of rv to p and if p was not ready, stores a std::future\_error exception with condition broken\_promise there |
| swap(p1,p2) | Swaps states of p1 and p2 |
| p1.swap(p2) | Swaps states of p1 and p2 |
| p.get\_future() | Yields a future object to retrieve the shared state (outcome of a thread) |
| p.set\_value(val) | Sets val as (return) value and makes the state ready (or throws std::future\_error) |
| p.set\_value\_at\_thread\_exit(val) | Sets val as (return) value and makes the state ready at the end of the current thread (or throws std::future\_error) |
| p.set\_exception(e) | Sets e as exception and makes the state ready (or throws std::future\_error) |
| p.set\_exception\_at\_thread\_exit(e) | Sets e as exception and makes the state ready at the end of the current thread (or throws std::future\_error) |

You can call get\_future() only once. In general, if no shared state is associated, a std::future\_error with the error code std::future\_errc::no\_state might be thrown.

All member functions that set the value or exception are thread safe.

# std::packaged\_task in Detail

Class std::packaged\_task<> is provided to hold both some functionality to perform and its outcome, which might be a return value or an exception raised by the functionality.

Operations available for class packaged\_task are:

|  |  |
| --- | --- |
| Operation | Effect |
| packaged\_task pt | Default constructor; creates a packaged task with no shared state and no stored task |
| packaged\_task pt(f) | Creates an object for the task f |
| packaged\_task pt(alloc, f) | Creates an object for the task f using allocator alloc |
| packaged\_task pt(rv) | Move constructor; moves the packaged task rv (task and state) to pt (rv has no shared state afterward) |
| pt.~packaged\_task() | Destroys \*this (might make shared state ready) |
| pt = rv | Move assignment; move assigns the packaged task rv (task and state) to pt (rv has no shared state afterward) |
| swap(pt1,pt2) | Swaps packaged tasks |
| pt1.swap(pt2) | Swaps packaged tasks |
| pt.valid() | Yields true if pt has a shared state |
| pt.get\_future() | Yields a future object to retrieve the shared state (outcome of the task) |
| pt(args) | Calls the task (with optional arguments) and makes the shared state ready |
| pt.make\_ready\_at\_thread\_exit(args) | Calls the task (with optional arguments) and at thread exit makes the shared state ready |
| pt.reset() | Creates a new shared state for pt (might make the old shared state ready) |

Trying to call the task or get\_future() if no state is available throws a std::future\_error with the error code std::future\_errc::no\_state.

Calling get\_future() a second time throws an exception of type std::future\_error with the error code std::future\_errc::future\_already\_retrieved.

Calling the task a second time throws a std::future\_error with the error code std::future\_errc::promise\_already\_satisfied.

The destructor and reset() abandon the shared state, which means that the packaged task releases the shared state and, if the shared state was not ready yet, makes the state ready with a std::future\_error with error code std::future\_errc::broken\_promise stored as outcome.

As usual, the make\_ready\_at\_thread\_exit() function is provided to ensure the cleanup of local objects and other stuff of a thread ending the task before the result gets processed.

# Namespace this\_thread

For any thread, including the main thread, <thread> declares namespace std::this\_thread, which provides the thread-specific global functions listed as following:

|  |  |
| --- | --- |
| Operation | Effect |
| this\_thread::get\_id() | Yields the ID of the current thread |
| this\_thread::sleep\_for(dur) | Blocks the thread for duration dur |
| this\_thread::sleep\_until(tp) | Blocks the thread until timepoint tp |
| this\_thread::yield() | Hint to reschedule to the next thread |

The operation this\_thread::yield() is provided to give a hint to the system that it is useful to give up the remainder of the current thread’s time slice so that the runtime environment can reschedule to allow other threads to run.

**Example**

To give up control when you wait or “poll” for another thread or an atomic flag to be set by another thread

while (!readyFlag) { // loop until data is ready

std::this\_thread::yield();

}

As another example, when you fail to get a lock or a mutex while locking multiple locks/mutexes at a time, you can make the application faster by using yield() prior to trying the locks/mutexes in a different order.

# END