# C++ Course 10 : Concurrency.

2020 by Oleksiy Grechnyev

# Concurrency in C++

Before C++ 11: Windows vs Posix (Unix) threads. Boost threads.

C++ 11: Concurrency in the standard library, cross-platform.

1. **async()** + **future** : Task-based approach
2. **thread** : Thread-based approach (lower level)
3. **mutex, atomic** : Protecting shared data
4. **promise, packaged_task** : Using **future** in a **thread**
5. **condition_variable** : A monitor with **wait**/ **notify** syntax.

C++ Concurrency in Action: Practical Multithreading by Anthony Williams

Physical thread : Implemented by the CPU

Logical thread : Implemented by the OS

How to enable C++ threads (links thread library on Unix/Linux)?

Don't try to put "**-pthread**" anywhere. The proper cross-platform CMake way is:

```
find_package(Threads)
target_link_libraries(${PROJECT_NAME} ${CMAKE_THREAD_LIBS_INIT})
```

# sleep_for, sleep_until and yield

**this_thread::...** : useful functions for the current (or only) thread.

Sleep for a while : (**std::chrono::duration**)

```
this_thread::sleep_for(milliseconds(20));
this_thread::sleep_for(seconds(2));
using namespace std::chrono_literals;
this_thread::sleep_for(100ms);  // 100 milliseconds
```

Sleep until a time point : (**std::chrono::time_point**)

```
this_thread::sleep_until(tp);
```

Sleep a bit (undefined time, until next time quantum ?) :

```
this_thread::yield();
```

Print current thread id :

```
cout << this_thread::get_id() << " : starting !\n";
```

# future and async() : Launch a task, wait for the result

"Future" is programming means an object which *eventually* produces some result.

**future<int>** : Value of type **int** which can be obtained later with **get()**

**async()** : Run a task synchronously or asynchronously

**async()** can receive additional arguments (3, 7) here

But I prefer to use lambda wrappers with capture to avoid the arguments!

```
future<int> f = async([](int x, int y)->int{
   return x*y;
}, 3, 7);     // Calculate 3*7
...
f.wait();     // Optional
cout << "f.get() = " << f.get() << endl;    // Get int the result
```

**get()** : Wait to the task to finish and return the result

**wait()** : Wait for the task to finish (not really needed if we use **get()**)

**get()** can be only called once for each future !

For **future<void>** (task with no result), **get()** is preferable to **wait()** (because of exceptions)

# Asynchronous (launch::async) vs deferred (launch::deferred)

Asynchronous launch (in a separate thread) :

```
future<int> f = async(launch::async, []{
        cout << "launch::async !!!" << endl;
});
```

Starts immediately, **f.wait()/f.get()** wait for the task to finish

Deferred launch (in the same thread, no concurrency) :

```
future<int> f = async(launch::deferred, []{
        cout << "launch::deferred !!!" << endl;
});
```

Starts, runs and finishes when **f.wait()/f.get()** is called !

Default launch : equivalent to :

```
future<int> f = async(launch::deferred | launch::async, []{ ... });
```

**async()** decides itself what to do (usually this means asynchronous launch).

But use **launch::async** for a *guaranteed* multithreading!

# What if a future is destroyed before wait()/get() is called ?

```
{
        future<int> f = async(...);
}   // f is destroyed here
```

What does the destructor of a **future** do ?

Asynchronous tasks :

Wait for the task to finish, discard the result (implicit **wait**).

Deferred tasks :

The task is never started at all !

This is only for the futures created by **async()** !

Other futures : do nothing at all.

Conclusion : Use **wait()/get()** explicitly and be in control !

Note: futures can be moved (and put into a **vector**), but NOT copied !

# Running tasks in parallel using async

```
// Print a char and sleep a bit (100 times)
auto lamChar = [](char c)->void{
     cout << this_thread::get_id() << " : starting !\n";
     for (int i = 0; i< 100; ++i) {
          cout << c;
          this_thread::sleep_for(milliseconds(1));    // Sleep a bit
     }
};

// Run 2 tasks in parallel
future<void> fA = async(launch::async, lamChar, 'A');
future<void> fB = async(launch::async, lamChar, 'B');
// The last one will never start, no get/wait !
future<void> fC = async(launch::deferred, lamChar, 'C');
```

```
6 : starting !
7 : starting !
BAABBAABABBABABABABAABABABBABABABAABBABABABABAABABABBABAABABBABABAABBABAABABAB
ABABABBABABAABABBABBAABABABABBABABABAABABBABAABABABABBAABBABABABAABABABABBABAB
ABAABABBABABAABABABABABABBAABABBABABAABBABAA
```

# wait_for() : a timed wait on a future f

**future_status result = f.wait_for(milliseconds(100));** // f == some future

Returns the status *after* waiting up to 100 ms :

**future_status::deferred** : Deferred task, not started yet

**future_status::ready** : The task has finished

**future_status::timeout** : The task is running, not finished yet

**wait_for()** does NOT start deferred tasks !

Use **wait_for(seconds(0))** to check on a task :

```
while (f.wait_for(seconds(0)) == future_status::timeout) {
        cout << "Still waiting ..." << endl;
        this_thread::sleep_for(milliseconds(100));
}
```

Of course we could have used **wait_for(milliseconds(100))** ...

# future, async() and exceptions

```
future <int> f = async(launch::async, [](int x, int y)->int{
    if (x<0 || y<0) throw runtime_error("The user is an IDIOT !!!");
    return x*y;     // Suppose we don't like negative numbers ...
}, 7, -3);

cout << "Task started ..." << endl;
this_thread::sleep_for(milliseconds(10));  // Give it some time
cout << "Before get ..." << endl;
cout << "f.get() = " <<  f.get() << endl;
cout << "After get ..." << endl;
```

An exception is thrown inside an **async()** task !

What happens ?

# future, async() and exceptions

```
future <int> f = async(launch::async, [](int x, int y)->int{
    if (x<0 || y<0) throw runtime_error("The user is an IDIOT !!!");
    return x*y;     // Suppose we don't like negative numbers ...
}, 7, -3);

cout << "Task started ..." << endl;
this_thread::sleep_for(milliseconds(10));  // Give it some time
cout << "Before get ..." << endl;
cout << "f.get() = " <<  f.get() << endl;    // Thrown by get()!!!
cout << "After get ..." << endl;
```

An exception is thrown inside an **async()** task !
It is automatically caught and remembered in the **future** !

**get()** rethrows the exception.
**wait()** does not rethrow the exception, use **get()** even for **future<void>** !

# Threads : Lower level C++ concurrency

**std::thread** is a handle to a software thread.

```
thread t([]{
       cout << " A thread !" << endl;
});  // Start a thread
...   // Thread runs in parallel to the main thread
t.join();    // Wait to finish
```

You must **join()** or **detach()** every software thread!

**join()** : Wait for the thread to finish (GOOD style)

**detach()** : Detach software thread from the **std::thread** object (BAD style)

Otherwise the **std::thread** destructor stops the program!

**std::thread** does not return any results.

It is similar to **async()** + **future<void>**, but lower level.

But: Uncaught exception in a thread terminates the program.

Note: **std::thread** can be moved, not copied!

# Run 4 threads in parallel with thread

```
// Print a char and sleep a bit (100 times)
auto lamChar = [](char c)->void{
    cout << this_thread::get_id() << " : starting !\n";
    for (int i = 0; i< 100; ++i) {
        cout << c;
        this_thread::sleep_for(milliseconds(1));    // Sleep a bit
    }
};
// Run 4 threads in parallel
thread tA(lamChar, 'A');    // Join tA, tB, t0, but not tC !
thread tB(lamChar, 'B');
thread tC(lamChar, 'C');
thread t0([]{cout << "IDIOT\n";} );
tC.detach();  // Detach this one from the std::thread handle
cout << "Threads  started ..." << endl;
this_thread::sleep_for(milliseconds(10));
cout << "About to join threads ..." << endl;
tA.join();
tB.join();
t0.join();
```

# Run 4 threads in parallel

```
12 : starting !
14 : starting !
A13 : starting !
CThreads  started ...
BIDIOT
About to join threads ...CAB
BCACBACBABACBACABCACBABCACBBACBACACBABCACBACBCBABCABACBACBACBCAABCCBAABCCBACAB
ACBBACABCACBACBABCABCBACABCCBACBACBACABCABABCACBACBACBACBCBABACBCABACBACCABBCA
CBACABCBAACBABCCABABCACBABCACBACBCBABACBACBACBACBACCBABACBACBCABCABCACABACBACB
ACBACBACBBCACABABCACBABCBCABCACBACABCABCBACBABACABCABCACBABC
```

# Passing argument by reference : std::ref()

Both **thread** and **async** use an **std::bind**-like syntax to pass arguments.

To pass arguments by reference, you must wrap them in **std::ref** !

```
void doSomething(int par, int & data) {...}
...
int i, j;
thread t(doSomething, 13, ref(i));
future<void> f = async(doSomething, 14, ref(j));
```

Alternative: use a lambda wrapper (I prefer this !):

```
thread t([&i](){
        doSomething(13, i);
});
future<void> f = async([&j](){
        doSomething(14, j);
});  // DANGER ! Variables i, j can die before the tread finishes, especially with detach()  !!!
```

# Data shared between threads : Data Race !

```
vector<string> data;
thread t1([& data](){ ... });
thread t2([& data](){ ... });
```

If both **t1** and **t2** only *read* data, everything is OK !

If *either* **t1** or **t2** *write* data : DATA RACE = BAD !

C++ standard: the behavior is UNDEFINED !

Stupid junior developers will say:

...I tried it, it worked...

...What can possibly happen...

...I have a primitive type, not a class...

...I saw an example in the internet...

...I wanted to keep it simple...

...My cousin's friend, a senior dev, said that...

...It's all bullshit, in real life everything works...

# Data shared between threads : Data Race !

```
vector<string> data;
thread t1([& data](){ ... });
thread t2([& data](){ ... });
```

If both **t1** and **t2** only *read* data, everything is OK !

If *either* **t1** or **t2** *write* data : DATA RACE = BAD !

C++ standard: the behavior is UNDEFINED !

A good developer will say:

No fooling around !

I want my code reliable !

I will ALWAYS protect my data with **atomic** or **mutex** !

# Data race demo :

```
int result = 0;
auto lam = [& result]{
    for (int i=0; i < 10000; ++i)
        ++result;
};
vector <thread> v;
for (int i=0; i < 100; ++i)
    v.emplace_back(lam);
for (auto & t : v)
    t.join();
cout << "result = " << result << endl;
```

What is the result ???

# Data race demo :

```
int result = 0;
auto lam = [& result]{
    for (int i=0; i < 10000; ++i)
        ++result;
};
vector <thread> v;
for (int i=0; i < 100; ++i)
    v.emplace_back(lam);
for (auto & t : v)
    t.join();
cout << "result = " << result << endl;
```

The result should be 1'000'000 (one million), but ...

Actually it can be anywhere between 900'000 and one million !

And this was for Debug build, can be worse with optimization !

Conclusion : **int** is NOT *thread-safe*.

Even operator **++** is NOT *atomic* (i.e. unbreakable transaction)!

# Are standard types thread safe?

Most C++ types are NOT thread safe, including primitives !
Atomics, mutexes *are* thread-safe.

Especially strings and containers are dangerous ! Crash, corrupt data, memory leak !

I/O streams  (such as **cout**) are technically thread-safe. BUT:
The results often get mixed up when chaining.
Thread 1:
**cout << "a = " << 17 << endl;**
Thread 2:
**cout << "b = " << 20 << endl;**
Can give (for example):
**a=b=1720**

With 2 newlines in the end.

# atomic variables

**atomic<T>** is an atomic wrapper of type **T**. Specialized for most primitive types.

**T** must be *trivially copyable*, so **int** is OK, **string** or **vector** are not!

Basically it means: "primitive" or "struct of primitives".

Example:

```
atomic<int> a1(17);   // Ctor
atomic<int> a2;
a2.store(18);         // Set a value
a2 = 18;              // The same
++a1;                 // Atomic (thread) safe operation for some types
a2 += 3;              // Atomic (thread) safe operation for some types
int i = a1.load();    // Get a value
int i = a1;           // The same
a2 = a1;              // Error ! Atomics cannot be copied !
```

Atomics can be neither copied nor moved !

Using **load()** and **store()** is a good practice

# atomic<int> example :

```
atomic<int> result(0);
auto lam = [& result]{
    for (int i=0; i < 10000; ++i)
        ++result;
};
vector <thread> v;
for (int i=0; i < 100; ++i)
    v.emplace_back(lam);
for (auto & t : v)
    t.join();
cout << "result = " << result.load() << endl;
```

The result is exactly 1'000'000 (one million) !

**atomic<int>** works !

# mutex (MUTual EXclusion)

**mutex** is a lock which protects a shared data variable (usually).
Only one thread can lock the mutex and access shared the variable.
Other threads will then wait until the mutex gets unlocked. Danger: deadlocks!

```
vector<string> data;   // data and m must be captured by ref by >= 2 threads
mutex m;               // mutex protecting shared variable data
```

In thread 1:

```
m.lock();            // Wait for m to unlock (if locked), then lock
data.push_back(s);   // Exclusive access to data by this thread !
m.unlock();          // Unlock
```

In thread 2:

```
m.lock();            // Wait for m to unlock (if locked), then lock
for (const string & s : data)
      cout << s << endl; // Exclusive access to data by this thread !
m.unlock();          // Unlock
```

# Is everything OK with this code ?

```
set<string> data;
mutex m;
...
m.lock();
if (data.empty())
	throw runtime_error("No data !!!");
else if (data.count("QUIT"))
	return -1;
else
	...   // Do something with the data
m.unlock();
```

# Solution lock_guard !

```
set<string> data;
mutex m;
...
{
	lock_guard<mutex> lock(m);	// Lock until '}'
	if (data.empty())
		throw runtime_error("No data !!!");
	else if (data.count("QUIT"))
		return -1;
	else
		...   // Do something with the data
}		// Unlock here
```

When the lock is destroyed, the mutex is unlocked !

Resource acquisition is initialization (RAII) pattern (like **ofstream, unique_ptr**).

Versions which can be moved or copied: **unique_lock<mutex>, shared_lock<mutex>**

# mutex example :

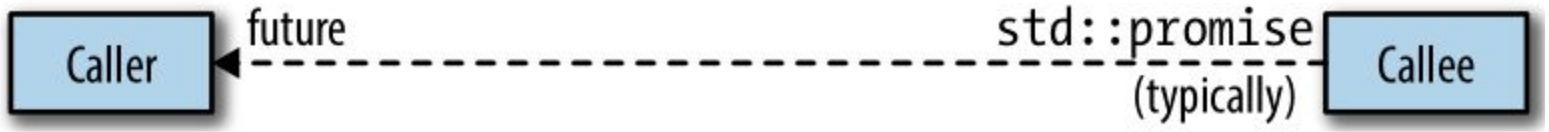
```
int result = 0;
mutex m;
auto lam = [& result, & m]{
    for (int i=0; i < 10000; ++i) {
        lock_guard<mutex> lock(m);    // Lock until '}'
        ++result;
    }                                  // Unlock here
};
vector <thread> v;
for (int i=0; i < 100; ++i)
    v.emplace_back(lam);
for (auto & t : v)
    t.join();
cout << "result = " << result << endl;
```

The result is exactly 1'000'000 (one million) !

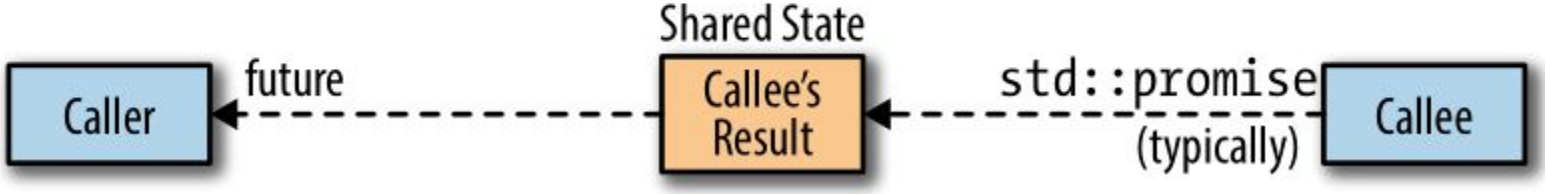
**mutex** works !

# Promises and futures come in pairs

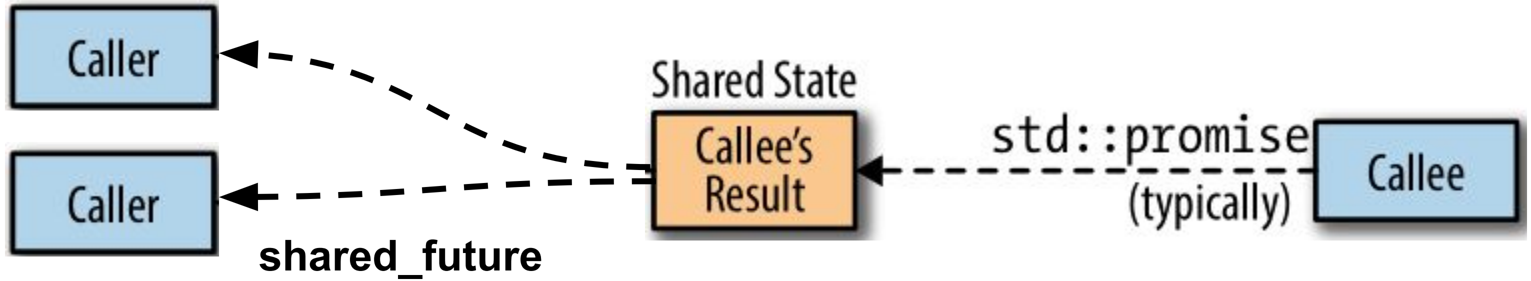
**std::promise** provides a result (or exception) to **std::future**




The result (or exception) is kept in *shared state* until you call **get()**




**shared_future** is a version which can be copied, and you can call **get()** many times



# Using promise and future with threads (without async)

1. Create a **promise** + **future** pair
2. Pass **promise** and/or **future** by reference to threads
3. Store the result/exception into **promise** in one thread : **set_value(), set_exception()**
4. And call **get()** on the **future** in the other one

```
promise<int> p;  // Create a promise/future pair
future<int> f = p.get_future();

// Now we start the thread, capturing promise by ref
thread t([&p](int x, int y)->void{
    // We use set_value() on promise instead of return !!!
    p.set_value(x*y);
}, 3, 7);

// Now we run get() on the future as usual
cout << "f.get() = " << f.get() << endl;
t.join();  // Don't forget to join, before or after get()
```

# Storing exceptions into std::promise : set_exception()

Exceptions in C++ can be of *any type*, not only **std::exception** !
There is a standard wrapper : **std::exception_ptr**.
There are 2 options (**p** = some **promise**, **f** = associated **future**), :

1. Create an **exception_ptr** directly by **make_exception_ptr()** :

```
p.set_exception(make_exception_ptr(runtime_error("The user is an IDIOT !!!")));
```

2. Use **current_exception()** within a **catch** clause:

```
try {
        ...
} catch(...) {
        p.set_exception(current_exception());
}
```

3. Now the exception is thrown in the other thread the moment **f.get()** is executed.

# Using packaged_task in a thread

**packaged_task** is a template similar to **function**, which stores result and exceptions in a **future**

```
auto lam = [](int x, int y)->int{
    if (x<0 || y<0)
        throw runtime_error("The user is an IDIOT !!!");
    else
        return x*y;
};
packaged_task<int(int, int)> pt3(lam), pt4(lam);  // Create 2 tasks
```

**packaged_task** can be launched in a **thread**, get result with **get()** :

```
future<int> f3 = pt3.get_future();      // Create future from packaged_task
thread t3(move(pt3), 3, 7);             // Run task in a thread. Must move !
f3.join();                              // Join thread before or after get()
cout << "f3.get() = " << f3.get() << endl;

future<int> f4 = pt4.get_future();      // Same with pt4
thread t4(move(pt4), 3, -7);            // Wrong input !
f4.join();
cout << "f4.get() = " << f4.get() << endl;  // Exception is thrown here !
```

# Thread interaction 1: flag

```
atomic<bool> stop(false);
auto lam = [&stop]{    // Capture flag by ref !
     int i = 0;
     while (!stop) {
          cout << i++;
          this_thread::sleep_for(milliseconds(50));
     }
};

thread t1(lam), t2(lam);  // Start two threads
this_thread::sleep_for(milliseconds(500));  // Let them run for a while
stop = true;                              // Signal stop
t1.join();
t2.join();
```

Prints (for example) : 0011223344556667788

Other atomics, or mutex-protected data can be used to communicate between threads.
Drawback: you will have to check repeatedly (**while** loop with sleep). Signals?

# Thread interaction 2: condition variables (Ugly !)

A **condition_variable** implements **wait()** and **notify()** logic:

Two threads share a **condition_variable**, a mutex, and other shared variable(s) **flag** or **a**, **b**, **c**.

1. Thread 1 calls **wait(condition)** and waits.
2. Thread 2 calls **notify_one()** or **notify_all()** .
3. Thread 1 wakes up if **condition** is true.

**condition** is typically **flag** or some other expression involving the shared variables.
**wait()** can be used without condition, but it's a BAD practice !

*Spurious wakeup* : Thread 1 can wake without **notify()** ! C++ standard allows it !

Note : **condition_variable** requires a **mutex** and a **unique_lock** to protect shared variables.
Even if you use an atomic **flag**, you still need a **mutex** and a **unique_lock** !!!

Dangerous: **notify()** before **wait()** means waiting forever: FREEZE !

# Condition variables: example

```
vector <string> data;
mutex m;                        // mutex protects both cv and data
condition_variable cv;

thread worker([&data, &m, &cv]{
    unique_lock<mutex> lk(m);     // We must use unique_lock !!!
    cv.wait(lk, [&data]{return !data.empty();} ); // Wait for data
    for (const string &s : data)
        cout << s << " ";
    cout << endl;
});    // Here we release the mutex

this_thread::sleep_for(milliseconds(10));  // Don't notify too soon !
m.lock();
data = {"Karin", "Lucia", "Anastasia"};    // Supply the data
m.unlock();
cv.notify_one();
worker.join();
```

# Thread interaction 3: promise and future for a 1-shot event

```
promise<void> p;    // promise + future pair
future<void> f = p.get_future();

thread t([&f]{
     f.get();              // Wait for the signal
     cout << "One !\n";
     this_thread::sleep_for(milliseconds(10));
     cout << "Two !\n";
     this_thread::sleep_for(milliseconds(10));
     cout << "Three !\n";
});

for (int i = 0; i < 10; ++i) {
     cout << i << endl;
     if (4 == i)
          p.set_value();             // Send the signal
     this_thread::sleep_for(milliseconds(10));
}

t.join();
```

# Library of the day: CTPL thread pool

When you use **thread** or **async()**, a new thread is created every time.
This means thread creation/destruction overheads, especially on Windows.
Bad if you want to run many small parallel tasks repeatedly.
Also you cannot (easily) set a fixed amount of threads + task queue.

Solution : Thread pool.

```
#include "ctpl_stl.h"
ctpl::thread_pool pool(7);  // Create a pool of 7 threads, pool.size() == 7
```

Now these 7 threads are constantly running, until **pool** is destroyed.
They are IDLE until you launch ("push") some tasks. More than 7 tasks will be queued.

Launching tasks is similar to **async()**, returns a **future** object, but note the **threadID** argument !

```
future<long> result = pool.push([...](int threadID)->long {...});
```

Some time later (**get()** waits for the task to finish):

```
cout << "Result = " << result.get() << endl;
```

# CTPL thread pool: Sum numbers from 1 to 100000

```
// Calculate block size
int nThread = pool.size(), bSize = nData / nThread;
if (nData % nThread)  ++bSize;
// Launch the threads
vector<future<long>> partialSums;  // Each thread gives a partial sum
for (int j = 0; j < nThread; ++j) {
        int i1 = j * bSize, i2 = min((j + 1) * bSize, nData) ; // Calculate first:last range
        partialSums.push_back(pool.push([i1, i2, &data](int threadID) -> long {
                long s = 0;       // Calculate a partial sum
                for (int i = i1; i < i2; ++i)  s += data[i];
                return s;
        }));
}
// Wait for the threads to finish and get the total sum
long sum = 0;
for (future<long> & f : partialSums) sum += f.get();
```

# Thank you for your attention !

# title

text