

# **Multi-Threading**

### Multi-Threading in C++

Stefano Quer and Alessandro Savino
Dipartimento di Automatica e Informatica
Politecnico di Torino

#### Introduction

- The header thread defines the class std::thread that can be used to start new threads
  - Using this class is the best way to use platform-independent threads The std::thread class encapsulates many of the details of thread management, making it easier to work with threads compared to using platform-specific APIs like POSIX threads.

```
#include <thread>
```

- Using it may require additional compiler flags
  - For gcc and clang, use -pthread

```
set(CMAKE_CXX_FLAGS
"${CMAKE_CXX_FLAGS} -std=c++14 -pthread")
```

#### **Introduction**

- The library is based on objects of type std::thread
  - ➤ The operator std::thread works with any callable object like a function, an instance of a class, a lambda expression

### Main thread primitives

- The library covers all main functionalities
  - ➤ But ... there is no way to automatically capture the data computed by a thread

Туре	Main characteristics
std::thread t;	Creates a thread object t.
std::thread t(f);	Creates a thread object t associated with the thread function f.
std::thread t(f,p1,p2,); std::thread t{f(p1,p2,)};	Creates a thread object t associated with the thread function f which receives the parameters p1, p2, etc.
t2=std::move(t1);	Move the thread associated to the thread object t1 to object t2. Moves the thread associated with the thread object t1 to object t2. After this operation, t1 no long the thread, and t2 becomes the owner.
t.detach()	Makes the thread t as detached Detaches the thread associated with t. This mean thread can continue to execute independently of the object. After detaching, the thread is no longer joint and thread is no longer in the continue to execute independently of the object.
t.join()	Waits the thread t for joining. Waits for the thread associated with t to finish its execution with the thread associated with th

## **Other primitives**

The thread library also contains useful functions related to starting and stopping threads

Туре	Main characteristics
std::this_thread::sleep_for	Stop the current thread for a given amount of time.
std::this_thread::sleep_until	Stop the current thread until a given point in time.
std::this_thread::yield	Let the operating system schedule another  The term "yield" in the context of std::this_thread::yield refers to a mechanism that allows a voluntarily give up its current time slice or CPU execution time to allow another thread to run
std::this_thread::get_id	Get the (operating-system-specific) id of the current thread identifier is specific to the operating system and can be used to uniquely identified.
std::thread::hardware_concur rency	Reports the actual max number of threads based on your architecture.

```
Function definitions
void f1() {
                           (without and with parameters)
void f2(int a, int b) {
                            Creates an object that
                          does not refer to a thread
std::thread t1;
                                    Starts an object thread that calls f1()
std::thread t2(f1);
                                                   Starts a thread that
std::thread t3(f2, 123, 456);
                                                   calls f2(123, 456)
```

creates a thread object t4 that is associated with the execution of a lambda function. The lambda function itself calls the function f2 with parameters and 456. This is a way to start a thread with a function call and parameters without explicitly defining a separate function.

std::thread t4([] { f2(123, 456); });

Works also with lambda functions

### **Join and Move**

- The member function join can be used to wait for a thread to finish
  - Function join must be called exactly once for each thread
- Standard threads are not copyable, but movable, so they can be used in containers
  - Moving an std::thread transfers all resources associated with the running thread
  - Only the new thread can be joined

Move semantics: Standard threads in C++ are not copyable, meaning you can't make a direct copy of a thread. However, they are movable. This means you can transfer ownership of a thread from one variable to another. When you move a thread, all resources associated with the running thread are transferred to the new owner. The original thread object no longer represents a running thread of execution, and thus can't be joined. Only the new owner of the thread can call join() on it.

```
void f(int &result) {
    ...
  result = ...;
}
```

Function definition (with a paramter by reference)

```
Parameter by reference

int main() {
    ...
    std::thread t (f, std::ref(i));
    ...
    t.join();
}

Variable i will assume the value once the execution is terminated
```

```
std::thread t1([] { std::cout << "Hi\n"; });
std::thread t2 = std::move(t1);
t2.join();
The thread originally started in t1 is joined</pre>
t1 is now empty
```

```
The output operation
                                    shoud be protected
void my_thread () {
  std::cout << "TID" <<
    std::this thread ::get id() << endl;</pre>
main() {
  std::thread t1{mythread};
  std::thread t2{mythread};
  t1.join();
  t2.join();
  return 1;
```

```
void safe_print (int i) {
    ... Enter critical section ...
    std::cout << i;
    ... Exit critical section ...
    return;
}</pre>
For example, use a mutex
(see, unit 06)
```

```
std::vector<std::thread> threadPool;

for (int i = 1; i <= 9; ++i) {
   threadPool.emplace_back([i] { safe_print(i); });
}

for (auto& t : threadPool) {
   t.join();
}</pre>
Digits 1 to 9 are
   printed (unordered)
```