Threads

Let's dive into C++ threads and their lifecycle.

WE'LL COVER THE FOLLOWING

- Creation
- Lifetime
- Arguments
- Operations

To use the multithreading interface of C++ you need the header .

Creation

A thread `std::thread` represents an executable unit. This executable unit, which the thread immediately starts, gets its work package as a callable unit. A callable unit can be a function, a function object or a lambda function:

```
//...
#include <iostream>
#include <thread>
//...
using namespace std;
void helloFunction(){
  cout << "function" << endl;
}
class HelloFunctionObject {
public:
  void operator()() const {
    cout << "function object" << endl;
  }
};

int main{
  thread t1(helloFunction); // function

  HelloFunctionObject helloFunctionObject;
  thread t2(helloFunctionObject); // function object

  thread t3([]{ cout << "lambda function"; }); // lambda function
}
```

Lifetime

The creator of a thread has to take care of the lifetime of its created thread. The executable unit of the created thread ends with the end of the callable. Either the creator is waiting until the created thread `t` is done `(t.join())` or the creator detaches itself from the created thread: `t.detach()`. A thread `t` is joinable if no operation `t.join()` or `t.detach()` was performed on it. A joinable thread calls in its destructor the exception `std::terminate`, and the program terminates.

```
//...
#include <thread>
//...
int main(){
    thread t1(helloFunction); // function

    HelloFunctionObject helloFunctionObject;
    thread t2(helloFunctionObject);// function object

    thread t3([]{ cout << "lambda function"; }); // lambda function
    t1.join();
    t2.join();
    t3.join();
}
```

A thread that is detached from its creator is typically called a daemon thread because it runs in the background.

> **ℹ Move threads with caution**
> You can move a callable from one thread to another.
>
> ```
> #include <thread>
> //...
> std::thread t([]{ cout << "lambda function"; }); std::thread t2;
> t2= std::move(t);
> std::thread t3([]{ cout << "lambda function"; });
> t2= std::move(t3); // std::terminate
> ```
>
> By performing `t2= std::move(t)` thread `t2` has the callable of thread `t`. Assuming thread `t2` already had a callable and is joinable the C++ runtime would call `std::terminate`. This happens exactly in `t2=std::move(t3)` because `t2` neither executed `t2.join()` nor

> `t2.detach()` before.

Arguments

A `std::thread` is a variadic template. This means in particular that it can get an arbitrary number of arguments by copy or by reference. Either the callable or the thread can get the arguments. The thread delegates them to the callable: `tPerCopy2` and `tPerReference2`.

```
#include <thread>
//...
using namespace std;

void printStringCopy(string s){
  cout << s;
}
void printStringRef(const string& s){
  cout << s;
}

int main(){
  string s{"C++"};

  thread tPerCopy([=]{ cout << s; });
  thread tPerCopy2(printStringCopy, s);

  tPerCopy.join();
  tPerCopy2.join();

  thread tPerReference([&]{ cout << s; });
  thread tPerReference2(printStringRef, s);
  tPerReference.join(); tPerReference2.join();
}
```

The first two threads get their arguments by copy, the second two by reference.

> **i By default threads should get their arguments by copy**

```
//...
#include <thread>
//...
using std::this_thread::sleep_for;
using std::this_thread::get_id;

struct Sleeper{
  Sleeper(int& i_):i{i_}{};
  void operator() (int k){
    for (unsigned int j= 0; j <= 5; ++j){
      sleep_for(std::chrono::milliseconds(100));
```

```
      i += k;
     }
   std::cout << get_id();

  }
private:
    int& i;
};

int main(){
  int valSleeper= 1000;
  // undefined behaviour
  std::thread t(Sleeper(valSleeper), 5);
  t.detach();
  std::cout << valSleeper; // undefined behaviour
}
```

This program snippet has two pieces of undefined behavior. First the lifetime of `std::cout` is bound to the lifetime of the main thread. Second, the created thread gets its variable `valSleeper` by reference. The issue is that the created thread lives longer than its creator, therefore `std::cout` and `valSleeper` lose their validity if the main thread is done.

Operations

You can perform many operations on a thread `t`.

Method	Description
`t.join()`	Waits until thread t has finished its executable unit.
`t.detach()`	Executes the created thread t independent of the creator.
`t.joinable()`	Checks if thread t supports the calls join or detach.
`t.get_id()` and `std::this_thread::get_id()`	Returns the identity of the thread.
`std::thread::hardware_concurrency()`	Indicates the number of threads that can be run in parallel.

`std::this_thread::sleep_until(absTime)`	Puts the thread t to sleep until time absTime.
`std::this_thread::sleep_for(relTime)`	Puts the thread t to sleep for the duration relTime.
`std::this_thread::yield()`	Offers the system to run another thread.
`t.swap(t2)` and `std::swap(t1, t2)`	Swaps the threads

You can only call `t.join()` or `t.detach()` once on a thread `t`. If you attempt to call these more than once, you get the exception `std::system_error`. `std::thread::hardware_concurrency` returns the number of cores or 0 if the runtime cannot determine the number. The `sleep_until` and `sleep_for` operations need a time point or a time duration as an argument. Threads cannot be copied but can be moved. A swap operation performs a move when possible.

```
#include <iostream>
#include <thread>
//...
using std::this_thread::get_id;

int main(){
    std::thread::hardware_concurrency();
    std::thread t1([]{ get_id(); });
    std::thread t2([]{ get_id(); });
    std::cout << t1.get_id() << std::endl;
    std::cout << t2.get_id() << std::endl;

    t1.swap(t2);

    std::cout << t1.get_id() << std::endl;
    std::cout << t2.get_id() << std::endl;
    std::cout << get_id();
    t1.join();
    t2.join();
}
```