Threads

This lesson will discuss the basic building block of multithreaded programs - threads.

WE'LL COVER THE FOLLOWING

Introduction

C++ has used a multithreading interface since C++11. This interface contains all the basic building blocks for creating multithreaded programs. These are called ***threads***, which are synchronization primitives for shared data such as ***mutexes*** and ***locks***, ***thread-local data***, synchronization mechanism for threads such as ***condition variables***, and ***tasks***. These ***tasks*** are usually called ***promises*** and ***futures***, and they provide a higher level of abstraction than native threads.

Creation of Threads

To launch a thread in C++, you must include the `<thread>` header.

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 is an entity that behaves like a function.

A callable unit can be a

- **Function**

```
std::thread t(function);
```

- **Function Object**

```
std::thread t(FunctionObject());
```

- **Lambda Expression**

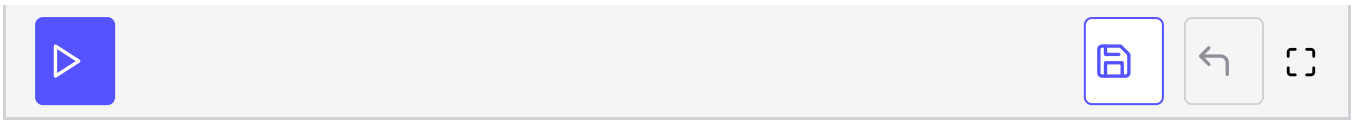
```
std::thread t([]{ std::cout << "I'm running" << std::endl; });
```

```
#include<thread>
#include<iostream>

int main(){
  std::string s{"C++11"};

  std::thread t1([=]{ std::cout << s << std::endl;});
  t1.join();

  std::thread t2([&]{ std::cout << s << std::endl;});
  t2.detach();
}
```



Lifetime of Threads

The parent has to take care of its children. This simple principle has big consequences for the lifetime of a thread. This small program starts a thread that displays its identity.

```
// threadWithoutJoin.cpp

#include <iostream>
#include <thread>

int main(){

  std::thread t([]{std::cout << std::this_thread::get_id() << std::endl;});

}
```

The program will not print the ID.

What's the reason for that exception?

`join` and `detach`

The lifetime of a created thread `t` ends with its callable unit. The creator has two choices.

1. It can wait until its child is done: `t.join()`.
2. It can detach itself from its child: `t.detach()`.

A `t.join()` call is useful when the subsequent code relies on the result of the calculation performed in the thread. `t.detach()` permits the thread to execute independently from the thread handle `t`; The detached thread will run for the lifetime of the executable. Typically, you will use a detached thread for a long-running background service such as a server.

A thread `t` with a callable unit is called joinable if neither a `t.join()` nor a `t.detach()` call occurred. Note: you can create threads without a callable unit. The destructor of a joinable thread throws the `std::terminate` exception. This explains why the program execution of `threadWithoutJoin.cpp` terminated with an exception. If you invoke `t.join()` or `t.detach()` more than once on a thread `t`, you get a `std::system_error` exception.

The solution to this problem is quite simple: call `t.join()`

```
// threadWithJoin.cpp

#include <iostream>
#include <thread>

int main(){

  std::thread t([]{std::cout << std::this_thread::get_id() << std::endl;});

  t.join();

}
```

Argument of Threads

A thread, such as any arbitrary function, can get its arguments by copy, by move, or by reference. `std::thread` is a variadic template. This means it takes an arbitrary number of arguments.

In that case, when your thread gets its data by reference, you must be extremely careful about the lifetime of the arguments; not doing so may result in an undefined behavior.

Copy or Reference

Let's take a look at a small code snippet to demonstrate this operation:

```
#include<thread>
#include<iostream>

int main(){
  std::string s{"C++11"};
  // by copy
  std::thread t1([=]{ std::cout << s << std::endl;});
  t1.join();
  // by reference
  std::thread t2([&]{ std::cout << s << std::endl;});
  t2.detach();
}
```

Thread `t1` gets its argument by copy; thread `t2` by reference.

Methods of Threads

Here is the interface of `std::thread t` in a concise table. For additional details, please refer to cppreference.com.

Method	Description
`t.join()`	Waits until thread `t` has finished its executable unit.
	Executes the created thread `t`

`t.detach()`	independently of the creator.
`t.joinable()`	Returns `true` if thread `t` is still joinable.
`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 concurrently.
`std::this_thread::sleep_until(absTime)`	Puts thread `t` to sleep until the time point `absTime`.
`std::this_thread::sleep_for(relTime)`	Puts thread `t` to sleep for the time duration `relTime`.
`std::this_thread::yield()`	Enables the system to run another thread.
`t.swap(t2)` and `std::swap(t1, t2)`	Swaps the threads.

> 💡 The arguments of the `sleep` methods are time objects.

`std::thread::hardware_concurrency` returns the number of cores. It returns 0 if the runtime cannot determine the number of threads, according to the C++ standard. The `sleep_until` and `sleep_for` operations need a ***time point*** or a ***time duration*** as an argument.

> **Access to the system-specific implementation**
>
> The C++11 threading interface is a wrapper around the underlying implementation. You can use the method `native_handle` to get access to the system-specific implementation. This also holds true for threads, mutexes and condition variables.

> mutexes and condition variables.

Threads cannot be copied but can be moved. The swap method performs a move when possible.

The examples in the following lesson will build on your understanding of the topic.