- Example

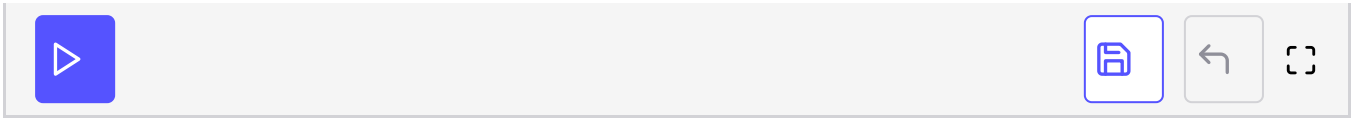
An example of thread-local data.

WE'LL COVER THE FOLLOWING

Example

```
// threadLocal.cpp

#include <iostream>
#include <string>
#include <mutex>
#include <thread>

std::mutex coutMutex;

thread_local std::string s("hello from ");

void addThreadLocal(std::string const& s2){

  s += s2;
  // protect std::cout
  std::lock_guard<std::mutex> guard(coutMutex);
  std::cout << s << std::endl;
  std::cout << "&s: " << &s << std::endl;
  std::cout << std::endl;

}

int main(){

  std::cout << std::endl;

  std::thread t1(addThreadLocal,"t1");
  std::thread t2(addThreadLocal,"t2");
  std::thread t3(addThreadLocal,"t3");
  std::thread t4(addThreadLocal,"t4");

  t1.join();
  t2.join();
  t3.join();
  t4.join();
```

```
}
```



Explanation

- By using the keyword `thread_local` in line 10, the thread-local string `s` is created. Threads `t1` - `t4` (lines 27 - 30) use the function `addThreadLocal` (lines 12 - 21) as their work package.
- Threads get the strings `t1` to `t4` respectively as their arguments, and add them to the thread-local string `s`. In addition, `addThreadLocal` displays the address of `s` in line 18.
- The output of the program shows it implicitly in line 17 and explicitly in line 18. The thread-local string is created for each string `s`. First, each output shows a new thread-local string. Second, each string `s` has a different address.

> **From a Single-Threaded to Multithreaded Program**
>
> Thread-local data helps to port a single-threaded program to a multithreaded environment. If the global variables are thread-local, there is a guarantee that each thread will get its own copy of the data. Accordingly, there is no shared mutable state to cause a data race or undefined behavior.

For further information, read thread_local.

In the next lesson, we will discuss condition variables in the context of concurrency in C++.