Thread-Safe Initialization of Data

In this lesson, we will learn about the three ways for thread-safe initialization of data.

WE'LL COVER THE FOLLOWING

If the variable is never modified there is no need for synchronization by using an expensive `lock` or an `atomic`. We must ensure that it is initialized in a thread-safe way.

There are **four** ways in C++ to initialize variables in a thread-safe way.

- Constant expressions.
- The function `std::call_once` in combination with the flag `std::once_flag`.
- A static variable with block scope.

Thread-safe initialization in the main-thread

The easiest and fourth way to initialize a variable in a thread-safe way: initialize the variable in the main-thread before we create any child threads.

Constant Expressions

Constant expressions can be evaluated by the compiler at compile time. They are implicitly thread-safe. Placing the keyword `constexpr` in front of a variable makes the variable a constant expression. This constant expression must be initialized immediately.

```
constexpr double pi = 3.14;
```

In addition, user-defined types can also be constant expressions. For those types, there are a few restrictions that must be met in order to be initialized at compile time.

- They must not have virtual methods or a virtual base class.
- Their constructor must be empty and itself be a constant expression.
- Their methods should be callable at compile time and must be constant expressions.

Instances of `MyDouble` satisfy these requirements, which makes it possible to instantiate them at compile time. This instantiation is thread-safe.

```
// constexpr.cpp

#include <iostream>

class MyDouble{
  private:
    double myVal1;
    double myVal2;
  public:
    constexpr MyDouble(double v1,double v2):myVal1(v1),myVal2(v2){}
    constexpr double getSum() const { return myVal1 + myVal2; }
};

int main() {

  constexpr double myStatVal = 2.0;
  constexpr MyDouble myStatic(10.5, myStatVal);
  constexpr double sumStat= myStatic.getSum();
  std::cout << "SumStat: "<<sumStat << std::endl;
}
```

`std::call_once` and `std::once_flag`

By using the `std::call_once` function, we can register a callable. The `std::once_flag` ensures that only one registered function will be invoked. We can also register additional functions via the same `std::once_flag`. Only one function from that group is called.

`std::call_once` obeys the following rules:

- Only one execution of exactly one of the functions is performed. It is undefined which function will be selected for execution. The selected function runs in the same thread as the `std::call_once` invocation that it was passed to.
- No invocation in the group returns before the execution of the selected function completes successfully.
- If the selected function exits via an exception, it is propagated to the caller. Another function is then selected and executed.

Static Variables with Block Scope

Static variables with block scope will be created only once and lazily, meaning that they are created at the moment of usage. This characteristic is the basis of the so-called Meyers Singleton, named after Scott Meyers. This is by far the most elegant implementation of the singleton pattern. With C++11, static variables with block scope have an additional guarantee: they will be initialized in a thread-safe way.

Let's take a look at a few examples regarding thread-safe initialization of data in the next lesson.