Paralleles add and sub

In diesem Beispiel werden wir in `main()` folgendes Lambda `addsub()` von mehreren Threads ausführen lassen. Diese Funktion addiert zu `int shared_int;` 5000 Mal dieselbe Zahl `n`, und subtrahiert sie eben oft wieder, sodass der Anfangswert eigentlich erhalten bleiben muss.

```
int main()
{    int x{};
     auto addsub = [&x](int n)    {
            for (int i{}; i < 500'000; ++i)  x += n;
            for (int i{}; i < 500'000; ++i)  x -= n;
        };
```

In **`add_and_sub-2.cpp`** starten wir 10 Threads wie beim Beispiel 1:

```
    cout << "x = " << x << '\n';

    vector<jthread> worker;
    for (int i{}; i < 9; ++i)
        worker.emplace_back(addsub, i);

    cout << "x = " << x << '\n';      // wir sind zu früh!
```

Hier bekommt man fast immer ein anderes Resultat für `x`, was aber leicht zu erklären ist: Da wir nicht auf das Ende der Threads warten, geben wir `x` wahrscheinlich zu früh aus. Wir müssen also vorher alle `std::thread` joinen bzw, alle `std::jthread` terminieren. Letzteres geht sehr einfach, indem man diese in einen eigenen Scope gibt!
`add_and_sub-3.cpp`:

```
    cout << "x = " << x << '\n';

    {   // ein neuer Scope
        vector<jthread> worker;
        for (int i{}; i < 9; ++i)
            worker.emplace_back(addsub, i);
    }   // am Ende des neuen Scopes warden die Threads automagisch gejoined.

    cout << "x = " << x << '\n';      // wir sind nicht mehr zu früh!
```

Leider sind die Ergebnisse oft immer noch falsch, was man jetzt nicht mehr so leicht erklären kann: Die Ursache ist diesmal ein sogenannter Data-Race auf das gemeinsam genutzte `x`. `+=` wird nämlich im Maschinencode nicht in **eine** Operation übersetzt, sondern läuft in 3 Schritten ab:

1) der aktuelle Wert von `x` wird in die CPU geladen.
2) die Addition wird ausgeführt.
3) der neue Wert von `x` wird zurückgespeichert.

Würde jeder Thread warten, bis der vorige Thread alle 3 Aktionen fertig ausgeführt hat, wäre alles in Ordnung. Leider kann/wird es vorkommen, dass ein Thread sich während der 3 Phasen eines anderen Threads einmischt und so ein falsches Resultat erzeugt.

Die Lösung des Problems liegt darin, obige 3 Schritte von keinem anderen Thread unterbrechen zu lassen, d.h. sie **atomar** auszuführen. Dafür gibt es grundsätzlich 2 Techniken: atomare Objekte und Locking.

Atomare Objekte:

C++ besitzt im Includefile `atomic` Deklarationen von einigen atomaren Datentypen, die für manche ihrer Operatoren eine atomare Ausführung garantieren. Für Integertypen gibt es atomares `++, --, =, +=, -=, *=, …` , für andere Typen leider nur weniger atomare Operationen. Wir ändern also die Definition von `x` ab:
add_and_sub-4.cpp: `atomic<int> x{};`

Dadurch läuft das Programm korrekt aber auch ein wenig langsamer.

Locking:

Hierbei muss man **jeden Zugriff** auf eine geteilte Ressource mit einem Schloss absichern. In der Datei `mutex` findet man verschiedene Schlösser, einfache (`std::mutex`) wie auch zeitgesteuerte! Alle beteiligten Threads müssen das gleiche Schloss verwenden. Wir instanzieren daher vor dem Lambda ein solches Schloss und zeichnen es per Referenz auf!
add_and_sub-5.cpp

```
int main()
{   int x{};
    std::mutex lock_x;
    auto addsub = [&x, &lock_x](auto n)
        {   for (int i{}; i < 500'000; ++i) {
                lock_x.lock();
                x += n;
                lock_x.unlock();
            }
```

…

Beim Aufruf der Methode `lock()` des Schlosses erhält ein Thread Zugang zum weiteren Code, wenn das Schloss nicht versperrt ist. Gleichzeitig wird das Schloss natürlich geschlossen. Ist das Schloss versperrt, so wird der Thread in eine Warteschlange eingereiht, und kommt erst dann wieder zur Ausführung, wenn der Zugang möglich ist.

Achtung, dieser Code ist schlecht!!! Es ist für das Locking extrem wichtig, dass ein Schloss auch zuverlässig wieder freigegeben wird. Der Programmierer könnte das `unlock()` vergessen oder der Thread könnte durch eine Exception daran gehindert werden. Das hätte einen **Deadlock** des gesamten Programmes zur Folge: **`add_and_sub-6.cpp`**

Viel bessere Lösung: C++ empfiehlt den Einsatz eines Schlosshüters **std::scoped_lock**, dem man die Verantwortung für das Schloss überträgt. Dieser schließt im Konstruktor das Schloss und gibt es im Destruktor wieder frei. Da C++ auch bei einer Exception die Destruktoren der Objekte aufruft, wird so das Schloss auch in diesem Fall wieder aufgesperrt.
`add_and_sub-7.cpp:`

```
auto addsub = [&x, &lock_x](auto n)
    {
        for (int i{}; i < 500'000; ++i) {
            std::scoped_lock lg{lock_x};      // Lock Guard sperrt zu
            x += n;
        }                                     // auto-unlock durch den Lock Guard
…
```

Wenn der Thread die Exception selbst auffängt, läuft das Programm jetzt korrekt zu Ende und die übrigen Threads können ihre Arbeit vollständig erledigen.

Ein häufiger Programmierfehler: Man vergisst, dem Lock Guard einen Namen zu geben!
`add_and_sub-8.cpp:`

```
std::scoped_lock{lock_x};        // Lock Guard sperrt zu und sofort wieder auf
```

Da hier nur eine temporäre Lock Guard-Instanz erzeugt wird, sperrt diese zu und sofort wieder (im Destruktor) auf und der eigentliche Zugriff verläuft ungeschützt. Das Programm liefert nun wieder falsche Resultate, obwohl auf den ersten Blick alles richtig aussieht.