UNIVERSITÄT OSNABRÜCK

PD Dr. Thomas Wiemann

10. Übungsblatt zur Vorlesung „Einführung in die Programmiersprache C++“

Wintersemester 2022 / 2023

Aufgabe 1: Functionals (30 Punkte)

Betrachten Sie folgendes Code-Fragment:

```
vector<int> v = {1, 4, 2, 8, 5, 7};

copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
cout << endl;

auto it = remove_if(v.begin(), v.end(),
                    bind(bind(equal_to<int>(),_1, 0),
                              bind(modulus<int>(), _1, 2)));

copy(v.begin(), v.end(), ostream_iterator<int>(cout, " "));
cout << endl;

return 0;
```

Beantworten Sie folgende Fragen schriftlich:

1. Welches Problem soll mit dem vorgelegten Code-Fragment offensichtlich gelöst werden?
2. Erklären Sie die Signatur und Nutzung von `std::bind` im vorliegenden Beispiel. Wie werden die benutzten Funktionen und Argumente verknüpft?
3. Wenn Sie den Code ausführen, werden Sie sehen, dass die Ausgabe nicht den Erwartungen entspricht. Was muss geändert werden, damit die Ausgabe den Erwartungen entspricht.

Aufgabe 2: Thread-Synchronisation (30 Punkte)

In dieser Aufgabe schreiben wir ein Programm, das die Quadratzahlen bis zu einem gegebenen N (hier 20) aufsummiert. Wir gehen davon aus, dass das Quadrieren einer Zahl eine gewisse Zeit benötigt. Also wollen wir das Programm parallelisieren, so dass jeder Thread genau ein Quadrat berechnet und zum globalen Ergebnis hinzufügt. Dazu schreiben wir folgendes Programm, das C++-Threads benutzt:

```
#include <iostream>
#include <vector>
#include <thread>
using namespace std;

int square_sum = 0;

void pow2(int x)
{
    square_sum += x * x;
}

int main()
{
    vector<thread> threads;
    for (int i = 1; i <= 20; i++)
    {
        threads.push_back(thread(&pow2, i));
```

```
    }

    vector<thread>::iterator it;
    for (it = threads.begin(); it != threads.end(); it++)
    {
       (*it).join();
    }
    cout << "Sum auf squares up to 20 is = " << square_sum << endl;
    return 0;
}
```

Compilieren Sie das Programm mit `g++`. Sie wissen, dass die korrekte Lösung für das vorgegebene Problem 2870 lautet. Wenn Sie das Programm laufen lassen, fällt Ihnen auf, dass meistens die korrekte Lösung gefunden wird, in einigen wenigen Fällen allerdings nicht. Dies ist auf eine so genannte "Race Condition" zurückzuführen. Wodurch wird diese hier verursacht? Zudem fällt Ihnen auf, dass das Benutzen von globalen Variablen schlechter Programmierstil ist. Schreiben Sie das Programm so um, dass pow2 eine Referenz auf die Variable, in der das Ergebnis gespeichert wird, erhält: `void pow2(int& square_sum, int x)`. Die Variable `square_sum` muss hier zwingend eine Referenz sein, da wir das neue Ergebnis in ihr speichern wollen. Das Instanzieren eines Threads funktioniert so allerdings nicht wie gewohnt: `thread(&square, square_sum, i)` wird einen Compilerfehler erzeugen. Kapseln Sie `square_sum` in einem `std::ref`-Objekt. Warum ist das nötig? Zur Lösung des ursprünglichen Problems werden wir drei verschiedene Lösungen erarbeiten. Schreiben Sie diese jeweils in eine eigene Datei mit dem Namen `threadn.cpp` (n=1, 2, 3).

Ansatz 1 - Mutex

Locken sie die Variable `square_sum` mit einem `std::mutex`-Objekt. Achten Sie darauf, keinen Deadlock zu produzieren! Wie wird die Race-Condition durch den `Mutex` aufgelöst?

Ansatz 2 - Atomic

C++ stellt Ihnen den `std::atomic`-Container zur Verfügung. Dieser kann ebenfalls zur Lösung des gegebenen Problems verwendet werden. Warum kann es bei der direkten Verwendung eines `Atomic`-Containers hier keinen Deadlock geben?

Ansatz 3 - Tasks

Ein weiterer Ansatz ist von Threads zu Tasks zu abstrahieren. Betrachten Sie dazu das folgende Beispiel:

```
#include <iostream>
#include <future>
#include <chrono>
using namespace std;

int square(int x) {
    return x * x;
}

int main() {
    auto a = async(&square, 10);
    int v = a.get();

    cout << "The thread returned " << v << endl;
    return 0;
}
```

Das `async`-Konstrukt nutzt dabei zwei Objekte: *Promise* and *Future*. Promise verspricht das Ergebnis einer Berechnung zu liefern. Das Future-Objekt wird mit dem Promise-Objekt verknüpft und kann jederzeit nach einem Ergebnis gefragt werden, indem `get()` aufgerufen wird. Wurde das Ergebnis noch nicht geliefert, wird so lange gewartet, bis es berechnet wurde. Das `async`-Objekt versteckt die Implementierung weitestgehend. Als Ergebnis wird ein `future<int>`-Objekt zurückgegeben. Da der Compiler dies weis, kann er den entsprechenden Typ mittels `auto` ableiten.

Nutzen Sie das `async`-Konstrukt, um das Problem der Aufsummierung von Quadratzahlen zu lösen. Instanzieren Sie 20 `async`-Objekte und sammeln Sie die erzeugten `future<int>`-Objekte in einem Vektor. Iterieren Sie anschließend über den Vektor, um die Summe zu bestimmen.

Aufgabe 2: Threads für Bullets (40 Punkte)

Ziel der Übung:

Ziel der Übung ist es, sich mit Threads in C++ auseinander zu setzen. In dieser Übung werden wir die Software um die Funktionalität erweitern, die Raumschiffmodelle schießen zu lassen. Jede abgefeuerte Kugel wird dazu in einen in einem eigenen Thread verwaltet. In dieser Übung werden wir zwei Klassen implementieren bzw. weiterentwickeln: `Bullet` und `SpaceCraft`. Die Grundsignaturen, die Sie bei Bedarf erweitern dürfen, finden Sie in Ihrem Repository im Unterordner `/rendering`.

Beschreibung der zu implementierenden Funktionalität:

Klasse Bullet:

Der Konstruktor der Klasse bekommt als Parameter die aktuelle Position und Ausrichtung des Modells, von dem die Kugel abgeschossen wurde. In der `run()`-Methode läuft eine Schleife `m_lifetime` Iterationen. In jedem Schleifendurchgang wird die Position der Kugel um eine Einheit entlang `m_flightAxis` bewegt. Nach Beendigung der Schleife wird `m_alive()` von `true` auf `false` gesetzt, um anzuzeigen, dass die Kugel ihre maximale Schussdistanz erreicht hat und nun nicht mehr aktiv ist. Nutzen Sie zur Abarbeitung der `run()`-Methode einen `std::thread`. Fügen Sie diesen der Klasse hinzu. Achten Sie darauf, dass der Thread erst startet, nachdem `start()` aufgerufen wurde. Ein Aufruf von `stop()` joint den Thread und setzt `m_alive` auf `false`.

Ob die Kugel ihre Endposition erreicht hat, soll von `isAlive()` berichtet werden. Dazu wird schlicht der Status von `m_alive` zurückgegeben. Stellen Sie sicher, dass die Schleife in `run()` nicht zu schnell läuft, indem Sie den Thread nach jeder Iteration für 1000 Mikrosekunden schlafen legen. Dazu können versenden sie die Funktion `std::this_thread::sleep_for` mit einer entsprechenden Dauer. Warum ist das notwendig und sinnvoll? Nutzen Sie zum Rendern des Bullets das interne Objekt `m_sphere`. Als initialer Radius bietet sich ein Wert von 10 Einheiten an.

Klasse SpaceCraft:

Aufruf der Methode `shoot()` erzeugt eine neue Bullet-Instanz und fügt sie in `m_bullets` ein. In jedem Aufruf von `render()` wird zunächst das Modell gezeichnet. Anschließend wird durch die Liste der Kugeln iteriert und alle aktiven Kugeln werden an ihrer aktuellen Position gerendert. Sobald eine Kugel nicht mehr aktiv ist (`isAlive()` gibt `false` zurück), soll sie aus der Liste der zu rendernden Kugeln entfernt werden. Nutzen Sie dazu einen geeigneten Iterator auf `m_bullets`. Eine vorhandene Kugel können Sie mittels `erase()` aus dem Vektor löschen. Fügen Sie der Methode `handleKeyInput` einen `shoot()`-Aufruf hinzu, der durch Drücken der Space-Taste aktiviert wird.

Abgabe

Checken Sie Ihre Lösung des Aufgabenblattes bis Montag den 16.01.2023, 8:00 Uhr in den Master-Branch des git-Repositorys Ihrer Gruppe ein.