Wie erzeugt man Threads

Um Threads zu erzeugen, braucht man in C++ eine Funktion, ein Lambda oder ein anderes Callable. Betrachten wir den Quellcode in Demo 1, in dem 2 solche Funktionen `print_plus(), print_minus()` programmiert sind, die man durch selbsterzeugte Threads ausführen lassen könnte. Dieses Programm macht davon noch keinen Gebrauch und startet die Funktionen sequentiell durch den Haupt-Thread.

```
void print_plus()
{   for (int i{}; i < 5000; ++i)    cout << '+'; }

void print_minus()
{   for (int i{}; i < 5000; ++i)    cout << '-';}

int main()
{
    print_plus();
    print_minus();
}
```

Nach dem Inkludieren von `<thread>` kann man neue Software-Threads erzeugen (die auf anderen Hardware-Threads der CPU ausgeführt werden **können**). Ein Konstruktoraufruf einer `std::thread` Instanz mit der Startfunktion reicht aus:

```
int main()
{
     std::thread{print_plus}; // Thread, der print_plus() abarbeitet
     std::thread{print_minus};// Thread, der print_minus() abarbeitet
}
```

Unter Linux **kommt es hier zu einem Absturz**, unter Windows jedoch nicht: Für jede `std::thread` Instanz muss laut Standard die `join()` – Methode aufgerufen werden (oder die `detach()`-Methode), sonst erfolgt ein Fehlerabbruch. Daher wird das Programm unter Linux korrekterweise terminiert, während es unter Windows normal endet?

Wir müssen in unser Programm also entweder die `join()` Methode einbauen oder statt dessen den Typ `std::jthread` verwenden, der im Destruktor selbst die `join()` Methode automagisch aufruft:

```
int main()
{
     std::jthread{print_plus};
     std::jthread{print_minus};
}
```

Damit funktioniert das Programm auch unter Linux und es ist multithreaded, aber leider arbeiten die Threads nicht parallel sondern sequentiell! Wir haben den Threads keine Namen gegeben, daher werden sie als temporäre Objekte erzeugt und gleich nach der Erzeugung wieder vernichtet. Im Destruktur wird die `join()`– Methode aufgerufen. Diese wartet, bis der Thread seine Arbeit beendet hat. Deshalb wird der zweite Thread erst erzeugt, nachdem der erste „fertig hat“. Einfache Lösung: Threads mit Namen!

```
int main()
{
     std::jthread t1{print_plus};
     std::jthread t2{print_minus};
}    // t1 und t2 rufen hier automagisch join() auf.
```

Und wenn die Thread-Funktion Parameter hat?

In der nächsten Variante habe ich die 2 Funktionen sinnvollerweise durch eine ersetzt:

```
void print_char(c)
{   for (int i{}; i < 5000; ++i)      cout << c; }
```

Die Threads müssen jetzt folgendermaßen gestartet werden:

```
     std::jthread t1{print_char, '+'};
     std::jthread t2{print_char, '-'};
```

d.h. es wird der Name der Funktion und danach alle deren Argumente geschrieben. Besitzt die Funktion **Argumente mit Defaultwert**, so müssen diese hier **explizit** angegeben werden!

```
void print_char(c, int n = 5000)   // n hat Defaultwert
{   for (int i{}; i < n; ++i)      cout << c; }

std::jthread t1{print_char, '+' };          // Fehler!
std::jthread t1{print_char, '+', 5000 };    // ok!
```

Wie kompiliert man ein Programm mit Threads?

Windows: Sie inkludieren `thread` (oder `future`), fertig
Linux, Mac?: Zusätzlich verwenden Sie die Option **`–pthread`**

Und wenn man viele Threads erzeugen muss/will?

Sie speichern einfach alle `thread` oder `jthread` Instanzen in einem Vektor:

```
std::vector<std::jthread> worker;
for (int i{}, i < 20; ++i
     worker.push_back(std::jthread{print_char, 'a'+i, 5000});
```

Bei dieser Variante **MUSS** man anonyme Thread-Objekte erzeugen: Da diese temporär sind, kopiert `push_back()` diese Instanzen nicht (Threads sind nicht kopierbar!), sondern „weidet sie aus“ und erzeugt die Kopie aus den Bestandteilen (`move`-Semantik).

Eine bessere Alternative ist die Verwendung dem `emplace_back()` Methode des Containers. Hierbei wird nicht ein existierendes Objekt in den Vektor verschoben, sondern es wird das Objekt im Container an der richtigen Stelle erzeugt und es entsteht dadurch auch keine Kopie. Allerdings muss man diese Methode auch richtig aufrufen! Zuerst falsch:

```
worker.emplace_back(std::jthread{print_char, 'a'+i, 5000});
```

Dieser Aufruf von `emplace_back()` macht das gleiche wie `push_back`(), weil Sie eine schon fertige Threadinstanz übergeben. Der einzig sinnvolle Aufruf ist:

```
worker.emplace_back(print_char, 'a'+i, 5000);
```

d.h. Sie übergeben alles, was `emplace_back()` braucht, um den Thread selbst zu erzeugen!