BETRIEBSSYSTEME
PRAKTISCHE UMSETZUNG IN C++

Prof. Dr. Jörg Mielebacher
mail@mielebacher.de

Die folgenden Folien zeigen praktischen Umsetzungen der vergangenen Betriebssystemvorlesungen in C++. Zu jeder Folie sind Notizenseiten erfasst.

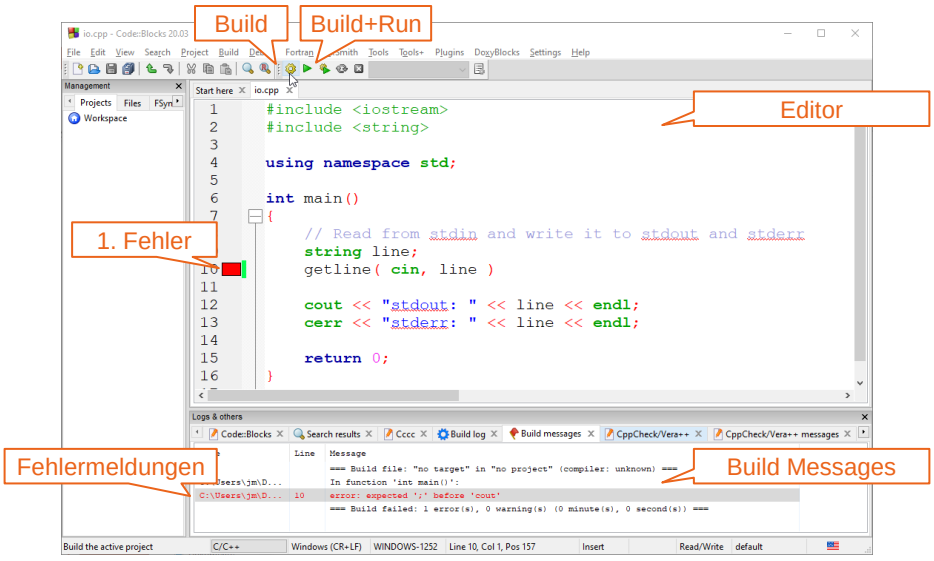
Verbesserungsvorschläge und Fehlerhinweise können Sie gerne an die Adresse mail@mielebacher.de senden.

Wiederholung

- Das Betriebssystem ermöglicht Anwendungen den Zugriff auf das Dateisystem.
- Das Dateiformat beschreibt, wie Dateien aufgebaut und codiert sind; dies ist keine Aufgabe des Betriebssystems.
- Das Betriebssystem verwaltet für jeden Prozess die Datei-Handles im Prozesskontext.
- Konventionelle Festplatten adressieren durch CHS, SSD durch Pages und Blocks; verallgemeinert wird dies durch LBA.
- RAID fasst mehrere Festplatten zu einem logischen Laufwerk zusammen und bietet Redundanz.
- FAT32 bzw. exFAT, NTFS und ext4 sind verbreitete Dateisysteme. Es gibt noch viele weitere.
- Partitionierung, Formatierung, Einhängen, Prüfung und Überwachung sind typische Verwaltungsaufgaben.
- NFS, SMB und WebDAV erlauben den Zugriffe auf Dateien über Netzwerke.

Vorarbeiten

Code::Blocks



Code::Blocks ist eine einfache IDE für C++. Sie steht kostenfrei zur Verfügung unter www.codeblocks.org (bzw. für Linux über die Paketquellen) und kann unter Windows und Linux problemlos eingesetzt werden. Beim Download für Windows muss man darauf achten, dass man die Version mit dem Compiler MinGW installiert.

Nach dem Start und dem Öffnen/Anlegen einer Quellcode-Datei sieht man den Editor, darunter die Logs – vor allem mit den Build Messages – und links daneben der Management-Bereich, der nur bei Projekten gebraucht wird.

Mit dem Build-Button kann man den Quellcode übersetzen oder mit Build+Run übersetzen und bei Erfolg ausführen. Findet der Compiler Fehler werden diese mit der Zeilennummer in den Build Messages aufgelistet. Durch Doppelklick auf einen Fehler gelangt man zur betreffenden Quellcode-Zeile. Nach dem Übersetzen wird der erste gefundene Fehler im Quellcode mit einem roten Balken bei der Zeilennummer angezeigt.

Kommandozeilenparameter

Kommandozeilenparameter

```
#include <iostream>

using namespace std;

int main( int argc, char *argv[] )
{
   cout << endl << "Call:" << endl;

   for( int parnr = 0; parnr < argc; parnr++ )
      cout << parnr << ": " << argv[ parnr ] << endl;

   return 0;
}
```

Anzahl der übergebenen Parameter

Feld mit übergebenen Parametern

```
C:\Users\jm\Desktop>cmdargs arg1 arg2 arg3

Call:
0: cmdargs
1: arg1
2: arg2
3: arg3
```

Das Hauptprogramm eines C++-Programms kann zusätzliche Parameter besitzen, um sogenannte Kommandozeilenparameter auszuwerten. Auf diesem Weg lassen sich einem Programm direkt beim Aufruf bereits Eingaben (meist Einstellungen oder zu öffnende Dateien) mitgeben. Zum Beispiel würde man mit nano test das Programm nano aufrufen und ihm als zusätzlichen Parameter test übergeben (hier: um diese Datei zu öffnen).

C und C++ verwenden hierzu argc und argv. argc enthält die Anzahl der übergebenen Parameter (inkl. des Programmaufrufs selbst). argv ist ein Feld von C-Strings (also char-Feldern), mit den übergebenen Parametern. argv[0] enthält den Programmaufruf selbst.

Wie die Kommandozeilenparameter in den Adressraum des Prozesses gelangen, unterscheidet sich von Betriebssystem zu Betriebssystem. Unter Windows sorgt das Laufzeitsystem dafür, dass die Parameter in den Stack des Prozesses geladen werden. Dieses sorgt anschließend dafür, dass main() aufgerufen wird.

Aufgabe

- Finden Sie Beispiele für den Einsatz von Kommandozeilenparametern.

Ein/Ausgabe-Umleitung

Aufgabe

- Erstellen Sie ein Programm numbers, das die Zahlen von 1 bis 100 auf dem Bildschirm ausgibt.
- Das Programm sum liest Zahlen von der Standardeingabe ein, solange Zahlen eingegeben werden, und summiert diese auf.
- Rufen Sie numbers und sum so auf, dass die Ausgabe von numbers von sum verarbeitet wird und in die Datei result.txt geschrieben wird.

```
#include <iostream>

using namespace std;

int main()
{
    double sum = 0.0;
    double val = 0.0;

    while( cin >> val )
        sum += val;

    cout << "Summe: " << sum << endl;

    return 0;
}
```

Dateien

Dateien in C++

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ofstream file( "helloworld.txt" );

    if( !file )
    {
        cout << "File not ready." << endl;
        return -1;
    }

    file << "Hello world" << endl;

    file.close();

    return 0;
}
```

Datei zum Schreiben öffnen

Datei nicht geöffnet

Ausgabe in Datei

Datei schließen

Wie bereits früher gezeigt, können Sie in C++ mit ofstream Dateien zum Schreiben öffnen. Ist sie noch nicht vorhanden, wird sie angelegt. Ist sie bereits vorhanden, wird ihr Inhalt gelöscht. Die Prüfung, ob das Öffnen erfolgreich war, ist wichtig, um Folgefehler zu vermeiden. Anschließend kann man mit dem Shift-Operator (<<) Inhalte in die Datei schreiben. Die close()-Methode schließt die Datei, automatisch wird dies aber ohnehin im Destruktor von fstream ausgeführt – hier ist es nur enthalten, um die Methode vorzustellen.

Aufgabe

```cpp
#include <fstream>

using namespace std;

int main()
{
    ofstream dst( "test.txt" );

    if( !dst )
        return -1;

    for( int i = 0; i < 1000000; i++ )
        dst << i << '\n';

    return 0;
}
```

- Führen Sie das Programm aus für 10.000, 100.000 und 1.000.000 Wiederholungen. Vergleichen Sie die Laufzeiten.
- Ersetzen Sie nun ‘\n‘ durch endl und wiederholen Sie die Messungen.
- Vergleichen und erklären Sie die Ergebnisse.

Threads

Threads in C++

```
#include <iostream>
#include <thread>

using namespace std;

void t1()
{
  cout << "t1 starting" << endl;
  for( int i = 0; i < 500000000; i++ );
  cout << "t1 done" << endl;
}

void t2()
{
  cout << "t2 starting" << endl;
  for( int i = 0; i < 1000000000; i++ );
  cout << "t2 done" << endl;
}

int main()
{
  cout << "Starting..." << endl;

  thread first (t1);
  thread second (t2);

  first.join();
  second.join();

  cout << "All done." << endl;

  return 0;
}
```

Thread-Header

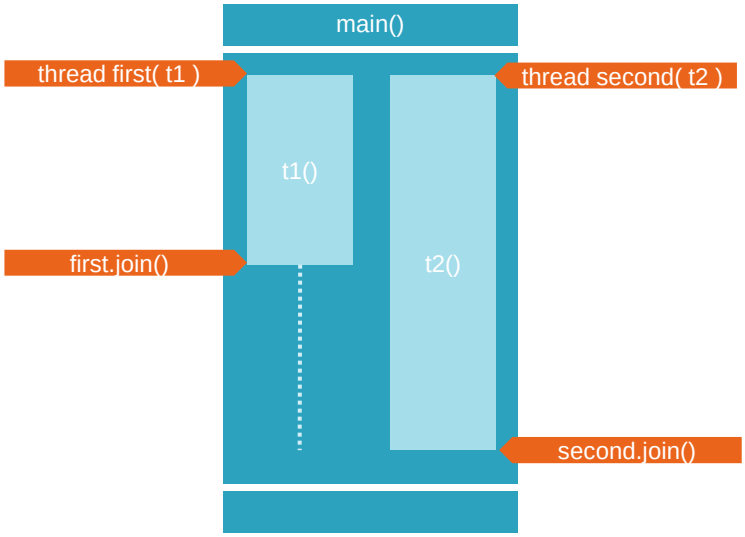
Lang laufende Funktion

Parallele Threads erzeugen

Auf Abschluss der Threads warten

Threads waren in C++ bis zum Standard C++11 nur mit zusätzlichen Bibliotheken möglich. Seit C++11 steht die plattformübergreifende thread-Klasse zur Verfügung. Der gleichnamige Header muss hierfür eingebunden werden. Ein einfacher Weg besteht darin, Threads auf der Grundlage von Funktionen zu erzeugen. Dem Konstruktor des Threads übergibt man hierfür die jeweilige Funktion. Der Thread wird dann gestartet, d.h. die zugeordnete Funktion (hier: t1 bzw. t2) wird ausgeführt. Durch Aufrufen der join()-Methode wartet man auf den Abschluss des jeweiligen Threads. Die Thread-Objekte werden automatisch durch ihren jeweiligen Destruktor zerstört.

Threads in C++



Den Ablauf kann man hier nochmal gut erkennen: Die Ausführung von main() startet, dann wird zunächst Thread first angelegt, der t1() ausführt, und anschließend Thread second, der t2 ausführt. first.join() wartet auf den Abschluss von first, während second.join() auf den Abschluss von second. Sind beide beendet, werden die verbleibenden Anweisungen von main() ausgeführt.

Threads in C++

```
#include <iostream>
#include <thread>

using namespace std;

void func()
{
  cout << "func thread" << endl;
}

class FuncObj
{
public:
    void operator()() const
    {
        cout << "FuncObj thread" << endl;
    }
};

int main()
{
  cout << "Starting..." << endl;

  thread t1( func );
  FuncObj fo;
  thread t2( fo );
  thread t3( []{ cout << "Lambda thread" << endl; } );

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

  cout << "All done." << endl;

  return 0;
}
```

Funktion als Thread

Funktionsobjekt als Thread

Lambda-Funktion als Thread

Wie gerade gesehen, können Funktionen als Grundlage für einen Thread dienen. Daneben sind aber auch Funktionsobjekte (d.h. mit dem ()-Operator überladen) und Lambda-Funktionen möglich.

Aufgabe

```
#include <iostream>
#include <thread>

using namespace std;

void add( int& sharedcnt, int& privatecnt )
{
    for( int i = 0; i < 100000000; i++ )
    {
        sharedcnt++;
        privatecnt++;
    }
}

int main()
{
    int shared = 0;  // Shared counter
    int cnt1   = 0;  // Counter of first thread
    int cnt2   = 0;  // Counter of second thread

    thread first( add, ref( shared ), ref( cnt1 ) );
    thread second( add, ref( shared ), ref( cnt2 ) );

    first.join();
    second.join();

    cout << "Shared: " << shared << endl
         << "cnt1:   " << cnt1 << endl
         << "cnt2:   " << cnt2 << endl;

    return 0;
}
```

Gemeinsamen und Thread-privaten Zähler erhöhen

Gemeinsamen und Thread-privaten Zähler als Referenz übergeben

Zählerstände ausgeben

Das hier gezeigte Programm soll in zwei Threads jeweils einen gemeinsamen Zähler und einen Thread-privaten Zähler erhöhen. Um die Zähler by-reference aus main() zu übergeben, werden C++-Referenzen verwendet. Bei der Übergabe an den Thread-Konstruktor muss aus syntaktischen Gründen die ref()-Funktion für diese Parameter verwendet werden.

Aufgabe

```
#include <iostream>
#include <thread>

using namespace std;

void add( int& sharedcnt, int& privatecnt )
{
    for( int i = 0; i < 100000000; i++ )
    {
        sharedcnt++;
        privatecnt++;
    }
}

int main()
{
    int shared = 0;  // Shared counter
    int cnt1   = 0;  // Counter of first thread
    int cnt2   = 0;  // Counter of second thread

    thread first( add, ref( shared ), ref( cnt1 ) );
    thread second( add, ref( shared ), ref( cnt2 ) );

    first.join();
    second.join();

    cout << "Shared: " << shared << endl
         << "cnt1:   " << cnt1 << endl
         << "cnt2:   " << cnt2 << endl;

    return 0;
}
```

- Welche Ausgabe erwarten Sie für dieses Programm?

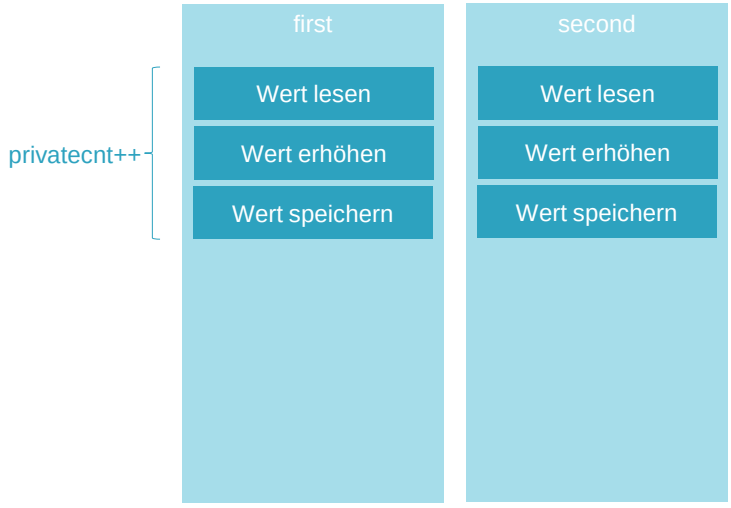
Aufgabe

```
#include <iostream>
#include <thread>

using namespace std;

void add( int& sharedcnt, int& privatecnt )
{
    for( int i = 0; i < 100000000; i++ )
    {
        sharedcnt++;
        privatecnt++;
    }
}

int main()
{
    int shared = 0;  // Shared counter
    int cnt1   = 0;  // Counter of first thread
    int cnt2   = 0;  // Counter of second thread

    thread first( add, ref( shared ), ref( cnt1 ) );
    thread second( add, ref( shared ), ref( cnt2 ) );

    first.join();
    second.join();

    cout << "Shared: " << shared << endl
         << "cnt1:   " << cnt1 << endl
         << "cnt2:   " << cnt2 << endl;

    return 0;
}
```

- Wie erklären Sie sich diese Ausgabe?

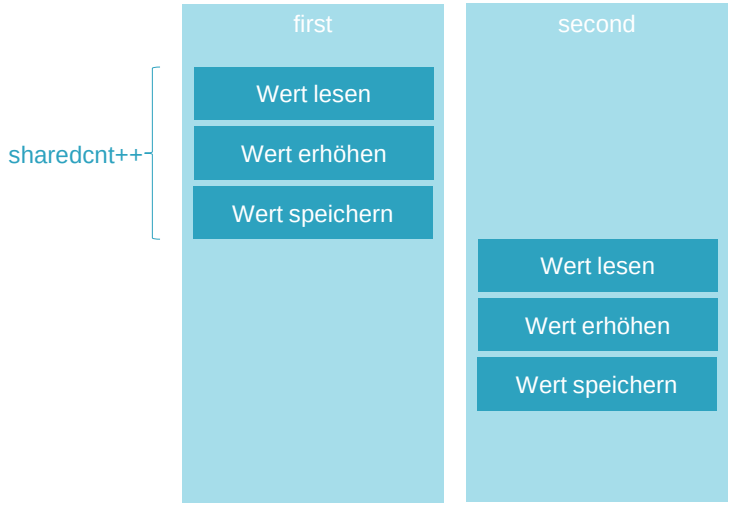
```
Shared: 125317071
cnt1:   100000000
cnt2:   100000000

Process returned 0 (0x0)   execution time : 1.243 s
Press any key to continue.
```

Private Zähler



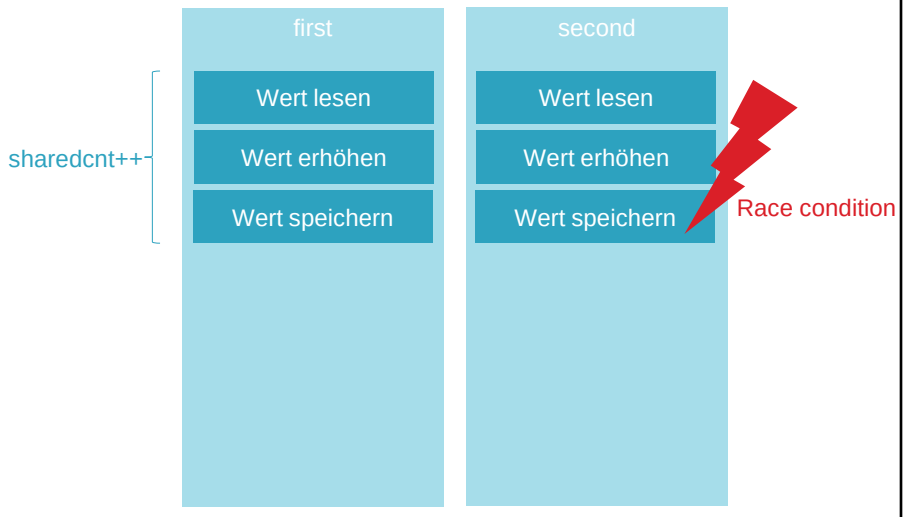
Bei den privaten Zählern ist die Lage einfach: Beide Threads operieren alleine auf dem jeweiligen Zähler. Die Threads beeinflussen sich nicht gegenseitig.

Gemeinsamer Zähler



Der gemeinsame Zähler wird von beiden Threads bearbeitet. Hier ist es unkritisch, wenn erst der eine Thread und dann der andere arbeitet.

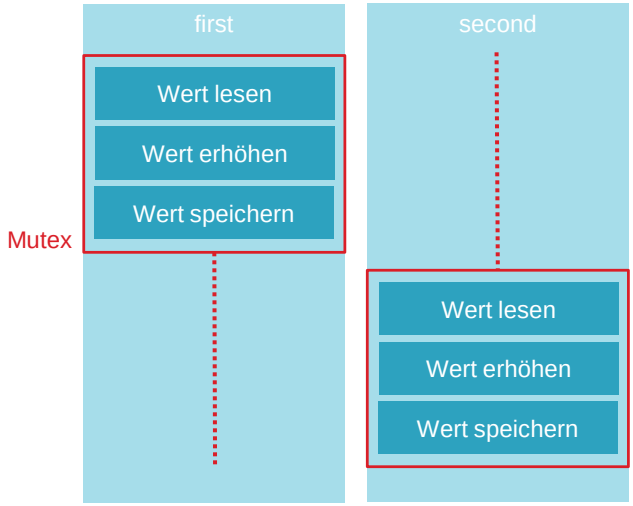
Gemeinsamer Zähler



Greifen aber beide Threads zur selben Zeit auf den Zähler zu, kann es zu Problemen kommen (was man auch an der geringen Höhe des gemeinsamen Zählers sehen kann). Der ++-Operator ist nämlich nicht atomar, vielmehr erfordert er mehrere Maschinenanweisungen: Der Wert des Zählers muss geladen werden, dann wird er erhöht und schließlich gespeichert.

Wenn also nun beide Threads zur selben Zeit den Wert lesen, ihn erhöhen und dann speichern, wird der Zähler u.U. nur einmal tatsächlich erhöht. Was hier passiert, bezeichnet man als Race Condition. Solche Race Conditions versucht man bei der parallelen Programmierung zu verhindern.

Gemeinsamer Zähler



Eine Gegenmaßnahme besteht darin, die Erhöhung des Zählers so zu schützen, dass der andere Thread während der ++-Operation warten muss und erst nach vollständigem Abschluss selbst den Zähler sperren und erhöhen darf. Man spricht hierbei von Mutex (Mutual Exclusion). Den zu schützenden Abschnitt des Codes bezeichnet man als kritischen Abschnitt.

Mutex in C++

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

void add( int& sharedcnt, int& privatecnt, mutex& addmutex )
{
    for( int i = 0; i < 100000000; i++ )
    {
        addmutex.lock();
        sharedcnt++;
        privatecnt++;
        addmutex.unlock();
    }
}
int main()
{
    int shared = 0;  // Shared counter
    int cnt1   = 0;  // Counter of first thread
    int cnt2   = 0;  // Counter of second thread

    mutex addmutex;
    thread first( add, ref( shared ), ref( cnt1 ), ref( addmutex ) );
    thread second( add, ref( shared ), ref( cnt2 ), ref( addmutex ) );

    first.join();
    second.join();

    cout << "Shared: " << shared << endl
         << "cnt1:   " << cnt1 << endl
         << "cnt2:   " << cnt2 << endl;

    return 0;
}
```

mutex-Header einbinden

mutex by-reference übergeben

Kritischen Bereich sperren

Kritischen Bereich entsperren

mutex-Objekt erzeugen

Die mutex-Klasse erlaubt in C++ die einfache Verwendung eines Mutex. Hierfür benötigt man den mutex-Header. Der Mutex wird hier im main() angelegt, um ihn nicht global anlegen zu müssen. Daher muss er aber auch by-reference an die Funktion add übergeben werden.

Der Mutex kann vor Beginn des kritischen Bereichs mit lock() gesperrt werden und danach mit unlock() wieder entsperrt werden. Dies kann zu Fehlern führen, wenn zwar gesperrt, aber nicht wieder entsperrt wird.

Mutex in C++

```
#include <iostream>
#include <thread>
#include <mutex>

using namespace std;

void add( int& sharedcnt, int& privatecnt, mutex& addmutex )
{
    for( int i = 0; i < 100000000; i++ )
    {
        lock_guard<mutex> guard( addmutex );
        sharedcnt++;
        privatecnt++;
    }
}

int main()
{
    int shared = 0;  // Shared counter
    int cnt1   = 0;  // Counter of first thread
    int cnt2   = 0;  // Counter of second thread

    mutex addmutex;
    thread first( add, ref( shared ), ref( cnt1 ), ref( addmutex ) );
    thread second( add, ref( shared ), ref( cnt2 ), ref( addmutex ) );

    first.join();
    second.join();

    cout << "Shared: " << shared << endl
         << "cnt1:   " << cnt1 << endl
         << "cnt2:   " << cnt2 << endl;

    return 0;
}
```

lock_guard ersetzt lock()/unlock()

```
Shared: 200000000
cnt1:   100000000
cnt2:   100000000

Process returned 0 (0x0)   execution time : 12.243 s
Press any key to continue.
```

Aus dem genannten Grund nutzt man für mutex-Objekt typischerweise das sog. lock_guard-Template. Mit Anlegen des lock_guards wird lock() ausgeführt, am Ende des Gültigkeitsbereichs (hier: nach privatecnt++) ruft der lock_guard-Destruktor wiederum unlock aus.

Aufgabe

```
void add( int& sharedcnt, int& privatecnt, mutex& addmutex )
{
    lock_guard<mutex> guard( addmutex );
    for( int i = 0; i < 100000000; i++ )
    {
        sharedcnt++;
        privatecnt++;
    }
}
```

- Warum nicht so?

Aufgabe

```
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

using namespace std;

void func( mutex& mutex1, mutex& mutex2 )
{
    mutex1.lock();
    this_thread::sleep_for( chrono::milliseconds( 10 ) );
    mutex2.lock();

    mutex1.unlock();
    mutex2.unlock();
}

int main()
{
    mutex mutexa;
    mutex mutexb;

    thread first( func, ref( mutexa ), ref( mutexb ) );
    thread second( func, ref( mutexb ), ref( mutexa ) );

    first.join();
    second.join();

    return 0;
}
```

- Was passiert hier?

Aufgabe

- Erstellen Sie eine thread-sichere Nachrichten-Warteschlage, die Methoden für das Einfügen (put) und das Entnehmen (take) besitzt.
- Tipp: Die STL-Queue ist eine gute Grundlage – ihr können Elemente mit push() hinzugefügt werden. Mit pop() kann man das nächste Element entfernen, das man aber zuvor mit front() abrufen muss.

Zusammenfassung

- Kommandozeilenparametern werden beim Aufruf eines Programms mit übergeben.
- Die Ein- und Ausgabe ist meist gepuffert.
- Threads werden in C++ durch Objekte der Klasse thread erzeugt.
- Race conditions müssen verhindert werden.
- Die kritischen Bereichen werden durch mutex-Objekte geschützt; so kann stets nur ein Thread darauf zugreifen, während die anderen warten.
- Deadlocks können auftreten, wenn Threads sich gegenseitig sperren.