

PROGRAMMIEREN II

DHBW Stuttgart Campus Horb INF2018

AGENDA

- Threads
- Synchronisierung
- Spiele

WAS MACHT DIESER CODE WOHL?

```
// Must allocate our own memory  
Test *ptr = (Test *)malloc(sizeof(Test));
```

```
// Use placement new  
new (ptr) Test;
```

```
// Must call the destructor ourselves  
ptr->~Test();
```

```
// Must release the memory ourselves  
free(ptr);
```


WAS BRAUCHE ICH UM
EINEN STATIC_CAST
AUSFÜHREN ZU KÖNNEN?

WER HAT ZUGEHÖRT?

```
#ifndef __C_API_TO_MY_PROGRAM
#define __C_API_TO_MY_PROGRAM
extern "C" {
    void do_something_smart();
    void do_something_not_so_smart(int a, int b, int c);
}
#endif
```

ERZEUGT DAS EINEN VEKTOR?

```
std::vector<float> myVector();
```

-

WAS MACHT DIESES TEMPLATE?

```
template <typename T>  
T myMax (T x, T y)  
{  
    return (x > y) ? x : y;  
}
```

WOMIT KANN MAN DIESES TEMPLATE VERWENDEN?

```
template <typename T>  
T myMax (T x, T y)  
{  
    return (x > y) ? x : y;  
}
```


WELCHER CAST WIRD WANN AUSGEFÜHRT?

- `static_cast`
- `reinterpret_cast`
- `const_cast`
- `dynamic_cast`

PROGRAMMIER AUFGABE

ZUM WARMWERDEN

- Lies die Datei bin_out in den Speicher und knacke den Verschlüsselungscode
- Die Datei ist wie folgt abgelegt:
 - int16_t für die Größe der folgenden Nachricht
 - Dann ein Block in der Größe mit der Nachricht
 - Die Nachricht ist in verkehrter Reihenfolge und jeder Char + 1 gerechnet

PARAMETER AUS DER KOMMANDOZEILE

```
int main(int argc, char **argv)
{
    int x = 100;
    std::string fileName = "bin_out";
    if (argc > 1)
    {
        x = atoi(argv[1]);
    }
    if(argc == 3){
        fileName = argv[2];
    }
    // ....
}
```


THREAD

- Konstrukt für Nebenläufigkeit
- Gerne auch als leichtgewichtiger Prozess bezeichnet
- Es gibt unterschiedliche Arten von Threads
 - Kernel Threads
 - User Threads auch Green threads genannt

KERNEL THREAD

- Ein Thread ist ein sequentieller Bearbeitungsablauf innerhalb eines Prozesses
- Er teilt sich mit anderen Threads die sogenannten Betriebsmittel:
 - Codesegment
 - Datensegment
 - Dateidescriptoren (häufig im Datensegment)

KERNEL THREAD

- Jeder Thread besitzt eine Reihe von eigenen Ressourcen, die im sogenannten Threadkontext zusammengefasst werden:
 - Unabhängiger Registersatz inkl. Instruction Pointer
 - Einen eigenen Stack (jedoch geteilten Heap!)
 - Eventuell: Thread Local Storage

KONFLIKTE

- Da der Heap geteilt ist und Zeiger eventuell in mehreren Threads aktiv sind kann es zu Ressourcenkonflikten kommen
- —> Es werden Synchronisationsmechanismen benötigt (siehe VL ParaProg)

C++ UND THREADS

- Bis C++11 keine Standardisierte Thread Library
- Es musste direkt auf der entsprechenden API gearbeitet werden (Windows Threads, POSIX)
- Boost::thread als Zwischenlösung
- Dieser wurde in C++11 standardisiert

THREAD EXAMPLE

```
// thread example
#include <iostream>           // std::cout
#include <thread>              // std::thread

void foo()
{
    // do stuff -> Heavy Computing...
}

void bar(int x)
{
    // do stuff -> Heavier Computing...
}

int main()
{
    std::thread first (foo);    // spawn new thread that calls foo()
    std::thread second (bar,0); // spawn new thread that calls bar(0)
    std::cout << "main, foo and bar now execute concurrently...\n";
    // synchronize threads:
    first.join();               // pauses until first finishes
    second.join();              // pauses until second finishes
    std::cout << "foo and bar completed.\n";
    return 0;
}
```


WAS SEHEN WIR IM ERSTEN BEISPIEL?

- `std::thread` als Implementierung von Threads
- Der Konstruktor nimmt eine Funktion / einen Funktionspointer und die zu übergebenden Parameter an
- Thread wird mit dem Konstruktor direkt gestartet
- Mit `Join` wird ein klassisches Zusammenführen der Threads ermöglicht

THREAD EXAMPLE II

```
#include <thread>
#include <chrono>
#include <iostream>
void do_something(int p)
{
    for (int i = 0; i < p; ++i)
    {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::cout << "Hello from Thread!" << std::endl;
    }
}

int main()
{
    std::cout << "Hallo Starte Thread" << std::endl;
    std::thread sread(do_something, 5);
    std::cout << "Thread gestartet caiptain!" << std::endl;
    sread.join();
    std::cout << "It's all done!" << std::endl;
}
```

WAS MACHE ICH WENN ICH EINEN THREAD ALS MEMBER NUTZEN MÖCHTE?

```
std::thread second (bar,0); // spawn new thread that calls bar(0)
```

?



DIE LÖSUNG!

- Pointer!
- Die noch bessere Lösung?
 - Smartpointer

WAS MACHT DIESER CODE?

```
#include <thread>
#include <iostream>
#include <vector>
int main()
{
    int max = 100000;
    int counter = 0;

    auto fun = [&counter, max]() {while(++counter<max); };
    std::vector<std::thread*> threads;
    for (int i = 0; i < 10; ++i)
    {
        std::thread *t = new std::thread(fun);
        threads.push_back(t);
    }
    for (auto &t : threads)
    {
        t->join();
        delete t;
    }

    std::cout << counter << std::endl;
}
```

SEHR HÜBSCH

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
localhost:day7 mhptorsten$ ./count
100006
localhost:day7 mhptorsten$ ./count
100002
localhost:day7 mhptorsten$ ./count
100004
localhost:day7 mhptorsten$ ./count
100003
localhost:day7 mhptorsten$ ./count
100004
localhost:day7 mhptorsten$ ./count
100003
localhost:day7 mhptorsten$ □
```


MUTEX

```
#include <iostream>
#include <chrono>
#include <thread>
#include <mutex>
#include <map>
#include <string>
std::map<std::string, std::string> g_pages;
std::mutex g_pages_mutex;
void save_page(const std::string &url)
{
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::string result = "fake content";

    g_pages_mutex.lock();
    g_pages[url] = result;
    g_pages_mutex.unlock();
}
int main()
{
    std::thread t1(save_page, "http://foo");
    std::thread t2(save_page, "http://bar");
    t1.join();
    t2.join();
    g_pages_mutex.lock();
    for (const auto &pair : g_pages) {
        std::cout << pair.first << " => " << pair.second << '\n';
    }
    g_pages_mutex.unlock();
}
```

PROBLEME MIT MUTEX?

```
#include <iostream>
#include <chrono>
#include <thread>
#include <mutex>
#include <map>
#include <string>
std::map<std::string, std::string> g_pages;
std::mutex g_pages_mutex;
void save_page(const std::string &url)
{
    std::this_thread::sleep_for(std::chrono::seconds(2));
    std::string result = "fake content";

    g_pages_mutex.lock();
    g_pages[url] = result;
    g_pages_mutex.unlock();
}
int main()
{
    std::thread t1(save_page, "http://foo");
    std::thread t2(save_page, "http://bar");
    t1.join();
    t2.join();
    g_pages_mutex.lock();
    for (const auto &pair : g_pages) {
        std::cout << pair.first << " => " << pair.second << '\n';
    }
    g_pages_mutex.unlock();
}
```

PROBLEM LÖSUNG FÜR EXCEPTIONS:

```
#include <thread>
#include <mutex>
#include <iostream>
int g_i = 0;
std::mutex g_i_mutex; // protects g_i
void safe_increment()
{
    std::lock_guard<std::mutex> lock(g_i_mutex);
    ++g_i;
    std::cout << std::this_thread::get_id() << ": " << g_i << '\n';
    // g_i_mutex is automatically released when lock
    // goes out of scope
}
int main()
{
    std::cout << "main: " << g_i << '\n';
    std::thread t1(safe_increment);
    std::thread t2(safe_increment);
    t1.join();
    t2.join();
    std::cout << "main: " << g_i << '\n';
}
```


LESEN != SCHREIBEN

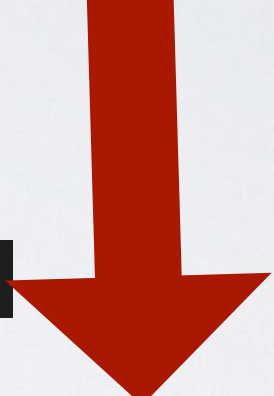
```
#include <iostream>
#include <mutex> // For std::unique_lock
#include <shared_mutex>
#include <thread>
class ThreadSafeCounter
{
public:
    ThreadSafeCounter() = default;
    // Multiple threads/readers can read the counter's value at the same time.
    unsigned int get() const
    {
        std::shared_lock<std::shared_mutex> lock(mutex_);
        return value_;
    }
    // Only one thread/writer can increment/write the counter's value.
    void increment()
    {
        std::unique_lock<std::shared_mutex> lock(mutex_);
        value_++;
    }
    // Only one thread/writer can reset/write the counter's value.
    void reset()
    {
        std::unique_lock<std::shared_mutex> lock(mutex_);
        value_ = 0;
    }
private:
    mutable std::shared_mutex mutex_;
    unsigned int value_ = 0;
};
```

FUTURES

```
int sum_it_up(int count, int steps)
{
    auto c = 0;
    for (int i = 0; i < count; i += steps)
    {
        if (rand() % 2)
        {
            std::this_thread::sleep_for(std::chrono::milliseconds(2));
        }
        ++c;
    }
    return c;
}
```

FUTURE DER WICHTIGE TEIL

Magie



```
int main()
{
    srand(time(0));
    std::deque<std::future<int>> deque;
    for (int i = 0; i < 100; ++i)
    {
        deque.push_back(std::async(std::launch::async, []
() { return sum_it_up(10000, 1); }));
    }
    for (std::future<int> &f : deque)
    {
        auto x = f.get();
        std::cout << "Nach hochzählen: " << x << std::endl;
    }
}
```


ATOMICS

- Es gibt auch atomics, genau wie in Java....
- `std::atomic_int...`

PROBLEMLÖSUNG FALLS ES FEHLER GIBT

```
g++ -std=c++11 -o main -pthread main.cpp
```

```
-Wl,--no-as-needed alone does not work.
```

```
-lpthread alone does not work.
```

```
-Wl,--no-as-needed and -lpthread together work:
```

```
g++ -std=c++11 -o main -Wl,--no-as-needed main.cpp -lpthread
```

PROGRAMMIER AUFGABE II

- Nehme die Aufgabe von vorhin und erzeuge mit `gen_bin.cpp` mehrere solcher Binär-Dateien (oder nimm die 4 fertigen)
- Schreibe ein Programm, dass eine Liste von Dateien als Start-Parameter bekommt
- Jede dieser Dateien soll in einem eigenen Thread entschlüsseln
- Am Ende soll auf der Kommandozeile ausgegeben werden, welche Datei mit welcher Größe und welcher Anzahl Nachrichten entschlüsselt wurden sowie wie viele Sekunden dies benötigt hat