Teil 3: Threads

Herb Sutter ("The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software," Herb Sutter, Dr. Dobb's Journal, 30(3), March 2005.):

Applications will increasingly need to be concurrent if they want to fully exploit continuing exponential CPU throughput gains.

Efficiency and performance optimization will get more, not less, important.

threads

BUT:

The 1998 C++ Standard doesn't acknowledge the existence of threads, and the operational effects of the various language elements are written in terms of a sequential abstract machine. Not only that, but the memory model isn't formally defined, so you can't write multithreaded applications without compiler-specific extensions to the 1998 C++ Standard. [Anthony Williams: C++ Concurrency in Action - Practical Multithreading, February, 2012,ISBN: 9781933988771]

C++11 behebt all diese Defizite:

- `std::thread` (<thread>)
- Synchronisation und Ausschluss
- verschiedene Konsistenzmodelle des Speicherzugriffs
- Atomare Typen

threads

- nebenläufige (und damit potentiell parallele) Aktionen zum initialen `main`-Thread sind immer an ein `std::thread`-Objekt bei dessen Konstruktion zu knüpfen
- diese Aktion kann einen beliebige *callable entity* sein:
  - Funktion, Funktionszeiger
  - funktionale Objekte (`operator ()`)
  - `std::function`-Objekte (`<functional>`)
  - Memberfunktionen und Zeiger auf Memeberfunktionen
  - Lambdas (!)
- der Thread startet SOFORT (kein `run()` o. ä.) als (potentiell) parallele Aktion
- wenn das `std::thread`-Objekt verschwindet (nicht vor dem Ende des Threads selbst), muss entschieden sein, ob der startende Thread (initial `main`) auf die Beendigung des neuen Threads warten soll (`thread::join()`) oder von diesem entkoppelt weiterläuft (`thread::detach()`), andernfalls wird `std::terminate()` gerufen !

threads

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

void hello() {
    std::cout<<"Hello Concurrent World\n";
}

int main()
{
    std::thread t(hello); // very bad effort/benefit ratio

    t.join();
}
```

erreicht der `main`-Thread sein Ende, ist das Programm beendet (ggf. Abbruch noch laufender abgekoppelter Threads)

DEMO: simple_threads

threads

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

class background_task {
public:
    void operator()() const {
        do_something();
        do_something_else();
    }
};

int main(){
    background_task f;
    std::thread t(f); // f will be copied into the storage of the new thread
    t.join();
}

// beware: std::thread(backgound_task()); ???
           std::thread((backgound_task())); OR
           std::thread{backgound_task()};
```

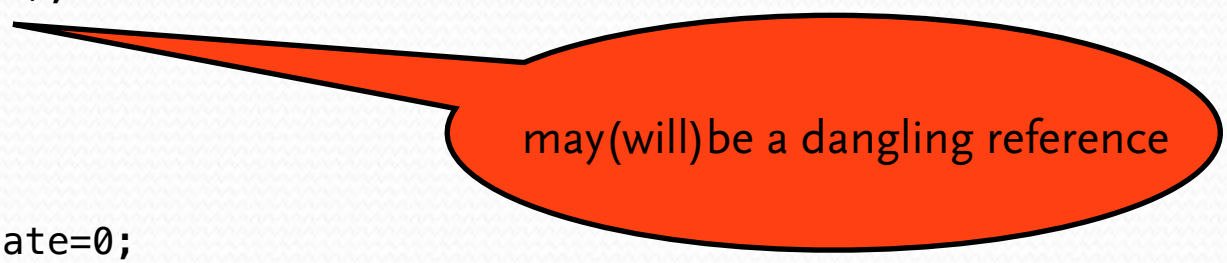
threads

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

int main(){
    std::thread t([]{   // Lambda: „inline action“
                        do_something();
                        do_something_else();
                    });
    t.join();
}
```

threads

bei abgekoppelten Threads (`detach()`) können u.U. leicht (fehlerhaft) undefinierte Oprationen auftreten:

```
struct func {
  int& i;
  func(int& i_):i(i_){}
  void operator()() {
    for(unsigned j=0;j<1000000;++j) {
      do_something(i);
    }
  }
};

void oops() {
  int some_local_state=0;
  func my_func(some_local_state);
  std::thread my_thread(my_func);
  my_thread.detach();
}
```



threads

kann man Argumente an die Thread-Aktion übergeben?

JA: *variadic templates* machen das möglich:

```
void f(int i,std::string const& s);

std::thread t(f,3,”hello”); // std::string(”hello”)
```

Vorsicht bei impliziten Umwandlungen der Argumente: die Quelle muss noch garantiert existieren, wenn die Umwandlung (irgendwann vor dem Start des Threads) stattfindet

```
void f(int i,std::string const& s);
void oops(int some_param) {
  char buffer[1024];
  sprintf(buffer, "%i",some_param);
  std::thread t(f,3,buffer); // maybe undefined !
  t.detach();
}
```

threads

kann man Argumente an die Thread-Aktion übergeben?

JA: *variadic templates* machen das möglich:

```
void f(int i,std::string const& s);

std::thread t(f,3,”hello”); // std::string(”hello”)
```

Vorsicht bei impliziten Umwandlungen der Argumente: die Quelle muss noch garantiert existieren, wenn die Umwandlung (irgendwann vor dem Start des Threads) stattfindet

```
void f(int i,std::string const& s);
void ok(int some_param) {
  char buffer[1024];
  sprintf(buffer, "%i",some_param);
  std::thread t(f,3,std::string(buffer)); // always defined !
  t.detach();
}
```

threads

Argumente werden IMMER kopiert, selbst wenn die Thread-Funktion eigentlich eine Referenz erwartet:

Was, wenn der Thread am Original eine Änderung bewirken soll?

```
void update_data_for_widget(widget_id w, widget_data& data);

void oops_again(widget_id w) {
  widget_data data;
  std::thread t(update_data_for_widget, w, data);
  display_status();
  t.join();
  process_widget_data(data); // data unchanged !
}
```

threads

Was, wenn der Thread am Original eine Änderung bewirken soll? Die Referenz explizit bereitstellen:

```
void update_data_for_widget(widget_id w, widget_data& data);

void ok_again(widget_id w) {
  widget_data data;
  std::thread t(update_data_for_widget, w, std::ref(data));
  display_status();
  t.join();
  process_widget_data(data); // data changed !
}
```

threads

unique/shared_ptr können an threads übergeben werden:

```
void process_big_object(std::unique_ptr<big_object>);

std::thread t1(make_unique<big_object>(args)); // no leak


std::unique_ptr<big_object> b(new big_object(args));
b->prepare_data(42); // ... has a name – is an lvalue

std::thread t2(process_big_object,std::move(b));
```

threads

Memberfunktionen brauchen zum Aufruf (Memberfunktions-)Zeiger UND Objekt:

```
class X {
public:
    void do_lengthy_work(int i);
};

X x;

// call x.do_lengthy_work(); in a thread:

std::thread t( &X::do_lengthy_work, &x, 42 );

//          ... the function
//          ......................... this
//          .............................. more args
```

threads

Threads sind identifizierbar:

```
// #include <thread>

// ask a thread for its ID:
std::thread::id std::thread::get_id();

// ask this thread (the current) for its ID:
std::thread::id std::this_thread::get_id();
```

`std::thread::id` ist vergleichbar und geordnet

`==` gleicher Thread oder beide ohne Thread

`<` ohne explizite Semantik

`std::hash<std::thread::id>` ist vordefiniert

`operator<<` definiert aber impl. defined

threads

Welche Parallelität kann maximal erreicht werden ?

```
// #include <thread>

[static] unsigned thread::hardware_concurrency();
```

1. *Returns:* The number of hardware thread contexts. [*Note:* This value should only be considered to be a hint. —*end note*] If this value is not computable or well defined an implementation should return 0.
2. *Throws:* Nothing.

Offenbar eine knappe Ressource:

- feste Zuordnung, überhaupt Zuordnung durch Nutzer besser vermeiden
- Parallelitätsbegriff ohne explizite Benutzung von Threads: `std::async`
- Thread-Pools (keine direkte Unterstützung durch C++11)