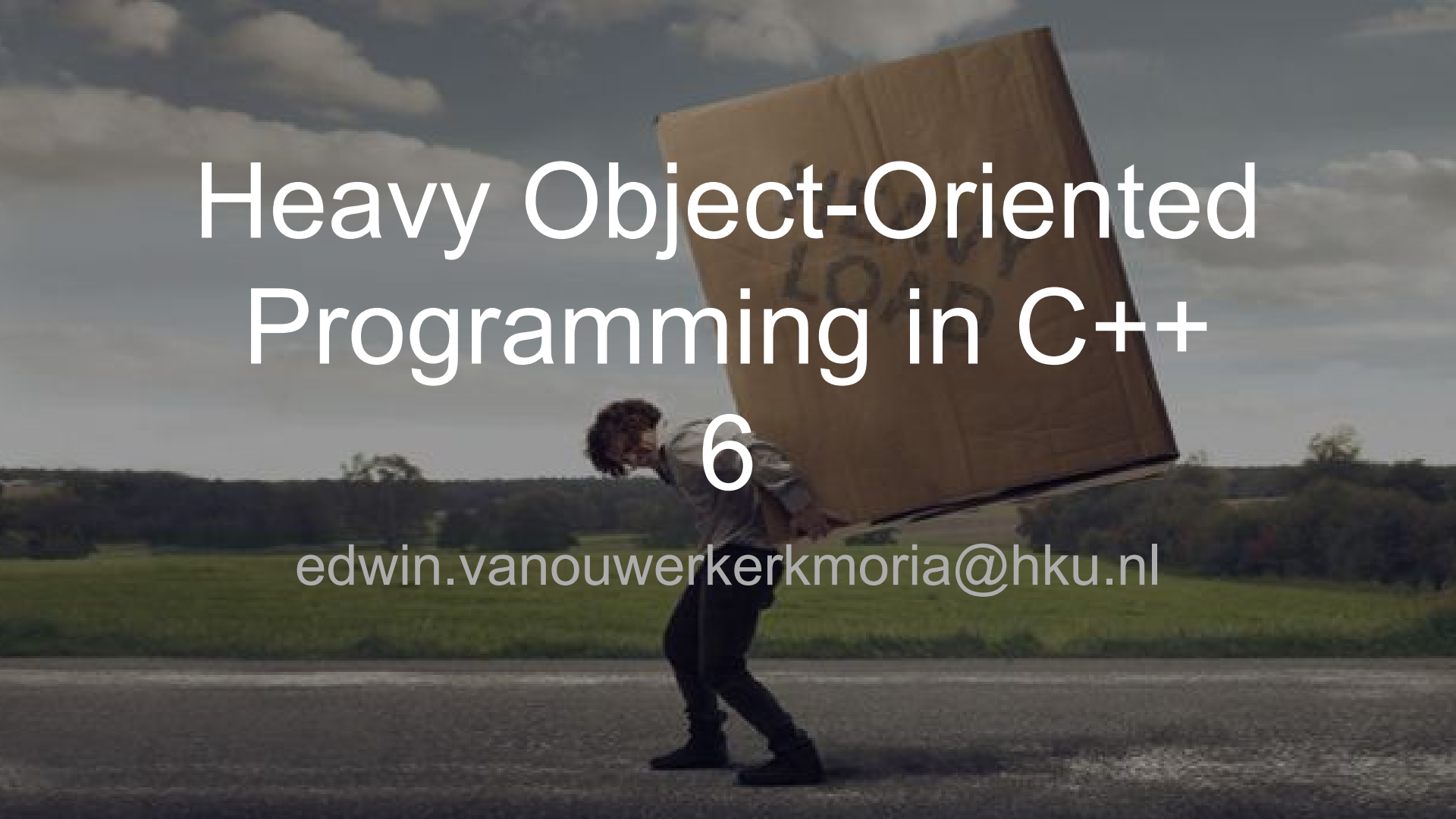


Heavy Object-Oriented Programming in C++

6

edwin.vanouwerkerkmoria@hku.nl

Vorige keer

Operator overloading

- als member-function
- als non-member function

Stream operators

Const-correctness

Opdracht vorige keer

Modelleer een bankrekening, met een transactie-log

- class `Bankrekening` (saldo, transactie-historie)
- class `Transactie` (bij/afschrijving, hoeveelheid, datum)

Definieër + / - operators om bij/afschrijvingen te verwerken

Definieër outputstream *insertion* operators:

```
cout << bankrekening << endl;
```

geeft saldo, en historie van transacties...

Uit opdracht: iterators & const

Bij overloaden stream operator is de RHS parameter ('t object wat je wilt streamen) const gedeclareerd. In de implementatie van de operator:

```
vector<Transaction>::iterator i = ...vector.begin();
```

krijg je een compiler error:

```
Error C2679 binary '=': no operator found which takes a
right-hand operand of type
'std::_Vector_const_iterator<std::_Vector_val<std::_Simple_t
ypes<_Ty>>>' (or there is no acceptable conversion)
```

declareer een `const_iterator`*, of gewoon* `auto`

Vandaag: threading in C++

Nieuw feature van de taal: C++ 11

```
#include <thread>
```

- threading bestond al wel, maar OS afhankelijk
  - *Windows:*
  - *Unix-achtigen:* ***pthread*** *(+Android)*
  - *iOS/OS X:* ***NSThread***
- gebaseerd op **Boost.Thread**
  - Nuttige library: Boost: (boost.org

Thread maken: wrap een functie

```
void myFunction() {
    // dit wordt in de thread uitgevoerd
}

std::thread thread(myFunction);

thread.join();
```

Altijd verplicht een .join() of .detach() - anders crash bij dtor van thread()

Thread maken: wrap een member function

```
class Task {
public:
    void execute(std::string command);
};


    Task * taskPtr = new Task();
    std::thread th(&Task::execute, taskPtr, "Sample Task");
    th.join();
```

Nuttige functies in `this_thread`

```
std::this_thread::sleep_for(std::chrono::milliseconds(10));
std::this_thread::sleep_until(std::chrono::milliseconds(10))
```

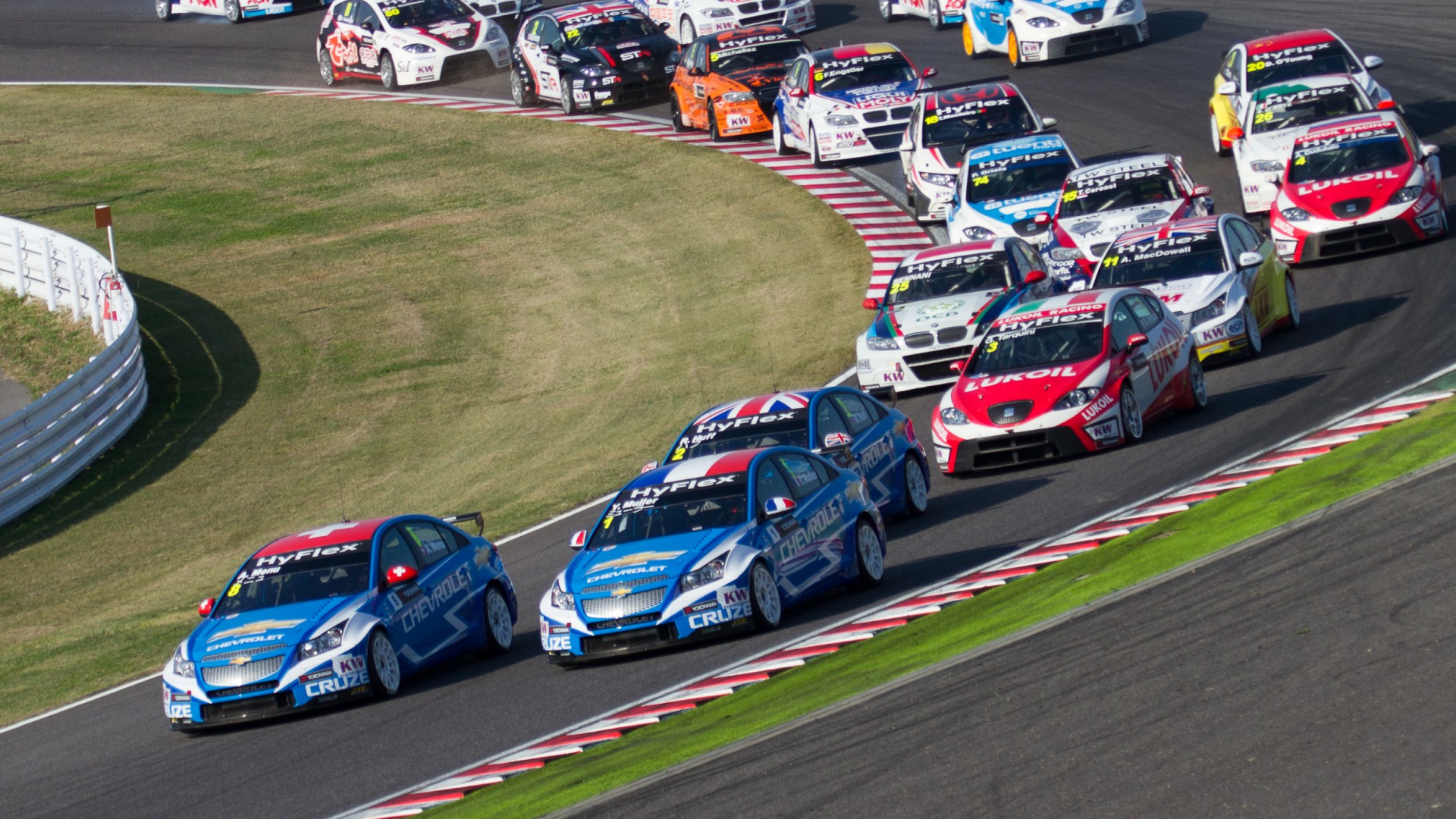
zet huidige thread stil

```
std::this_thread::get_id();
```

unieke ID van een thread

```
std::this_thread::yield();
```

geeft huidige timeslice op (OS kan thread reschedulen)


HyFlex
LUKOIL
CHEVROLET
CRUZE
KW

Race-conditions

Race-conditions ontstaan als 2 threads tegelijk in dezelfde data (of data-structuren zitten te schrijven.

Thread 1

```
int saldo = getBankSaldo();

saldo = saldo+150;

updateSaldo(saldo);
```

Thread 2

```
int saldo = getBankSaldo();

saldo = saldo - 50;

updateSaldo(saldo);
```

Synchronisatie van threads: Locks

Met een *lock* kun je code (tijdelijk) tegenhouden. De simpelste variant is een *mutex* (***mu***tually ***ex***clusive)

```
std::mutex mutex;
mutex.lock();
mutex.unlock();
```

Wrapper voor mutex: `std::lock_guard` - helpt bij 't unlocken

Er zijn nog veel meer varianten locks: zie

`http://en.cppreference.com/w/cpp/thread`

Resultaten uit een thread: future & promise

Een *future* en een *promise* geven de mogelijkheid om data terug te krijgen uit een thread.

`std::future` en `std::promise` worden altijd in combinatie gebruikt.

In de *promise* zet the thread 't resultaat. Uit de *future* haalt het aanroepende code 't resultaat op met `std::Promise<>::get()`. Deze methode blockt totdat 't resultaat beschikbaar is.

Synchronisatie: condition_variable

Als 2 threads van elkaar afhankelijk zijn kun je in de ene thread een vlaggetje zetten zolang de andere nog niet klaar is. Nadeel daarvan is dat de wachtende thread (of threads) staat te 'spinnen'.

Een `std::condition_variable` is een implementatie van 't **Observer** pattern: de wachtende thread slaapt, en luistert totdat 'm verteld wordt dat 'ie verder kan.

Je kunt met deze constructie meerdere luisterende threads hebben...

Opdracht volgende keer

1) Een C++ `std::vector` is standaard *niet* thread-safe. Maak een nieuwe class, `concurrent_vector` die gebruik maakt van locking om wél thread-safe te zijn. (Als interne datastructuur kun je wel `std::vector` gebruiken)

   Bewijs dat deze nu wel thread-safe is: voeg vanuit twee verschillende threads elementen toe aan de vector, en check de resultaten.

2) Run de producer-consumer code uit de klas-repository: Het resultaat zou 0 moeten zijn, maar is 't niet door een race-condition. Maak de code thread-safe met behulp van een condition-variable.