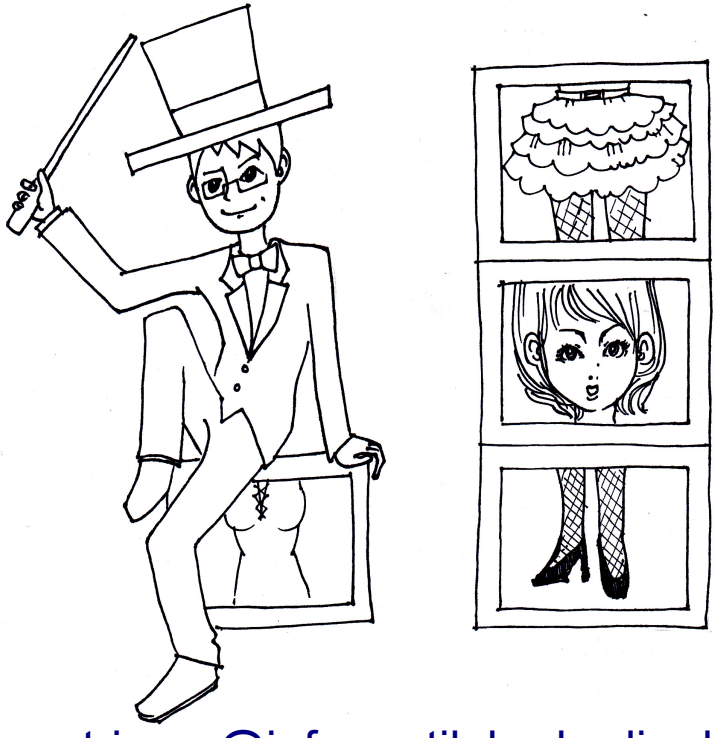
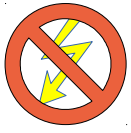
async Magic



(en)light**e**ning talk



trieger@informatik.hu-berlin.de

async as a code reordering specifier

Everything else is just multithreading

std::async

- **Standard C++ function template**
- **Supposed to make threads (easier)**
- **Already have `std::thread`, `pthread`, `boost::thread`, …**
- Takes functions, function objects and lambdas with an arbitrary number of parameters
  - No `void *` shenanigans
- Returns a future with a problem
- **Optional first parameter**
  - **`std::launch::async` → Creates a thread**
  - **`std::launch::deferred` → Does not create a thread**
  - **`std::launch::async | std::launch::deferred` → Let the compiler decide**

Reordering

```
void foo(){
    somelib::doStuffs();
    cout << “Hi there!\n”;
}
```

```
void foo(){
    auto f = async(
        somelib::doStuffs();
    );
    cout << “Hi there!\n”;
    f.wait();
}
```

```
void foo(){ //pseudo code
    if (cout_mutex.try_lock()){
        cout_withou_lock << “Hi there!\n”;
        cout_mutex.unlock();
        somelib::doStuffs();
    }
    else{
        somelib::doStuffs();
        cout << “Hi there!\n”;
    }
}
```

Also works for volatile memory accesses, mutexes and atomics

Magic

async can make single threaded code faster!

```
int var;
int spill(){
    var = 5;
    somelib::doSomething();
    return var + 7;
}
```

Thread semantic without threads

Data race free →
somelib::doSomething must not access var →
reordering possible →
performance → 

```
int var;
int spill(){
    var = 5;
    auto f = async([&]{var = var;});
    somelib::doSomething();
    f.wait();
    return var + 7;
}
```

Intend and result

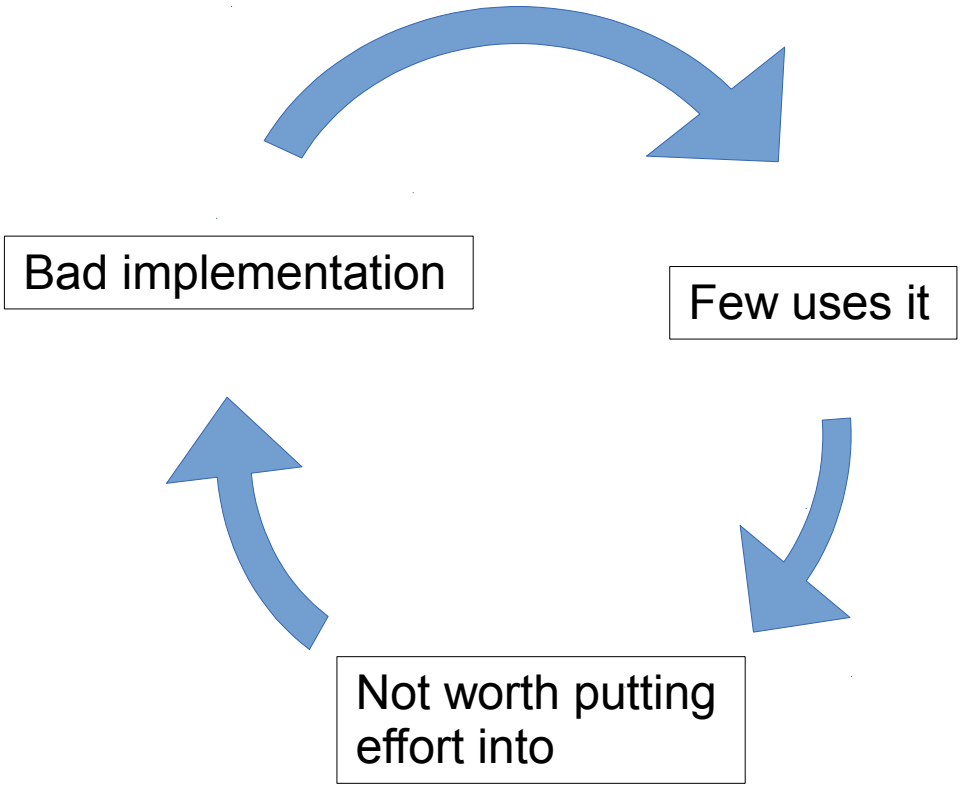
What the C++ standard committee specified:

- Templates for generic programming
- const correct STL
- **async for threads**

What the C++ standard committee realized later on:

- Templates are Turing-complete → Language in a language
- Redefinition of const to mean thread safe
- **Specification of a code reordering specifier**

Implementation



Implementation