Programmieren in C++

Weiterführende Themen

Inhalt

- Reguläre Ausdrücke
- Rechnen mit Einheiten
- Wahrscheinlichkeitsverteilungen
- Ranges und Views
- Coroutinen
- Parallel arbeiten
  - Multithreaded Maschinenmodell
  - Thread
  - Future und Promise (Kommunikationskanal zwischen Threads)
  - Mutex, Lock und mehr
  - Memory Order und Fence

Reguläre Ausdrücke

- Zeichenketten nach Mustern durchsuchen und Teile daraus extrahieren
- Beispiel

```
#include <regex>
static const regex natelNr(R"(07[689]\d{7,7})");

bool isNatel(const string& s) { return regex_match(s, natelNr); }
bool containsNatel(const string& s) { return regex_search(s, natelNr); }
```

- Syntax
  - Default: ECMAScript Syntax (wird auch in JavaScript verwendet)
  - weitere Syntaxen sind implementationsabhängig

Rechnen mit Einheiten

- Klasse ratio stellt auf Ganzzahlen normalisierte Brüche dar
  - darauf aufbauend lassen sich typsichere Umrechnungen implementieren
  - vordefinierte SI-Präfixe

```
template <intmax_t _Nx, intmax_t _Dx = 1>
struct ratio { // holds the ratio of _Nx to _Dx
    static constexpr intmax_t num = _Sign_of(_Nx)*_Sign_of(_Dx)*_Abs(_Nx)/_Gcd(_Nx, _Dx);
    static constexpr intmax_t den = _Abs(_Dx)/_Gcd(_Nx, _Dx);
    using type = ratio<num, den>;
};

template <class _Rx1, class _Rx2>
struct _Ratio_add { // add two ratios
    static constexpr intmax_t _Nx1 = _Rx1::num;
    static constexpr intmax_t _Dx1 = _Rx1::den;
    static constexpr intmax_t _Nx2 = _Rx2::num;
    static constexpr intmax_t _Dx2 = _Rx2::den;
    static constexpr intmax_t _Gx = _Gcd(_Dx1, _Dx2);

    // typename ratio<>::type is necessary here
    using type = typename ratio<_Safe_add(_Safe_mult<_Nx1, _Dx2 / _Gx>::value,
    _Safe_mult<_Nx2, _Dx1 / _Gx>::value), Safe_mult<_Dx1, _Dx2 / _Gx>::type;
};

template <class _Rx1, class _Rx2>
using ratio_add = typename _Ratio_add<_Rx1, _Rx2>::type;
```

Rechnen mit Längenangaben

```
template<typename R>
concept IsNatural = requires {
   requires R::den == 1;
};

template<typename S>
struct Length {
   int64_t m_val;

   constexpr Length(int64_t val) : m_val(val) {}

   template<typename T, IsNatural D = std::ratio_divide<T, S>>
   constexpr Length(const Length<T>& other)
   : m_val(other.m_val*D::num)
   {}
};
```

```
using km = Length<std::kilo>;
using mm = Length<std::milli>;

constexpr km longdist = 5;
constexpr mm shortdist = 5;
constexpr mm dist1 = longdist;
constexpr km dist2 = shortdist;
```

Wahrscheinlichkeitsverteilungen

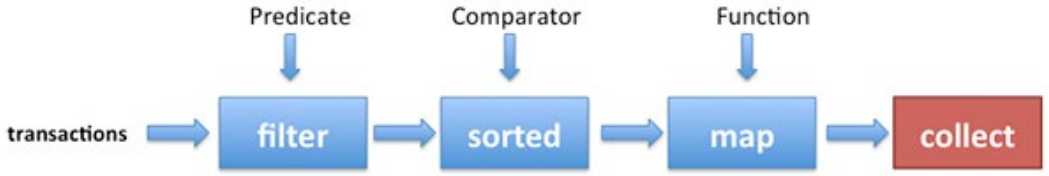
- Zufallszahlgenerator und
  20 verschiedene Wahrscheinlichkeitsverteilungen

```
#include <random>
template<typename D>
void experiment(D* distrib, size_t size) {
   default_random_engine e;
   array<size_t, size> counts;
   for(int i=0; i < 100000; ++i) ++counts[distrib(e)];
   for(auto c: counts) cout << c << endl;
}
int main() {
   // würfelt Zahl zwischen 0..5
   uniform_int_distribution<int> w6(0, 5);
   experiment(w6, 6);
   binomial_distribution<int> muenzen(10);
   experiment(muenzen, 11);
}
```

Ranges und Views

- ranges library
  - eine Erweiterung und Verallgemeinerung der Algorithmen- und Iterator-Bibliotheken, die Algorithmen zusammensetzen lässt und den Einsatz weniger fehleranfällig macht
- range (~ Interface Iterable)
  - ein *concept*, welches sicherstellt, dass der generische Typ einen *begin*- und einen *end*-Iterator besitzt, wobei der end-Iterator von einem anderen Typ und auch unerreichbar sein darf
  - *vector*, *list* oder auch *string* erfüllen das Konzept
- range view
  - leichtgewichtiges Objekt, das **indirekt** eine iterierbare Sequenz (range) mit Zustand darstellt, aber die iterierbaren Elemente nicht besitzt

Beispiel (v1)



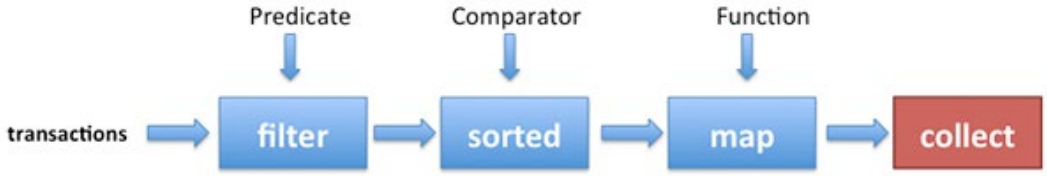
- Java

```
List<Integer> transactionsIds =
    transactions.stream()
    .filter(t -> t.getType() == Transaction.GROCERY)
    .sorted(comparing(Transaction::getValue).reversed())
    .map(Transaction::getId)
    .collect(Collectors.toList());
```

- C++ (Input verändert: sortierte Transaktionen)

```
std::ranges::sort(transactions, std::greater(), &Transaction::m_value);
vector<size_t> transactionIds;
std::ranges::copy(transactions
| views::filter([](auto& t) {
   return t.type == Transaction::Grocery; })
| views::transform(&Transaction::m_id),
  std::back_inserter(transactionIds));
```

Beispiel (v2)

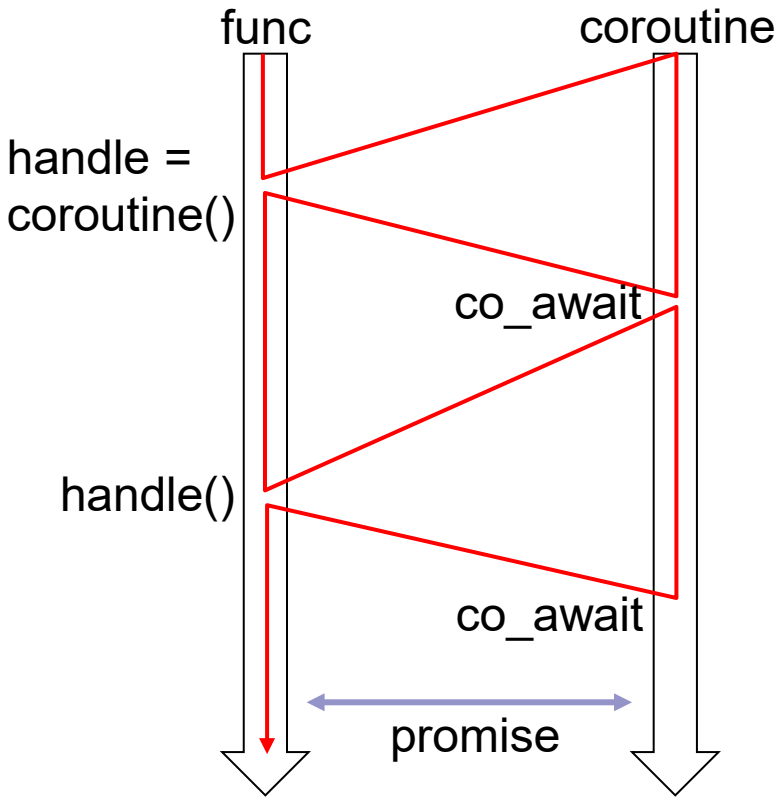


- C++ (unveränderter Input)

```
vector<size_t> indices(transactions.size());

std::ranges::iota(indices, 0);

std::ranges::sort(indices, [&transactions](size_t a, size_t b) {
   return transactions[a].m_value > transactions[b].m_value; });

auto filter = [&transactions](size_t idx) {
   return transactions[idx].m_type == Transaction::Type::Grocery; };

vector<size_t> transactionIds;

std::ranges::copy(indices | views::filter(filter),
   std::back_inserter(transactionIds));
```

Coroutines

- Schlüsselworte
  - co_await *awaitable*
    - Zustand der coroutine zwischenspeichern, handle erzeugen und in der aufrufenden Funktion (func) weiterfahren
    - handle(): coroutine in der Anweisung nach co_await fortsetzen
    - promise: Objekt für den Datenaustausch zwischen func und coroutine
  - co_yield *yieldValue*
    - speichert yieldValue im promise Objekt und führt co_await aus



Asynchrone Verarbeitung (Beispiel)

```
#include <thread>
#include <future>

static size_t fibrec(size_t n) {
   return (n < 2) ? 1 : fibrec(n – 2) + fibrec(n – 1);
}
struct Fibby {
   void operator()(size_t from, size_t to) {
      for(size_t n=from; n <= to; ++n) cout << fibrec(n) << endl;
   }
};
int main() {
   // Initialisieren und Starten in einem Schritt
   thread th1(Fibby(), 28, 33);
   auto th2 = async(fibrec, 35); // asynchron berechnen
   cout << th2.get() << endl;    // Ergebnis abwarten
   th1.join();                   // Ende des Threads abwarten
}
```

Threads als Basis der Parallelität

- Konstruktor und Exekutor
  - thread(*ausführbares Objekt, Parameter der ausführbaren Funktion*)
  - ausführbares Objekt
    - Funktor
    - Lambda
    - Funktionspointer
  - das ausführbare Objekt und die Parameter werden standardmässig kopiert, so dass der Thread auf seinen eigenen Daten arbeiten kann
- Beispiel

```
void printFibs(size_t from, size_t to);
struct Image {
   void fill(int r, int g, int b);
};
```

```
int main() {
    thread th1(printFibs, 28, 35);
    Image img;
    thread th2([&img] {
        img.fill(0,1,2); });
    th1.join(); th2.join();
}
```

Threads und Futures

- Thread
  - low-level
  - Austausch von Daten muss selber synchronisiert werden
  - nicht abgefangene Exceptions in der Thread-Funktion führen zum Abbruch des gesamten Programms
  - thread_local Speicherklasse: statische/globale Variablen werden pro Thread angelegt
- Future
  - Asynchrone Verarbeitung: parallel oder erst beim Aufruf von get()
  - Exceptions tauchen im Vater-Thread auf, wenn das Ergebnis mit get() abgeholt wird
  - wird der Scope des verantwortlichen Futures verlassen, so sorgt der Destruktor dafür, dass die Berechnung problemlos zu Ende geführt wird

async

- Asynchrone Verarbeitung
  - z.B. parallel zu anderen Arbeiten
  - oder nicht beim Starten von async(), sondern erst beim Aufruf von get()
- Rückgabewert von async()
  - future<RT>, wobei RT der Rückgabetyp der asynchron ausgeführten Funktion ist
- Launch Policy (Ausführungsrichtlinie)
  - async(launch::async, langeBerechnung): garantiert parallele Ausführung
  - async(launch::deferred, berechnung): Ausführung bei get()
  - async(launch::any, langeBerechnung): Plattform wählt aus (default)
- Hinter der Kulisse
  - ein future kann auch ohne async() erzeugt werden, dazu muss zuerst ein promise (eine Art Übertragungskanal) erstellt werden

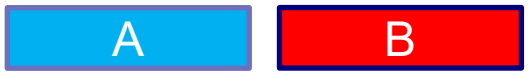
Packaged Task

- packaged_task
  - auszuführende Arbeit wird in einem Task-Objekt zusammengepackt
  - die Ausführung des Pakets kann selber gestartet werden
- Beispiel

```
#include <future>
int main() {
   packaged_task<size_t(void)> task1(bind(&fibrec, 35));
   packaged_task<size_t(size_t)> task2(&fibrec);
   auto f1 = task1.get_future();  // future für 1. Resultatübergabe
   auto f2 = task2.get_future();  // future für 2. Resultatübergabe
   thread th(move(task1));        // 1. Paket parallel ausführen
   task2(35);                     // 2. Paket direkt ausführen
   cout  << f1.get() << endl;     // auf 1. Resultat warten und ausgeben
   cout  << f2.get() << endl;     // auf 2. Resultat warten und ausgeben
   th.join();
}
```

Parallele Algorithmen (1)

- Reihenfolge
  - “sequenced-before” ist eine asymmetrische, transitive Beziehung zwischen zwei Ausführungen im gleichen Thread
  - A is sequenced before B



  - wenn weder A vor B noch B vor A garantiert sind, dann gibt es zwei Möglichkeiten:
    - A und B werden hintereinander (sequentiell) ausgeführt, aber die Reihenfolge ist unbestimmt

    - die Ausführungen von A und B überlagern sich oder erfolgen gleichzeitig

Parallele Algorithmen (2)

- Ausführungs-Policies
  - die meisten Standard-Algorithmen haben eine überladene Methode, welche eine Ausführungs-Policy als ersten Parameter entgegennimmt:

std::execution::seq	Ausführung ist nicht parallel Ausführung der Elementzugriffsfunktionen kann in unbestimmter Reihenfolge erfolgen kann sich von der sequentiellen Version unterscheiden
std::execution::unseq	vektorisierte Ausführung ist möglich in einem Thread ausgeführt können die Operationen auf mehren Daten parallel ausgeführt werden
std::execution::par	Ausführung kann auf mehrere Threads verteilt werden Ausführung der Elementzugriffsfunktionen kann in unbestimmter Reihenfolge erfolgen, auch innerhalb eines einzelnen Threads Data Races und Dead-Locks in den Elementzugriffs-funktionen müssen vermieden werden
std::execution::par_unseq	Ausführung kann parallel oder vektorisiert erfolgen und kann zwischen Threads ausgetauscht werden (work stealing) Ausführung der Elementzugriffsfunktionen ist ungeordnet, über mehrere Threads verteilt und innerhalb eines Threads nicht sequentiell der Einsatz von blockierenden Synchronisationsprimitiven (z.B. Mutex) kann zu Dead-Locks führen

Ausführungs-Policies (Beispiele)

```
int a[] = { 0, 1 };
vector<int> v;
for_each(execution::par, begin(a), end(a), [&](int i) {
    v.push_back(i*2+1);    // Error: data race (vector isn't thread safe)
});

int x = 0;
mutex m;
for_each(execution::par, begin(a), end(a), [&](int) {
    lock_guard<mutex> guard(m);
    ++x;              // Correct, because mutual exclusion is guaranteed
});                   // and sequence among parallel lambdas is irrelevant

for_each(execution::par_unseq, begin(a), end(a), [&](int) {
    lock_guard<mutex> guard(m);     // Error: calls m.lock() and several
                      // of these calls are unsequenced and can interleave
    ++x;
});
```

Lebensdauer von Daten (1)

- Bei parallelen Programmen ist die Kontrolle darüber, welche Daten von den Threads gemeinsam genutzt werden sollen, von zentraler Bedeutung!
  - maximale Sicherheit durch Trennung: thread() kopiert standardmässig alle Parameter
  - spezielle Massnahmen notwendig, um gemeinsame Daten einem Thread zu übergeben
- Beispiele

```
void show(int bgcolor, const Image& image);
```

```
void run1() {
   int red;
   Image img;
   thread th(show, red, img);


   th.join();
}
```

```
void run2() {
   int red;
   Image img;
   thread th([&img](int c) {
      show(c, img);
   }, red);
   th.join();
}
```

Lebensdauer von Daten (2)

- Verwendung von Referenzen ist verlockend, aber …

```
{
   Image pic;
   int red;
   // …
   thread th1([&] { show(red, pic); });  // riskante Referenz
   thread th2([=] { show(red, pic); });  // sichere Kopie
   thread th3(show, red, pic);           // sichere Kopie
   thread th4(show, red, ref(pic));      // Alternative für Referenz
   thread th5(show, red, move(pic));     // Verschieben anstatt Kopie
}                                        // pic wird ungültig
```

Gemeinsamer, ungeschützter Zugriff

```
int counter = 0;


void inc() {
   ++counter;
}
void million() {
   for(int i=0; i<1000000; i++) {
      inc();
   }
}
```

```
int main() {
   thread th1(million);
   thread th2(million);
   thread th3(million);

   for (auto th:{ &th1, &th2, &th3 }) {
      th->join();
   }
   cout << counter << endl;
}
```

Gemeinsamer, geschützter Zugriff

```
int counter = 0;
mutex mtx;

void inc() {
   lock_guard<mutex>(mtx);
   ++counter;
}
void million() {
   for (int i=0; i<1000000; i++) {
      inc();
   }
}
```

```
int main() {
   thread th1(million);
   thread th2(million);
   thread th3(million);

   for (auto th:{ &th1, &th2, &th3 }) {
      th->join();
   }
   cout << counter << endl;
}
```

Synchronisierter Zugriff

- Sobald einer der parallelen Threads gemeinsame Daten verändert, müssen die Datenzugriffe (lesend und schreibend) synchronisiert werden
- Synchronisationsprimitive
  - atomic_*xyz* — alle Zugriffe sind atomar (werden nicht unterbrochen)
  - atomic_flag — atomarer bool, jedoch lock-free
  - once_flag — verwendet in call_once, stellt sicher dass nur einer der parallelen Threads die Funktion ausführen wird
  - mutex — ermöglicht wechselseitigen Ausschluss
  - recursive_mutex — ermöglicht den gleichen Thread mehrfach in den kritischen Abschnitt einzutreten
  - lock_guard — schützt einen kritischen Abschnitt, sehr einfache Anwendung, kennt nur den Zustand locked
  - unique_lock — braucht sein eigenes Mutex, kennt locked und unlocked
  - condition_variable — blockiert den Thread bis er von einem anderen Thread ein Signal zur Fortsetzung erhält

Races

- Problematik
  - Compiler und Prozessoren nehmen sich die Freiheit heraus, Instruktionen und Speicherzugriffe umzuordnen
    - Erhöhung der Performance
    - Probleme bei Nebenläufigkeit
  - Beispiel

CPU 1	CPU 2
while(f == 0) {}	x = 42;
write(x);	f = 1;

  - Beispiel x = 0, y = 0

CPU 1	CPU 2
x = 1;	y = 1;
a = y;	b = x;

Singleton (single threaded)

```
class Singleton {
   static Singleton* s_instance;
public:
   static Singleton* instance();
};
Singleton* Singleton::s_instance = nullptr;

Singleton* Singleton::instance() {
   if (s_instance == nullptr) {
      s_instance = new Singleton();
   }
   return s_instance;
}
```

Singleton (double-checked locking)

```
class Singleton {
   static Singleton* s_instance;
public:
   static Singleton* instance();
};
Singleton* Singleton::s_instance = nullptr;

Singleton* Singleton::instance() {
   static mutex mtx;
   if (s_instance == nullptr) {
      lock_guard<mutex> lk(mtx);
      if (s_instance == nullptr) {
         s_instance = new Singleton();
      }
   }
   return s_instance;
}
```

Singleton (unter Einsatz von atomic)

```
class Singleton {
    static atomic<Singleton*> s_instance;
public:
    static Singleton* instance();
};
atomic<Singleton*> Singleton::s_instance = nullptr;

Singleton* Singleton::instance() {
    static mutex mtx;
    Singleton *p = s_instance;
    if (p == nullptr) {
        lock_guard<mutex> lk(mtx);
        if (p == nullptr) {
            p = new Singleton();
            s_instance = p;
        }
    }
    return p;
}
```

Singleton (unter Einsatz von fences)

```
class Singleton {
    static atomic<Singleton*> s_instance;
public:
    static Singleton* instance();
};
atomic<Singleton*> Singleton::s_instance = nullptr;

Singleton* Singleton::instance() {
    static mutex mtx;
    Singleton *p = atomic_load_explicit(&s_instance, memory_order_acquire);
    if (p == nullptr) {
        lock_guard<mutex> lk(mtx);
        if (p == nullptr) {
            p = new Singleton();
            atomic_store_explicit(&s_instance, p, memory_order_release);
        }
    }
    return p;
}
```