

Hauptseminar: Rechnerarchitektur und Programmierung

Neuerungen in C++11, 14 und 17 im Kontext Performance-Analyse
von HPC-Anwendungen

Jan Stephan (jan.stephan@mailbox.tu-dresden.de)

Betreuer: Ronny Brendel (ronny.brendel@tu-dresden.de)

07. Juli 2016

- Einleitung
 - C++11 und C++14
 - moderne C++-Philosophie
- Beispiel: Vektoraddition mit CUDA
 - **auto**, **constexpr**, **using**, smart pointers, variadische Templates
- Beispiel: Producer-Consumer-Queue
 - Lambdas, **thread**, **mutex**, **atomic**
- Empfehlungen für die Unterstützung in Score-P und Vampir
- Ausblick auf C++17
 - Standardisierungsprozess
- weiterführende Literatur
 - schriftliche Ausarbeitung

● C++11:

- zahlreiche Änderungen bei voller Rückwärtskompatibilität (C++98/C++03)
- neue Features hauptsächlich in Standardbibliothek
- Ziel: ansprechend für Anfänger, nützlich für Experten

● C++14:

- „Bugfix“-Release
- kleinere Änderungen an und Ergänzungen für C++11

C++ feels like a new language. That is, I can express my ideas more clearly, more simply, and more directly in C++11 than I could in C++98. Furthermore, the resulting programs are better checked by the compiler and run faster.

- Bjarne Stroustrup[TCPP]

Drücke Ideen direkt im Code aus!

[CPPCG](P.1)

```
void do_something(vector<string>& v)
{
    string val;
    cin >> val;
    // ...
    int index = -1; // unnötig kompliziert
    for(int i = 0; i < v.size(); ++i)
    {
        if(v[i] == val)
        {
            index = i;
            break;
        }
    }
    // ...
}
```

```
void do_something(vector<string>& v)
{
    string val;
    cin >> val;
    // ...
    auto p = find(begin(v), end(v), val);
    // ...
}
```

Drücke Deine Absicht aus!

[CPPCG](P.3)

Moderne C++-Philosophie → Iteration in C++03

```
int i = 0;
while(i < v.size())
{
    // tue etwas mit v[i]...
}
```

```
for(const auto& x : v) { /* tue etwas mit x */ }  
for(auto&& x : v) { /* tue etwas mit x */ }  
for_each(begin(v), end(v), [](int x) { /* tue etwas mit x */ });  
for_each(execution::par, begin(v), end(v), [](int x) {  
    /* tue etwas mit x */  
});
```

Vermeide Ressourcenlecks!

[CPPCG](P.8)

Moderne C++-Philosophie → Ressourcenakquise in C

```
void f(char* name)
{
    FILE* input = fopen(name, "r");
    // ...
    if(something) return;
    // ...
    fclose(input);
}
```



Moderne C++-Philosophie → Ressourcenakquise in C++

```
void f(char* name)
{
    ifstream input(name);
    // ...
    if(something) return;
    // ...
}
```

→ RAI (Resource Acquisition Is Initialization)

Benutze die Standardbibliothek!

[STC]

Moderne C++-Philosophie → Standardbibliothek

- `char*-Strings` → `std::string`
- `[]-Arrays` → `std::vector` / `std::array`
- Zeigerarithmetik → `<algorithm>`
- `new / delete` → `make_shared` / `make_unique`

Moderne C++-Philosophie → Folgen

- höherer Abstraktionsgrad
- weniger Quelltext
- messbar schnelleres Programm [IPM] → schriftliche Ausarbeitung

- Einleitung
 - C++11 und C++14
 - Moderne C++-Philosophie
- *Beispiel: Vektoraddition mit CUDA*
 - `auto`, `constexpr`, `using`, `enum class`, smart pointers
- Beispiel: Producer-Consumer-Queue
 - Lambdas, `thread`, `mutex`, `atomic`
- Empfehlungen für die Unterstützung in Score-P und Vampir
- Ausblick auf C++17
 - Standardisierungsprozess
- weiterführende Literatur
 - schriftliche Ausarbeitung

Vektoraddition mit CUDA → Kernel (alt)

```
__global__ void vector_add(const int* __restrict__ a,  
                           const int* __restrict__ b,  
                           int* __restrict__ c,  
                           size_t size)  
{  
    int i = threadIdx.x + blockIdx.x * blockDim.x;  
    if(i < size)  
        c[i] = a[i] + b[i];  
}
```

Vektoraddition mit CUDA → Kernel (neu)

```
__global__ void vector_add(const std::int32_t* __restrict__ a,  
                           const std::int32_t* __restrict__ b,  
                           std::int32_t* __restrict__ c,  
                           std::size_t size)  
{  
    auto i = threadIdx.x + blockIdx.x * blockDim.x;  
    if(i < size)  
        c[i] = a[i] + b[i];  
}
```

- Bibliothekserweiterung im C++11-Standard: `<cstdint>` [C++11](§18.4.1)
- enthält die aus dem C99-Standard bekannten Integer-Typen (`<stdint.h>`)

Neuerung in C++11: auto

- Spracherweiterung im C++11-Standard: `auto` [CPP11](§7.1.6.4)

- für Variablen:

```
auto x = 0; // wenn der Datentyp unwichtig ist
auto f = float(0); // bei Festlegung auf einen Datentypen
auto two = std::sqrt(4); // Herleitung aus Rückgabetypen
```

- für Funktionen:

```
auto foo(std::int32_t a, std::int32_t b) -> std::int32_t;

template <class T, class U>
auto bar(T a, U b) -> decltype(a + b);
```

Vektoraddition mit CUDA → Vektorerstellung (alt)

```
#define SIZE = 1000;

int* host_a = new int[SIZE];
int* host_b = new int[SIZE];

for(size_t i = 0; i < SIZE; ++i)
{
    host_a[i] = std::rand();
    host_b[i] = std::rand();
}

// ...

delete[] host_b;
delete[] host_a;
```

Vektoraddition mit CUDA → Vektorerstellung (neu)

```
constexpr auto size = 1000;  
  
auto host_a = std::make_unique<std::uint32_t[]>(size);  
auto host_b = std::make_unique<std::uint32_t[]>(size);  
  
std::generate(a.get(), a.get() + size, std::rand);  
std::generate(b.get(), b.get() + size, std::rand);  
  
// ...
```

Neuerung in C++11: constexpr

- Spracherweiterung im C++11-Standard: constexpr [CPP11](§7.1.5)

- für Variablen:

```
constexpr auto i = 0; // Wert steht zur Compilezeit fest
```

- für Funktionen:

```
constexpr auto max(std::int32_t a, std::int32_t b) -> std::int32_t  
{  
    // Auswertung zur Compile-Zeit möglich  
    return (a > b) ? a : b;  
}
```

- kombiniert:

```
constexpr auto i = max(2, 3); // Auswertung zur Compile-Zeit  
  
// Auswertung zur Compile-Zeit oder zur Laufzeit  
auto j = max(x, y);
```


Neuerung in C++11: Smart pointers

- Bibliothekserweiterung im C++11-Standard: smart pointers [C++11](§20.7)

- `unique_ptr`
- `shared_ptr`
- `weak_ptr`

- veraltet: `auto_ptr`

- Speicherallokation:

```
auto unique = std::make_unique<std::int32_t[]>(size); // C++14
auto shared = std::make_shared<std::int32_t[]>(size); // C++11
auto weak = std::weak_ptr<std::int32_t[]>(shared); // C++11
```

- Speicherfreigabe: automatisch dank RAII
- `new/delete` überflüssig

Vektoraddition mit CUDA → Speicherverwaltung

```
int* dev_a = NULL;
cudaMalloc((void**) &dev_a, SIZE * sizeof(int));

// ...

cudaFree(dev_a);

/* besser: */
auto dev_a = make_device_ptr<std::int32_t>(size);

// ...
```

Vektoraddition mit CUDA → Speicherverwaltung

```
// host_ptr analog
template <class T>
auto make_device_ptr(std::size_t size) -> device_ptr<T>
{
    auto p = static_cast<T*>(nullptr);
    cudaMalloc(reinterpret_cast<void**>(&p), size * sizeof(T));
    return device_ptr<T>(p);
}

struct device_deleter
{
    auto operator()(void* p) -> void { cudaFree(p); }
};

template <class T>
using device_ptr = std::unique_ptr<T[], device_deleter>;
```

- Spracherweiterung in C++11: using [C++11](§7.1.3/2)

```
typedef int my_type; // alt
using my_type = int; // neu

typedef void (*func_ptr)(double); // alt
using func_ptr = void (*)(double); // neu

// neu, funktioniert nicht mit typedef
template <class T>
using my_vec = std::vector<T, my_allocator>;
```

Vektoraddition mit CUDA → Kopiervorgang

```
cudaMemcpy(dev_a, host_a, SIZE * sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(dev_b, host_b, SIZE * sizeof(int), cudaMemcpyHostToDevice);

// ...

cudaMemcpy(host_c, dev_c, SIZE * sizeof(int), cudaMemcpyDeviceToHost);

/* angepasst: */

cudaMemcpy(dev_a.get(), host_a.get(), size * sizeof(std::int32_t),
            cudaMemcpyHostToDevice);
cudaMemcpy(dev_b.get(), host_b.get(), size * sizeof(std::int32_t),
            cudaMemcpyHostToDevice);

// ...

cudaMemcpy(host_c.get(), dev_c.get(), size * sizeof(std::int32_t),
            cudaMemcpyDeviceToHost);
```

Vektoraddition mit CUDA: Kernelaufruf

```
int block_size = 1024;
int grid_size = (int) std::ceil((float) SIZE / block_size);
vec_add<<grid_size, block_size>>>(dev_a, dev_b, dev_c, SIZE);
/* besser: */
cuda_launch(vec_add, dev_a.get(), dev_b.get(), dev_c.get(), size);
```

Vektoraddition mit CUDA: Kernelaufruf

```
template <class... Args>
auto cuda_launch(void (*kernel)(Args...), Args... args) -> void
{
    // berechne grid_size und block_size
    kernel<<<grid_size, block_size>>>(args...);
}
```

- Spracherweiterung in C++11: variadische Templates [CPP11](§14.2.15)
- „Parameter-Packs“ werden zur Compilezeit entpackt
- Funktionsweise ähnlich wie variadische Funktionen aus C

Vektoraddition mit CUDA → Zusammenfassung

```
constexpr auto size = 1000;

auto host_a = make_host_ptr<std::int32_t>(size);
auto host_b = make_host_ptr<std::int32_t>(size);
auto host_c = make_host_ptr<std::int32_t>(size);

auto dev_a = make_device_ptr<std::int32_t>(size);
auto dev_b = make_device_ptr<std::int32_t>(size);
auto dev_c = make_device_ptr<std::int32_t>(size);

std::generate(host_a.get(), host_a.get() + size, std::rand);
std::generate(host_b.get(), host_b.get() + size, std::rand);

cudaMemcpy(dev_a.get(), host_a.get(), size * sizeof(std::int32_t),
            cudaMemcpyHostToDevice);
cudaMemcpy(dev_b.get(), host_b.get(), size * sizeof(std::int32_t),
            cudaMemcpyHostToDevice);

cuda_launch(vec_add, dev_a.get(), dev_b.get(), dev_c.get(), size);

cudaMemcpy(host_c.get(), dev_c.get(), size * sizeof(std::int32_t),
            cudaMemcpyDeviceToHost);
```

- Einleitung
 - C++11 und C++14
 - Moderne C++-Philosophie
- Beispiel: Vektoraddition mit CUDA
 - `auto`, `constexpr`, `using`, `enum class`, smart pointers
- *Beispiel: Producer-Consumer-Queue*
 - Lambdas, `thread`, `mutex`, `atomic`
- Empfehlungen für die Unterstützung in Score-P und Vampir
- Ausblick auf C++17
 - Standardisierungsprozess
- weiterführende Literatur
 - schriftliche Ausarbeitung

Fallbeispiel: Producer-Consumer-Queue

- Aufgabe: Implementierung einer Producer-Consumer-Queue[PPP]
- Probleme vor C++11:
 - kein plattformunabhängiger Weg (Pthreads, Win32-Threads, ...)
 - kein nativer „C++-Weg“ (RAII [insbesondere Destruktoren], ...), Wrapper erforderlich
 - keine plattformunabhängige Unterstützung für Atomics
- seit C++11:
 - `<thread>` zur Erzeugung von Threads
 - `<mutex>` und `<condition_variable>` zur Threadsynchronisierung
 - `<atomic>` für atomare Operationen
 - `<future>` für asynchrone Operationen (ohne explizite Threaderzeugung durch Programmierer, hier nicht gezeigt)

Producer-Consumer-Queue: Threaderzeugung

```
auto main() -> int
{
    auto q = queue();
    auto p = std::thread([&q]() { q.push(create_object()); });
    auto c = std::thread([&q]() { use_object(q.pop()); });

    p.join();
    c.join();

    return 0;
}
```

Neuerung in C++11: Lambdas

- Neuerung im C++11-Standard: Lambdas [CPP11](§5.1.2)
- einfacher Weg, um simple Funktionen zu erzeugen

```
for_each(begin(v), end(v), [](auto& x) { ++x; });
```

```
auto val = 5;
```

```
transform(begin(in), end(in), begin(out), [=](const auto& x) {  
    return x + val; }); // val ist eine Kopie
```

```
transform(begin(in), end(in), begin(out), [&](const auto& x) {  
    return x + val; }); // val ist eine Referenz
```

Producer-Consumer-Queue: Implementierung mit Mutexes

```
template <class T>
class Queue {
    auto push(T t) -> void {
        auto lock = unique_lock<mutex>(mutex_);

        queue_.push(move(item));
        cv_.notify_one();
    }

    auto pop() -> T {
        auto lock = unique_lock<mutex>(mutex_);
        while(queue_.empty())
            cv_.wait(lock);

        auto ret = move(queue_.front());
        queue_.pop();

        return ret;
    }

    condition_variable cv_;
    mutex mutex_;
}
```

Producer-Consumer-Queue: Implementierung mit Atomics

```
template <class T> class Queue {  
    auto push(T t) -> void {  
        while(lock_)  
            thread::this_thread::yield();  
  
        lock_ = true;  
        queue_.push(move(t));  
        lock_ = false;  
    }  
  
    auto pop() -> T {  
        while(lock_ || queue_.empty())  
            thread::this_thread::yield();  
  
        lock_ = true;  
        auto ret = move(queue_.front());  
        queue_.pop();  
        lock_ = false;  
        return ret;  
    }  
  
    std::atomic<bool> lock_; // Initialisierung: false  
};
```

- Einleitung
 - C++11 und C++14
 - Moderne C++-Philosophie
- Beispiel: Vektoraddition mit CUDA
 - `auto`, `constexpr`, `using`, `enum class`, smart pointers
- Beispiel: Producer-Consumer-Queue
 - Lambdas, `thread`, `mutex`, `atomic`
- *Empfehlungen für die Unterstützung in Score-P und Vampir*
- Ausblick auf C++17
 - Standardisierungsprozess
- weiterführende Literatur
 - schriftliche Ausarbeitung

Empfehlungen für die Unterstützung in Score-P und Vampir

■ Probleme:

- Inlining der meisten STL-Methoden (Templates)
- keine Unterstützung von Threads in gelinkten Bibliotheken
- keine Unterstützung von Datei-E/A (auch nicht in C)
- `std::mutex` wird als Pthread betrachtet
- unintuitive Lambda-Namen
- teilweise fehlerhaftes Demangling von Lambdas (auto-Parameter)

■ Screenshots

● Lösungsvorschläge:

- Instrumentierung der Standardbibliothek-Interfaces
- Alternative: Einrichtung eines Zählers, der die Aufrufe der STL-Methoden hochzählt
 - nur die vom Programmierer aufgerufenen Methoden!
- nachträgliche Registrierung „externer“ Threads (statt Programmabsturz)

- Einleitung
 - C++11 und C++14
 - Moderne C++-Philosophie
- Beispiel: Vektoraddition mit CUDA
 - `auto`, `constexpr`, `using`, `enum class`, smart pointers
- Beispiel: Producer-Consumer-Queue
 - Lambdas, `thread`, `mutex`, `atomic`
- Empfehlungen für die Unterstützung in Score-P und Vampir
- *Ausblick auf C++17*
 - Standardisierungsprozess
- weiterführende Literatur
 - schriftliche Ausarbeitung

Ausblick auf C++17: Standardisierungsprozess

- seit 2012: dezentraler Arbeitsprozess
- verschiedene Arbeitsgruppen arbeiten an unterschiedlichen Projekten
 - *Programming Language C++*: Weiterentwicklung der Sprache
 - *Library Fundamentals*: fundamentale Aspekte der Standardbibliothek
 - *Networking*: Netzwerkfunktionalität
 - *Parallelism*: parallelisierbare und/oder vektorisierbare Algorithmen
 - *Ranges*: range-for statt Iteratoren für Container und die STL
 - *Modules*: Ersatz für `#include <header.h>`
 - *Concurrency*: Verbesserung der asynchronen STL-Funktionen

Ausblick auf C++17: Spracherweiterungen

- if mit Initialisierung

```
std::map<int, std::string> m;
```

```
// alt  
auto it = m.find(10);  
if(it != m.end()) { return it->size(); }
```

```
//neu  
if(auto it = m.find(10); it != m.end()) { return it->size(); }
```

- „structured bindings“

```
struct S { int x; volatile double y; };  
S f();  
const auto [x, y] = f();
```

```
foo(x); // x hat den Typ const int  
bar(y); // y hat den Typ const volatile double
```

- kombiniert:

```
if(auto [x, y, z] = foo(); x.valid()) { bar(y, z); }
```

- C11-Standard statt C99-Standard als Basis
- zusätzliche mathematische Funktionen (Polynomfunktionen, Betafunktion, elliptische Integrale, ...)
- Routinen zum Umgang mit dem Dateisystem (Dateien und Ordner)
- neue Algorithmen für `<algorithm>` (`sample`, `reduce`, `transform_reduce`, ...)
- optionale Parallelisierung der Algorithmen in `<algorithm>`

- Einleitung
 - C++11 und C++14
 - Moderne C++-Philosophie
- Beispiel: Vektoraddition mit CUDA
 - `auto`, `constexpr`, `using`, `enum class`, smart pointers
- Beispiel: Producer-Consumer-Queue
 - Lambdas, `thread`, `mutex`, `atomic`
- Empfehlungen für die Unterstützung in Score-P und Vampir
- Ausblick auf C++17
 - Standardisierungsprozess
- *weiterführende Literatur*
 - schriftliche Ausarbeitung

Weiterführende Literatur

- Bjarne Stroustrup: *A Tour of C++*, Addison-Wesley, 2014
- Bjarne Stroustrup: *The C++ Programming Language*, Addison-Wesley, 2013
- Scott Meyers: *Effective Modern C++*, O'Reilly, 2014
- Anthony Williams: *C++ Concurrency in Action*, Manning, 2012
- Stanley B. Lippmann, Josee Lajoie, Barbara E. Moo: *C++ Primer*, Addison-Wesley, 2012
- Bjarne Stroustrup: *C++ 11 FAQ*, <http://www.stroustrup.com/C++11FAQ.html>
- Herb Sutter: *Guru of the Week*, <https://herbsutter.com/gotw/>

Weiterführende Literatur → schriftliche Ausarbeitung

- weitere (kleinere) Beispielprogramme
- Vergleich mit Implementierungen in C und C++
- hier fehlende Features
- ausführlicherer Ausblick auf C++17

- [CPP11] ISO/IEC JTC1/SC22/WG21: *International Standard ISO/IEC 14882:2011(E) – Programming Language C++*, ISO/IEC, 2011
- [CPP14] ISO/IEC JTC1/SC22/WG21: *International Standard ISO/IEC 14882:2014(E) – Programming Language C++*, ISO/IEC, 2014
- [CPP17] ISO/IEC JTC1/SC22/WG21: *Working Draft, Standard for Programming Language C++*, ISO/IEC, Entwurf vom 30.05.2016
- [MCS] Herb Sutter: *Back to Basics: Modern C++ Style*, Folien zum Vortrag auf der CppCon 2014, letzte Änderung am 13.09.2014, zuletzt abgerufen am 12.06.2016,
<https://github.com/CppCon/CppCon2014/tree/master/Presentations/Back%20to%20the%20Basics!%20Essentials%20of%20Modern%20C%2B%2B%20style>
- [TCPP] Bjarne Stroustrup: *A Tour of C++*, Addison-Wesley, 2014
- [CPPCG] Bjarne Stroustrup, Herb Sutter: *C++ Core Guidelines*, letzte Aktualisierung am 05. April 2016, zuletzt abgerufen am 22.06.16,
<http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
- [PPP] Timothy G. Mattson, Beverly A. Sanders, Berna L. Massingill: *Patterns for Parallel Programming*, Addison-Wesley, 2005

Quellen

- [STC] Kate Gregory: *Stop Teaching C*, Folien zum Vortrag auf der CppCon 2015, letzte Änderung am 03.10.2015, zuletzt abgerufen am 29.06.16, <https://github.com/CppCon/CppCon2015/tree/master/Presentations/Stop%20Teaching%20C>
- [IPM] J. Daniel Garcia, Bjarne Stroustrup: *Improving performance and maintainability through refactoring in C++11*, August 2015

Vielen Dank!

