TECHNISCHE UNIVERSITÄT DRESDEN

Zentrum für Informationsdienste und Hochleistungsrechnen (ZIH)

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)

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

07. Juli 2016

ZIH
Zentrum für Informationsdienste und Hochleistungsrechnen

Moderne C++-Philosophie

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

Gliederung

- Einleitung
  - Moderne C++-Philosophie
  - Messmethodik
- Fallbeispiel: Vielfache von 3 und 5
  - Spracherweiterung: `constexpr`
- Fallbeispiel: Names scores
  - Spracherweiterungen: `auto`, range-for, Lambda-Ausdrücke
  - Bibliothekserweiterung: `forward_list`
- Fallbeispiel: Producer-Consumer-Queue
  - Bibliothekserweiterungen: `thread`, `mutex`, `atomic`
- Empfehlungen für die Unterstützung in Score-P und Vampir
- Ausblick auf C++17
- Weiterführende Literatur

Moderne C++-Philosophie

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]

Moderne C++-Philosophie

Drücke Ideen direkt im Code aus!

[CPPCG](P.1)

Moderne C++-Philosophie

```
void do_something(string<vector>& 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;
        }
    }
    // …
}
```

Moderne C++-Philosophie

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

Moderne C++-Philosophie

Drücke Deine Absicht aus!

[CPPCG](P.3)

Moderne C++-Philosophie

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

Moderne C++-Philosophie

```
for(const auto& x : v) { /* tue etwas mit x */ }

for(auto& x : v) { /* tue etwas mit x */ }

for_each(v, [](int x) { /* tue etwas mit x */ });

for_each(parallel.v, [](int x) { /* tue etwas mit x */ });
```

Moderne C++-Philosophie

Erzeuge keine Ressourcenlecks!

[CPPCG](P.8)

Moderne C++-Philosophie

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

Moderne C++-Philosophie

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

→ RAII (Resource Acquisition Is Initialization)

Moderne C++-Philosophie

Benutze die Standardbibliothek!

[STC]

Moderne C++-Philosophie

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

Moderne C++-Philosophie

- Höherer Abstraktionsgrad
- Weniger Quelltext
- Messbar schnelleres Programm

[IPM]

Messmethodik

- System:
  - Betriebssystem: Arch Linux (x86_64)
  - Kernel 4.6.2
  - Compiler: gcc 6.1.1, clang 3.8.0
  - Prozessor: Intel Core i5-3230M (2,6GHz, Turbo Boost ausgeschaltet)
- Messung:
  - `clock_gettime(CLOCK_MONOTONIC, …);`

Median/Minimum/Maximum

Beispiel: Vielfache von 3 und 5

Project Euler: Problem 1 [EP1]

Gesucht ist die Summe aller Vielfachen von 3 oder 5, die kleiner als 1000 sind.

Beispiel: Vielfache von 3 und 5 → Implementierung in C

```
int sum = 0;
for(int i = 0; i < 1000; ++i)
{
    if((i % 3 == 0) || (i % 5 == 0))
    sum += i;
}
```

- Compilerflags: `-O3`
- Ausgabe:

```
Sum: 233168
Time elapsed: 4184ns
```

- Compilerflags: `-O0`
- Ausgabe:

```
Sum: 233168
Elapsed time: 10458ns
```

- Problem: Berechnung findet zur Laufzeit statt

Beispiel: Vielfache von 3 und 5 → Implementierung in C++03

- Optimierungsansatz in klassischem C++: die Berechnung bereits vom Compiler durchführen lassen

  → Einsatz von Template-Metaprogrammierung

```
template <int i, bool = ((i % 3 == 0) || (i % 5 == 0))>
struct add_multiples {
    const static int value = add_multiples<i - 1>::value;
};

template <int i> struct add_multiples<i, true> {
    const static int value = add_multiples<i - 1>::value + i;
}

template <> struct add_multiples<0> {
    const static int value = 0;
};

// in main()

int sum = add_multiples<999>::value;
```

Beispiel: Vielfache von 3 und 5 → Implementierung in C++03

- Compilerflags: `-O3 -std=c++03 -ftemplate-depth=1000`
- Ausgabe:

```
Sum: 233168
Time elapsed: 301ns
```

- Vorteil: keine Berechnung zur Laufzeit, im Maschinencode steht das Ergebnis

```
movl $233168, -20(%rbp)
```

- Probleme:
  - unintuitive Syntax → kryptische Templates und wirkt nicht wie Funktionsaufruf
  - mehr Quelltext → erhöhter Wartungsaufwand
  - statisch: kann nur zur Compilezeit verwendet werden

Neuerung in C++11: constexpr

- Spracherweiterung im C++11-Standard: `constexpr` [CPP11](§7.1.5)
- Hinweis an den Compiler, dass der Wert der Variable oder Funktion zur Compilezeit ausgewertet werden kann
- solcherart markierte Variablen und Funktionen können überall dort verwendet werden, wo (zur Compilezeit) konstante Ausdrücke gebraucht werden
  - Arraydeklarationen
  - Template-Parameter
  - Konstruktoren für einfache Datenstrukturen
- Funktionen können auch zur Laufzeit aufgerufen werden, sofern der Compiler diese Aufrufe nicht wegoptimiert

Beispiel: Vielfache von 3 und 5 → Implementierung in C++11

```
constexpr int add_multiples(int i) // C++11: genau ein return-Ausdruck
{
    return (i == 0) ?
        0 :
        (((i % 3 == 0) || (i % 5 == 0)) ?
            i + add_multiples(i - 1) :
            add_multiples(i - 1));
}

constexpr int add_multiples(int limit) // Limitierung entfällt ab C++14
{
    int sum = 0;
    for(int i = 0; i < limit; ++i)
    {
        if((i % 3 == 0) || (i % 5 == 0))
            sum += i;
    }
    return sum;
}

// in main()
constexpr int sum = add_multiples(999); // Berechnung zur Compilezeit
int sum2 = add_multiples(999); // evtl. Berechnung zur Laufzeit
```

Beispiel: Vielfache von 3 und 5 → Implementierung in C++11

- Compilerflags: `-O3 -std=c++14`
- Ausgabe:

```
Result: 233168
Elapsed time: 260ns
Result 2: 233168
Elapsed time: 124ns
```

- Compilerflags: `-O0 -std=c++14`
- Ausgabe:

```
Result: 233168
Elapsed time: 314ns
Result 2: 233168
Elapsed time: 10480ns
```

- Vorteile:
  - Berechnung kann zur Compilezeit erfolgen
  - falls erforderlich, kann die Funktion zur Laufzeit aufgerufen werden
  - Syntax und Codemenge wie bei einer normalen Funktion

constexpr: Kompatibilität zu Score-P und Vampir

- `constexpr` impliziert `inline` [CPP14](7.1.5/2)
- Compiler wird Funktionen, die mit `constexpr` gekennzeichnet sind, automatisch in den Funktionsrumpf der aufrufenden Funktion einbauen
- Score-P kann Funktionsaufrufe dieser Art nicht automatisch erkennen, Vampir diese also auch nicht anzeigen
- Nutzen fragwürdig, da die C-Äquivalente in der Regel ebenfalls nicht betrachtet werden könnten

Hier Screenshot einfügen

Beispiel: Names scores

Project Euler: Problem 22

Aus einer Datei sollen ca. 5000 Vornamen ausgelesen und danach sortiert werden. Für jeden Namen soll der alphabetische Wert (Summe der Werte der Buchstaben, wobei A = 1, B = 2, etc.) berechnet und dieser dann mit der Position in der sortierten Liste multipliziert werden. Gesucht ist die Summe der Ergebnisse.

Beispiel: Names scores → Implementierung in C++03

```
using namespace std;

struct alphabet_val
{
    int operator()(int acc, char c) { return acc + (c - 64); }
};

// in main()
list<string> names;

// hier Datei öffnen und einlesen

names.sort();
size_t i = 1;
size_t total = 0;

for(list<string>::iterator it = names.begin(); it != names.end(); ++it, ++i)
{
    string& s = *it;
    int alpha_val = accumulate(s.begin(), s.end(), 0, alphabet_val());
    total += alpha_val * i;
}
```

Beispiel: Names scores → Implementierung in C++03

- Compilerflags: `-O3 -std=c++03`
- Ausgabe:

```
Total: 871198282
Elapsed time: 3925µs
```

- Probleme:

1) `std::list` ist relativ unflexibel (kein Random Access Iterator) und langsam (bidirektionaler Iterator). Austausch aber aufwendig, bedingt durch:
2) schlechte Wartbarkeit (Abhängigkeiten von `ranges` Datentyp im Quelltext)
3) Functor nötig, um eine Funktion auf alle Buchstaben eines `std::strings` anzuwenden
4) korrekter Umgang mit Iteratoren relativ kompliziert

Beispiel: Names scores → Implementierung in C++14

```
using namespace std;

// in main()
auto names = forward_list<string>{};

// hier Datei öffnen und einlesen

names.sort();
auto i = 1;
auto total = 0;


for(auto& s : names)
{
    auto alpha_val = accumulate(begin(s), end(s), 0,
        [](int acc, char c) { return acc + (c - 64) });
    total += alpha_val * i;
    ++i;
}
```

Beispiel: Names scores → Implementierung in C++14

- Compilerflags: `-O3 -std=c++14`
- Ausgabe:

```
Total: 871198282
Elapsed time: 3555µs
```

- Vorteile:
  - kein Functor nötig, kurze, eindeutige Berechnungsvorschrift am Ort der Anwendung
  - keine umständliche Iteratorensyntax, stattdessen klare Absichtsbekundung seitens des Programmierers (für jedes Element im Container…)
  - unabhängig vom Datentyp des Containers und des enthaltenen string-Objektes
  - keine Leistungseinbußen trotz höheren Abstraktionsgrades

Neuerung in C++11: auto

- Spracherweiterung im C++11-Standard: `auto` [CPP11](§7.1.6.4)
- `auto` signalisiert dem Compiler, dass der Typ einer Variablen aus deren Initialisierung hergeleitet wird:

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

- C++11: `auto` signalisiert dem Compiler, dass bei Funktionsdeklarationen der Rückgabetyp nachfolgt:

```
auto foo(int a, int b) -> int;
```

- C++14: `auto` ermöglicht dem Compiler, den Rückgabetypen einer Funktion selbst zu bestimmen (Funktionsrumpf muss sichtbar sein) [CPP14](§7.1.6.4/4):

```
auto foo(int a, int b); // Rückgabetyp unbekannt
auto foo(int a, int b) { … } // Rückgabetyp hergeleitet
```

Neuerung in C++11: auto

- Vorteile [MCS](Folie 13ff.)
  - Korrektheit

```
void f(const vector<int>& v) {
    vector<int>::iterator i = v.begin(); // Fehler
    vector<int>::const_iterator i = v.begin(); // korrekt
    auto i = v.begin(); // korrekt + kein Denkaufwand
}
```

  - Wartbarkeit

```
int i = f(1, 2, 3) * 42; // Datentyp korrekt
int i = f(1, 2, 3) * 42.0; // Datentyp verliert an Genauigkeit
auto i = f(1, 2, 3) * 42.0; // Datentyp korrekt
```

  - Leistung
    - automatische Herleitung garantiert, dass keine implizite Typkonversion stattfindet
  - Benutzbarkeit
    - Lambdas, `std::container<VeryLongTypeName>::const_iterator`, …
  - Bequemlichkeit (unwichtig, aber praktisch)

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 producer(queue<object>& q) -> void
{
    auto o = create_object();
    q.push(move(o));
}

auto consumer(queue<object>& q) -> void
{
    auto o = q.pop();
    do_something_with(o);
}

auto main() -> int
{
    auto q = queue();
    auto p = thread(&producer, ref(q)); // thread kopiert Parameter
    auto c = thread(&consumer, ref(q)); // ref erzwingt Referenz
    p.join();
    c.join();

    return 0;
}
```

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_)
            thread::this_thread::yield();

        lock_ = true;
        auto ret = move(queue_.front());
        queue_.pop();
        lock_ = false;

        return ret;
    }

    atomic_bool lock_; // Initialisierung: false
};
```

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*, 2012

- Bjarne Stroustrup: *C++11 FAQ*, http://www.stroustrup.com/C++11FAQ.html
- Herb Sutter: *Guru of the Week*, https://herbsutter.com/gotw/

Quellen

- [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
- [EP1] Project Euler: *Problem 1. Multiples of 3 and 5*, letzte Aktualisierung unbekannt, zuletzt abgerufen am 12.06.2016, https://projecteuler.net/problem=1
- [EP22] Project Euler: *Problem 22. Names scores*, letzte Aktualisierung unbekannt, zuletzt abgerufen am 12.06.2016, https://projecteuler.net/problem=22
- [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%2yle

Quellen

- [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
- [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!