Eine Einführung in modernes C++ mit CMake

Paul Nykiel

March 6, 2019

Einleitung

Was ist C++

- ~~C with classes~~

Was ist C++

- ~~C with classes~~
- C++11!

Was ist C++

- ~~C with classes~~
- C++11!
- Standardisiert und offen

Was ist C++

- ~~C with classes~~
- C++11!
- Standardisiert und offen
- Wird in quasi jeder Domäne genutzt

Was ist C++

- ~~C with classes~~
- C++11!
- Standardisiert und offen
- Wird in quasi jeder Domäne genutzt
- Ziele: Sicherer und performanter Code

C++ im Vergleich zu Java, C#

- Undefiniertes Verhalten

C++ im Vergleich zu Java, C#

- Undefiniertes Verhalten
- Keine automatische Speicherverwaltung

C++ im Vergleich zu Java, C#

- Undefiniertes Verhalten
- Keine automatische Speicherverwaltung
- Kleiner Sprachkern und kleine Standardlibrary

C++ im Vergleich zu Java, C#

- Undefiniertes Verhalten
- Keine automatische Speicherverwaltung
- Kleiner Sprachkern und kleine Standardlibrary
- Templates

C++ im Vergleich zu Java, C#

- Undefiniertes Verhalten
- Keine automatische Speicherverwaltung
- Kleiner Sprachkern und kleine Standardlibrary
- Templates
- Operatorenüberladung

C++ im Vergleich zu Java, C#

- Undefiniertes Verhalten
- Keine automatische Speicherverwaltung
- Kleiner Sprachkern und kleine Standardlibrary
- Templates
- Operatorenüberladung
- Tendentiell weniger tiefe Vererbung

C++ im Vergleich zu Java, C#

- Undefiniertes Verhalten
- Keine automatische Speicherverwaltung
- Kleiner Sprachkern und kleine Standardlibrary
- Templates
- Operatorenüberladung
- Tendentiell weniger tiefe Vererbung
- Mehrfachvererbung

C++ im Vergleich zu Java, C#

- Undefiniertes Verhalten
- Keine automatische Speicherverwaltung
- Kleiner Sprachkern und kleine Standardlibrary
- Templates
- Operatorenüberladung
- Tendentiell weniger tiefe Vererbung
- Mehrfachvererbung
- Definierte Objektlebenszeit

Ein erstes C++ Programm

Beispiel: Hello World

Vom Sourcecode zur ausführbaren Datei

- Präprozessor

Vom Sourcecode zur ausführbaren Datei

- Präprozessor
- C++-Compiler

Vom Sourcecode zur ausführbaren Datei

- Präprozessor
- C++-Compiler
- Linker

Vom Sourcecode zur ausführbaren Datei

- Präprozessor
- C++-Compiler
- Linker
- includes sichern Typkonsitenz

Vom Sourcecode zur ausführbaren Datei

- Präprozessor
- C++-Compiler
- Linker
- includes sichern Typkonsitenz
- Templates müssen im Header definiert werden

Beispiel: Eine zweite Übersetzungseinheit

CMake

Warum CMake

- Nur geänderte Dateien neu kompilieren

Warum CMake

- Nur geänderte Dateien neu kompilieren
- Einzelner Befehl an Compiler wird zu kompliziert

Warum CMake

- Nur geänderte Dateien neu kompilieren
- Einzelner Befehl an Compiler wird zu kompliziert
- Portabilität

Beispiel: CMake

Mehr C++

Speicherverwaltung

- ```
  std::list<int> a = b;
  std::list<int> c = f(a);
  ```
  ```

Speicherverwaltung

- ```
  std::list<int> a = b;
  std::list<int> c = f(a);
  ```
- Jegliche Zuweisung ist eine Kopie, auch für Funktionsargumente
  ```

Speicherverwaltung

- ```
  std::list<int> a = b;
  std::list<int> c = f(a);
  ```
- Jegliche Zuweisung ist eine Kopie, auch für Funktionsargumente
- Einfach verständlich
  ```

Speicherverwaltung

- ```
  std::list<int> a = b;
  std::list<int> c = f(a);
  ```
- Jegliche Zuweisung ist eine Kopie, auch für Funktionsargumente
- Einfach verständlich
- Für große Objekte unnötige Performanceeinbuße
  ```

Pointer

- Pointer

Pointer

- Pointer
- Angst!

Pointer

- Pointer
- Angst!
- Gefährlich!

Pointer

- Pointer
- Angst!
- Gefährlich!
- Böse!

Pointer

- Pointer
- Angst!
- Gefährlich!
- Böse!
- Nicht schlimm aber viel Fehlerpotential

Pointer

- Pointer
- Angst!
- Gefährlich!
- Böse!
- Nicht schlimm aber viel Fehlerpotential
- `int b = 17; int *a = &b;`

Pointer

- Pointer
- Angst!
- Gefährlich!
- Böse!
- Nicht schlimm aber viel Fehlerpotential
- `int b = 17; int *a = &b;`
- `int *c = new int();`

Pointer

- Pointer
- Angst!
- Gefährlich!
- Böse!
- Nicht schlimm aber viel Fehlerpotential
- `int b = 17; int *a = &b;`
- `int *c = new int();`
- `delete c;`

Pointer

- Pointer
- Angst!
- Gefährlich!
- Böse!
- Nicht schlimm aber viel Fehlerpotential
- `int b = 17; int *a = &b;`
- `int *c = new int();`
- `delete c;`
- Gehört nicht in die Anwendungslogik

Smart-Pointer

- Funktionen aus der Standardlibrary

Smart-Pointer

- Funktionen aus der Standardlibrary
- unique_ptr

Smart-Pointer

- Funktionen aus der Standardlibrary
- unique_ptr
- Genau ein Owner

Smart-Pointer

- Funktionen aus der Standardlibrary
- unique_ptr
- Genau ein Owner
- `std::unique_ptr<int> a = std::make_unique<int>(17);`

Smart-Pointer

- Funktionen aus der Standardlibrary
- unique_ptr
- Genau ein Owner
- `std::unique_ptr<int> a = std::make_unique<int>(17);`
- shared_ptr

Smart-Pointer

- Funktionen aus der Standardlibrary
- unique_ptr
- Genau ein Owner
- `std::unique_ptr<int> a = std::make_unique<int>(17);`
- shared_ptr
- Quasi immer nutzbar
- `std::shared_ptr<int> a = std::make_shared<int>(17);`

Referenzen

- Sprachfeature kein Library-Feature

Referenzen

- Sprachfeature kein Library-Feature
- Können nicht null sein

Referenzen

- Sprachfeature kein Library-Feature
- Können nicht null sein
- Können aber ungültig werden

Referenzen

- Sprachfeature kein Library-Feature
- Können nicht null sein
- Können aber ungültig werden
- `int b = 17; int &a = b;`

Zusammenfassung Pointer

- Raw-Pointer: Wird quasi nie verwendet

Zusammenfassung Pointer

- Raw-Pointer: Wird quasi nie verwendet
- Unique-Pointer: Ersatz für Raw-Pointer

Zusammenfassung Pointer

- Raw-Pointer: Wird quasi nie verwendet
- Unique-Pointer: Ersatz für Raw-Pointer
- Shared-Pointer: Sichere Pointer für beliebig viele Ownerr

Zusammenfassung Pointer

- Raw-Pointer: Wird quasi nie verwendet
- Unique-Pointer: Ersatz für Raw-Pointer
- Shared-Pointer: Sichere Pointer für beliebig viele Ownerr
- Referenzen: Oftmals um Kopien zu vermeiden

Beispiel: Pointer & Referenzen

Design Pattern

Const-Correctness

- Alles per Referenz: Super Effizient aber Fehlerquelle

Const-Correctness

- Alles per Referenz: Super Effizient aber Fehlerquelle
- Const-Referenzen

Const-Correctness

- Alles per Referenz: Super Effizient aber Fehlerquelle
- Const-Referenzen
- `const int &a = b;`

Const-Correctness

- Alles per Referenz: Super Effizient aber Fehlerquelle
- Const-Referenzen
- `const int &a = b;`
- Const-Memberfunktionen

Const-Correctness

- Alles per Referenz: Super Effizient aber Fehlerquelle
- Const-Referenzen
- `const int &a = b;`
- Const-Memberfunktionen
- `int getX() const {...`

Const-Correctness

- Alles per Referenz: Super Effizient aber Fehlerquelle
- Const-Referenzen
- `const int &a = b;`
- Const-Memberfunktionen
- `int getX() const {...`
- Mutable

Beispiel: Const-Correctness

RAII

- Resource acquisition is initialization

RAII

- Resource acquisition is initialization
- Objekt akquiriert Resourcen im Konstruktor und gibt sie im Destruktor frei

RAII

- Resource acquisition is initialization
- Objekt akquiriert Resourcen im Konstruktor und gibt sie im Destruktor frei
- ```
  int readJsonFromFile(const std::string &fileName) {
      File f{fileName}
      Json j = f.parse();
      return j.at("ABC");
  }
  ```
  ```

OOP in C++

Klassendeklaration

```
class A : public B {
    public:
        A(int c, int d);
        auto getC() const -> int;
    private:
        const int d;
};
```

Klassendefinition

```
A::A(int c, int d) : B{c}, d{d} {
    // More code
}

auto A::getC() const -> int {
    return this->c;
}
```

Beispiel: HelloWorld OOP

Noch mehr C++

Casts und Null-Pointer

- `static_cast<T>(a)`

Casts und Null-Pointer

- `static_cast<T>(a)`
- `dynamic_cast<T>(a)`

Casts und Null-Pointer

- `static_cast<T>(a)`
- `dynamic_cast<T>(a)`
- `0`, `NULL` und `nullptr`

Type-Deduction

```
float f = 0;
auto i = 0;
auto i2 = i;
auto i3 = static_cast<int>(f);
decltype(i3) i4 = 12;
```

Lambda-Ausdrücke und Funktionen

```
int c = 17;
auto mulFun = [&c](int i, int m) {
    c++;
    return i * m;
}
mulFun(17, 3);

auto doubleFun =
    std::bind(mulFun, std::placeholder::_1, 2);
doubleFun(17);
```

Kurzeinführung Templates als Generics

```
template<typename T>
auto max(T i, T j) -> T {
    if (i > j) {
        return i;
    } else {
        return j;
    }
}

max<int>(1,2);
max(1,2);
```

Praxis: Einrichten der Entwicklungsumgebung

Linux (Ubuntu basiert)

```
sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test
sudo apt install -y gcc-8 g++-8 make cmake build-essential
update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-8 800 --s
```

```
sudo snap install clion --classic
```

macOs

```
xcode-select --install
```

CLion runterladen und installieren:
`https://www.jetbrains.com/clion/download/#section=mac`

Siehe Anleitung:
`https://www.jetbrains.com/help/clion/quick-tutorial-on-configuring-clion-on-macos.html`

Windows

Anleitung befolgen und hoffen:
`https://www.jetbrains.com/help/clion/quick-tutorial-on-configuring-clion-on-windows.html`

STL

STL

- Standard Template Library

STL

- Standard Template Library
- Utility

STL

- Standard Template Library
- Utility
- Container

STL

- Standard Template Library
- Utility
- Container
- Algorithmen

STL

- Standard Template Library
- Utility
- Container
- Algorithmen
- IO

STL

- Standard Template Library
- Utility
- Container
- Algorithmen
- IO
- Concurrency

Utility

- `std::string`

Utility

- `std::string`
- `std::unique_ptr<T>`

Utility

- `std::string`
- `std::unique_ptr<T>`
- `std::shared_ptr<T>`

Utility

- `std::string`
- `std::unique_ptr<T>`
- `std::shared_ptr<T>`
- `std::chrono`

Utility

- `std::string`
- `std::unique_ptr<T>`
- `std::shared_ptr<T>`
- `std::chrono`
- `std::sin()` ...

Utility

- `std::string`
- `std::unique_ptr<T>`
- `std::shared_ptr<T>`
- `std::chrono`
- `std::sin()` ...
- `std::complex` ...

Utility

- `std::string`
- `std::unique_ptr<T>`
- `std::shared_ptr<T>`
- `std::chrono`
- `std::sin()` ...
- `std::complex` ...
- `std::normal_distribution` ...

Utility

- `std::string`
- `std::unique_ptr<T>`
- `std::shared_ptr<T>`
- `std::chrono`
- `std::sin()` ...
- `std::complex` ...
- `std::normal_distribution` ...
- `std::function<T(A...)>`

Container

- `std::array<T, N>`

Container

- `std::array<T, N>`
- `std::vector<T>`

Container

- `std::array<T, N>`
- `std::vector<T>`
- `std::deque<T>`

Container

- `std::array<T, N>`
- `std::vector<T>`
- `std::deque<T>`
- `std::list<T>` und `std::forward_list<T>`

Container

- `std::array<T, N>`
- `std::vector<T>`
- `std::deque<T>`
- `std::list<T>` und `std::forward_list<T>`
- `std::set<T>` und `std::map<K, V>`

Container

- `std::array<T, N>`
- `std::vector<T>`
- `std::deque<T>`
- `std::list<T>` und `std::forward_list<T>`
- `std::set<T>` und `std::map<K, V>`
- `std::tuple<T...>` und `std::pair<T1, T2>`

Container

- `std::array<T, N>`
- `std::vector<T>`
- `std::deque<T>`
- `std::list<T>` und `std::forward_list<T>`
- `std::set<T>` und `std::map<K, V>`
- `std::tuple<T...>` und `std::pair<T1, T2>`
- `std::optional<T>`

Iteratoren

```
std::vector<int> a = {1,2,17,42,1337};
int b = 0;

for (std::vector<int>::iterator it = a.begin();
        it != a.end(); it++) {
    b += *it;
}
```

for-each

```
std::vector<int> a = {1,2,17,42,1337};
int b = 0;

for (const auto &i : a) {
    b += i;
}
```

Algorithmen

- `std::transform`

Algorithmen

- `std::transform`
- `std::sort`

Algorithmen

- `std::transform`
- `std::sort`
- `std::max`, `std::max_element` ...

IO

- `std::istream` und `std::ostream`

IO

- `std::istream` und `std::ostream`
- `std::cout`, `std::cerr` und `std::cin`

IO

- `std::istream` und `std::ostream`
- `std::cout`, `std::cerr` und `std::cin`
- `std::fstream`

Concurrency

- `std::thread`

Concurrency

- `std::thread`
- `std::launch` und `std::future<T>`

Concurrency

- `std::thread`
- `std::launch` und `std::future<T>`
- `std::mutex` und `std::lock_guard<T>`

Concurrency

- `std::thread`
- `std::launch` und `std::future<T>`
- `std::mutex` und `std::lock_guard<T>`
- `std::atomic<T>`

Praxis

Praxis:

Praxis: Huffman-Codierer

Vorgehen

- Datei einlesen

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren
  - Menge aller Symbole mit zugehörigen Wahrscheinlichkeiten

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren
  - Menge aller Symbole mit zugehörigen Wahrscheinlichkeiten
  - Zwei Symbole geringster Wahrscheinlichkeit finden

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren
  - Menge aller Symbole mit zugehörigen Wahrscheinlichkeiten
  - Zwei Symbole geringster Wahrscheinlichkeit finden
  - Symbole aus Menge Entfernen

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren
  - Menge aller Symbole mit zugehörigen Wahrscheinlichkeiten
  - Zwei Symbole geringster Wahrscheinlichkeit finden
  - Symbole aus Menge Entfernen
  - Zu neuem Knoten kombinieren

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren
  - Menge aller Symbole mit zugehörigen Wahrscheinlichkeiten
  - Zwei Symbole geringster Wahrscheinlichkeit finden
  - Symbole aus Menge Entfernen
  - Zu neuem Knoten kombinieren
  - Knoten zu Menge hinzufügen

Vorgehen

- Datei einlesen
- Relative Häufigkeiten ($\approx$ Wahrscheinlichkeiten) berechnen (Byteweise)
- Huffman-Baum konstruieren
  - Menge aller Symbole mit zugehörigen Wahrscheinlichkeiten
  - Zwei Symbole geringster Wahrscheinlichkeit finden
  - Symbole aus Menge Entfernen
  - Zu neuem Knoten kombinieren
  - Knoten zu Menge hinzufügen
- Abbildung ausgeben

Tools

GTest

```
TEST(SqrtTest, Simple) {
  EXPECT_EQ(sqrt(4), 2);
}

TEST(SqrtTest, Negative) {
  EXPECT_THROW(sqrt(-1), std::runtime_error);
}
```

Doxygen

```
/**
 * Returns the maximum of two objects.
 *
 * @tparam T the type of the objects,
 *     the type must support the < operator.
 * @param i the first object
 * @param j the second object
 * @return the maximum value
 */
template<typename T>
auto max(T i, T j) -> T {
```

Debugging und Fehlersuche

- Debugger

Debugging und Fehlersuche

- Debugger
- Valgrind

Debugging und Fehlersuche

- Debugger
- Valgrind
- LibAdressSanitizer (Asan)

Debugging und Fehlersuche

- Debugger
- Valgrind
- LibAdressSanitizer (Asan)
- clang-tidy

Beispiel: Tools

Abschluss

Was fehlt?

- R-Value Referenzen, forward/universal Referenzen

Was fehlt?

- R-Value Referenzen, forward/universal Referenzen
- Move

Was fehlt?

- R-Value Referenzen, forward/universal Referenzen
- Move
- Destruktor und Copy / Move Konstruktor

Was fehlt?

- R-Value Referenzen, forward/universal Referenzen
- Move
- Destruktor und Copy / Move Konstruktor
- Operatorenüberladung

Was fehlt?

- R-Value Referenzen, forward/universal Referenzen
- Move
- Destruktor und Copy / Move Konstruktor
- Operatorenüberladung
- Friend Definition

Was fehlt?

- R-Value Referenzen, forward/universal Referenzen
- Move
- Destruktor und Copy / Move Konstruktor
- Operatorenüberladung
- Friend Definition
- Meta-Programming

Mehr Informationen

- en.cppreference.com

Mehr Informationen

- en.cppreference.com
- github.com/isocpp/CppCoreGuidelines

Mehr Informationen

- en.cppreference.com
- github.com/isocpp/CppCoreGuidelines
- godbolt.org

Mehr Informationen

- en.cppreference.com
- github.com/isocpp/CppCoreGuidelines
- godbolt.org
- github.com/SoPra-Team-10/CppCMakeIntro

Mehr Informationen

- en.cppreference.com
- github.com/isocpp/CppCoreGuidelines
- godbolt.org
- github.com/SoPra-Team-10/CppCMakeIntro
- Scott Meyers: Effective Modern C++