Eine Einführung in modernes C++ mit CMake

Paul Nykiel

September 29, 2019

Einleitung

Was ist C++

- ~~C with classes~~

Was ist C++

- ~~C with classes~~
- Standardisiert und offen

Was ist C++

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

Was ist C++

- ~~C with classes~~
- Standardisiert und offen
- Wird in quasi jeder Domäne genutzt
- Ziele: Performanter und sicherer Code

C++ im Vergleich zu Java

- Undefiniertes Verhalten

C++ im Vergleich zu Java

- Undefiniertes Verhalten
- Keine automatische Speicherverwaltung

C++ im Vergleich zu Java

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

C++ im Vergleich zu Java

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

C++ im Vergleich zu Java

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

C++ im Vergleich zu Java

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

C++ im Vergleich zu Java

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

C++ im Vergleich zu Java

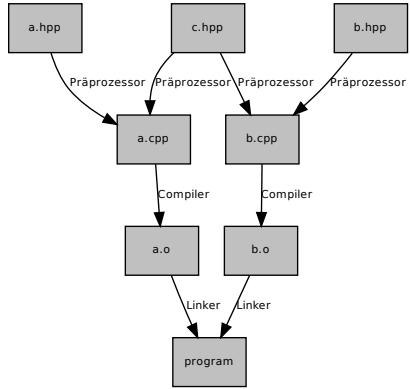
- Undefiniertes Verhalten
- Keine automatische Speicherverwaltung
- Kleiner Sprachkern und kleine Standardlibrary
- Templates
- Operatorenüberladung
- Tendentiell weniger tiefe Vererbung
- Mehrfachvererbung
- Definierte Objektlebenszeit

Ein erstes C++ Programm

Beispiel: Hello World

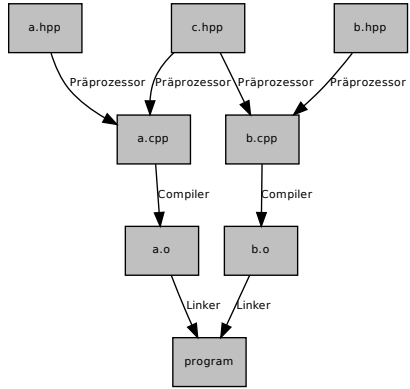
Vom Sourcecode zur ausführbaren Datei

- Präprozessor



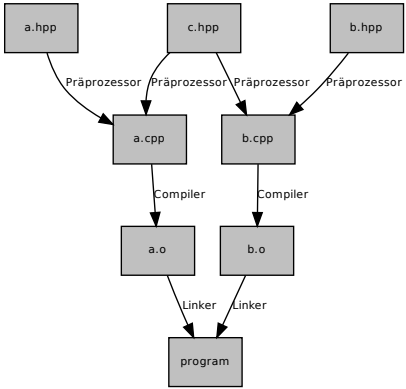
Vom Sourcecode zur ausführbaren Datei

- Präprozessor
- Compiler



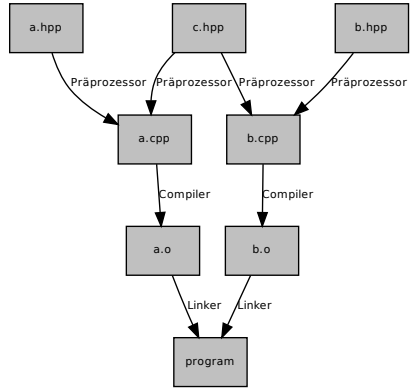
Vom Sourcecode zur ausführbaren Datei

- Präprozessor
- Compiler
- Linker



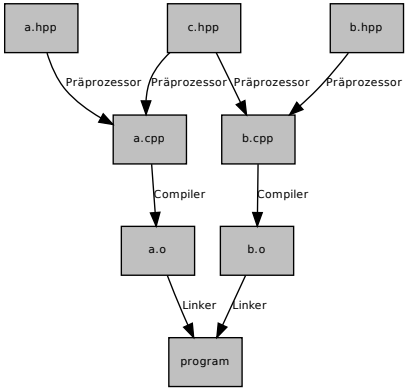
Vom Sourcecode zur ausführbaren Datei

- Präprozessor
- Compiler
- Linker
- `#include`s sichern Typkonsistenz



Vom Sourcecode zur ausführbaren Datei

- Präprozessor
- Compiler
- Linker
- `#include`s sichern Typkonsistenz
- Templates müssen im Header definiert werden



Beispiel: Eine zweite Übersetzungseinheit

CMake

Warum ein Buildsystem

- Nur geänderte Dateien neu kompilieren

Warum ein Buildsystem

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

Warum ein Buildsystem

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

CMake

Wird durch Datei `CMakeLists.txt` konfiguriert:

```
cmake_minimum_required(VERSION 3.10)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_FLAGS "-Wall")
if (CMAKE_BUILD_TYPE STREQUAL "Release")
    set(CMAKE_CXX_FLAGS  "${CMAKE_CXX_FLAGS} -O3")
endif ()

project(MyProject)
add_executable(${PROJECT_NAME} main.cpp mylib.cpp)
target_link_libraries(${PROJECT_NAME} pthread)
add_subdirectory(Tests)
```

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

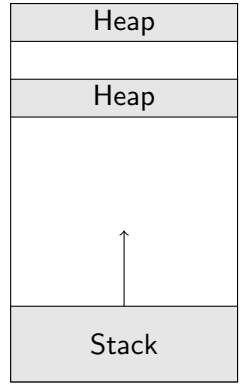
- Pointer
- Angst!
- Gefährlich!
- Böse!
- Nicht schlimm aber viel Fehlerpotential
- `int b = 17; int *a = &b;`
- `int *c = new int();`
- `delete c;`

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

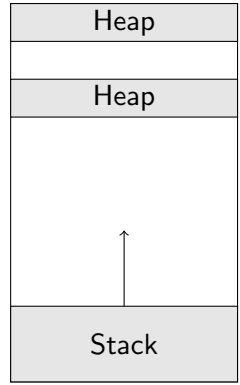
Stack und Heap

- Hauptspeicher (RAM) wird aus zwei Richtungen vergeben



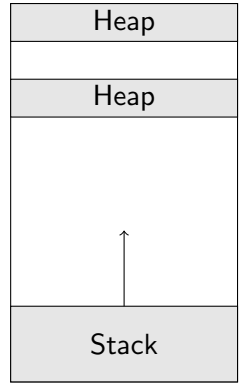
Stack und Heap

- Hauptspeicher (RAM) wird aus zwei Richtungen vergeben
- Stack wird für Funktion aufgebaut



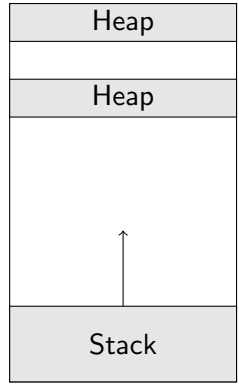
Stack und Heap

- Hauptspeicher (RAM) wird aus zwei Richtungen vergeben
- Stack wird für Funktion aufgebaut
- Heap für dynamischen Speicher



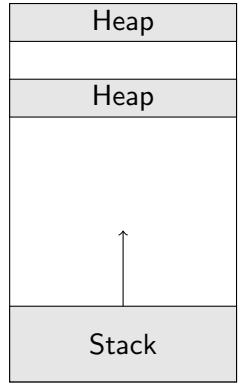
Stack und Heap

- Hauptspeicher (RAM) wird aus zwei Richtungen vergeben
- Stack wird für Funktion aufgebaut
- Heap für dynamischen Speicher
- Speicher auf dem Heap muss händisch reserviert und freigegeben werden



Stack und Heap

- Hauptspeicher (RAM) wird aus zwei Richtungen vergeben
- Stack wird für Funktion aufgebaut
- Heap für dynamischen Speicher
- Speicher auf dem Heap muss händisch reserviert und freigegeben werden
- Bei mehr als einem Owner Verwaltung kompliziert



Beispiel Pointer

```
int main() {
    int a = 0; // Liegt auf dem Stack
    int *aPtr = &a; // Zeigt auf den Stack

    int *bPtr = new int(); // Liegt auf dem Heap
    delete bPtr; // Speicher muss freigegeben werden
}
```

Smart-Pointer

- Standardlibrary kann Verwaltung übernehmen

Smart-Pointer

- Standardlibrary kann Verwaltung übernehmen
- `unique_ptr`

Smart-Pointer

- Standardlibrary kann Verwaltung übernehmen
- `unique_ptr`
- Genau ein Owner

Smart-Pointer

- Standardlibrary kann Verwaltung übernehmen
- `unique_ptr`
- Genau ein Owner
- `std::unique_ptr<int> a = std::make_unique<int>(17);`

Smart-Pointer

- Standardlibrary kann Verwaltung übernehmen
- `unique_ptr`
- Genau ein Owner
- `std::unique_ptr<int> a = std::make_unique<int>(17);`
- `shared_ptr`

Smart-Pointer

- Standardlibrary kann Verwaltung übernehmen
- `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: Sollten quasi nie verwendet werden

Zusammenfassung Pointer

- Raw-Pointer: Sollten quasi nie verwendet werden
- Unique-Pointer: Oftmals ersatz für Raw-Pointer

Zusammenfassung Pointer

- Raw-Pointer: Sollten quasi nie verwendet werden
- Unique-Pointer: Oftmals ersatz für Raw-Pointer
- Shared-Pointer: Sichere Pointer für beliebig viele Owner

Zusammenfassung Pointer

- Raw-Pointer: Sollten quasi nie verwendet werden
- Unique-Pointer: Oftmals ersatz für Raw-Pointer
- Shared-Pointer: Sichere Pointer für beliebig viele Owner
- 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
- ```
  void doStuff() {
      std::lock_guard<std::mutex> lockGuard{mutex};
      shared_resource = 17;
      shared_resource += functionThatCanThrow();
  }
  ```
  ```

OOP in C++

Klassendeklaration

```
class A : public B {
    public:
        A(int c, int d);
        int getD() const;
    private:
        int d;
};
```

Klassendefinition

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

int A::getD() const {
    return this->d;
}
```

Namespaces

```
namespace mynamespace {
    int calculate() { return 17; }
}

int main() {
    return mynamespace::calculate();
}
```

Namespaces

```
namespace mynamespace {
    int calculate() { return 17; }
}

int main() {
    return mynamespace::calculate();
}
```

- Struktur

Namespaces

```
namespace mynamespace {
    int calculate() { return 17; }
}

int main() {
    return mynamespace::calculate();
}
```

- Struktur
- Keinen Einfluss auf Sichtbarkeit

Namespaces

```
namespace mynamespace {
    int calculate() { return 17; }
}

int main() {
    return mynamespace::calculate();
}
```

- Struktur
- Keinen Einfluss auf Sichtbarkeit
- Namespaces nicht mit `using` einbinden

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`

Casts und Null-Pointer

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

```
// Normale Syntax
std::vector<std::set<double>> a(double b) {

// Trailing return-type
auto a(double b) -> std::vector<std::set<double>> {
```

Type-Deduction

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

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);
```

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

Container

	Auf Element zugreifen	Element einfügen
`std::array<T, N>`	$\mathcal{O}(1)$	X
`std::vector<T>`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`std::deque<T>`	$\mathcal{O}(1)$	$\mathcal{O}(n)$ bzw. $\mathcal{O}(1)$
`std::list<T>`	$\mathcal{O}(n)$	$\mathcal{O}(1)$

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;
}
```

Weitere Container und Aggregationstypen

- Assoziative-Container: `std::set<T>` und `std::map<K, V>`

Weitere Container und Aggregationstypen

- Assoziative-Container: `std::set<T>` und `std::map<K, V>`
- Sammlung verschiedener Objekte: `std::tuple<T...>` und `std::pair<T1, T2>`

Weitere Container und Aggregationstypen

- Assoziative-Container: `std::set<T>` und `std::map<K, V>`
- Sammlung verschiedener Objekte: `std::tuple<T...>` und `std::pair<T1, T2>`
- Objekt das nicht vorhanden sein muss: `std::optional<T>`

Tools

GTest

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

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

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

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`
- `git.spatz.wtf/spatzenhirn/cppcmakeintro`

Mehr Informationen

- `en.cppreference.com`
- `github.com/isocpp/CppCoreGuidelines`
- `godbolt.org`
- `git.spatz.wtf/spatzenhirn/cppcmakeintro`
- Scott Meyers: Effective Modern C++

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