

Eine Einführung in modernes C++

Teil 1 – Basics

Paul Nykiel

2. Mai 2020

- 1 C
- 2 Ein erstes C Programm
- 3 Buildprozess
- 4 Speicher
- 5 Pointer
- 6 C++
- 7 Ein erstes C++ Programm
- 8 Zero-Copy

C

Ein erstes C
Programm

Buildprozess

Speicher

Pointer

C++

Ein erstes
C++
Programm

Zero-Copy

C

Was ist C

- ANSI/ISO Standardisierte Programmiersprache

Was ist C

- ANSI/ISO Standardisierte Programmiersprache
- Entwickelt ab 1969 für Unix

Was ist C

- ANSI/ISO Standardisierte Programmiersprache
- Entwickelt ab 1969 für Unix
- „Portabler Assembler“

Was ist C

- ANSI/ISO Standardisierte Programmiersprache
- Entwickelt ab 1969 für Unix
- „Portabler Assembler“
- Einfach zu verstehen, schwierig zu verwenden

C

Ein erstes C
Programm

Buildprozess

Speicher

Pointer

C++

Ein erstes
C++
Programm

Zero-Copy

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

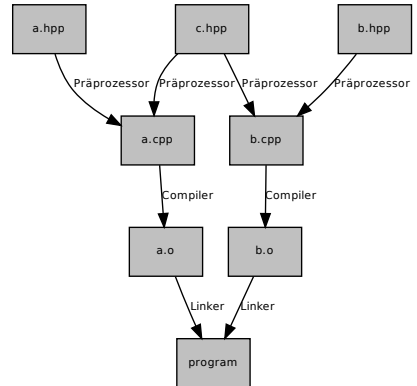
Ein erstes C Programm

Beispiel: Hello World

Buildprozess

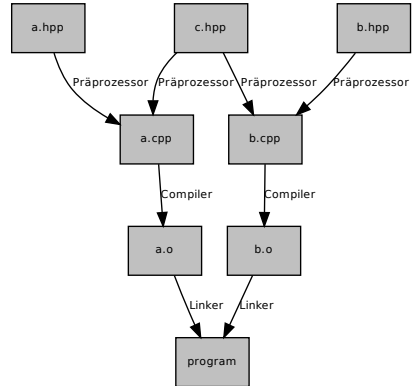
Vom Sourcecode zur ausführbaren Datei

- Präprozessor



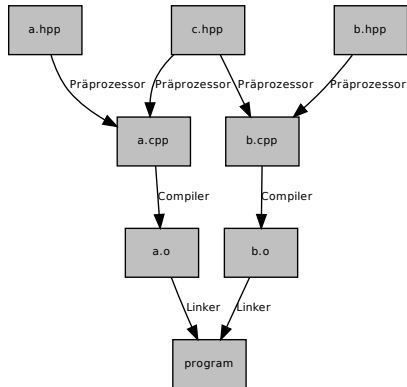
Vom Sourcecode zur ausführbaren Datei

- Präprozessor
- Compiler



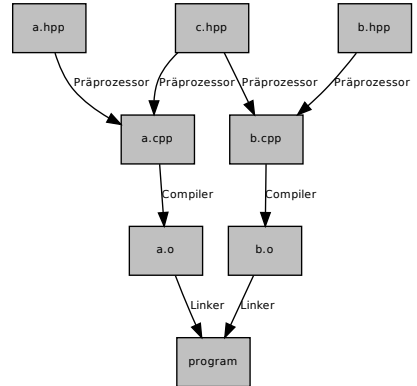
Vom Sourcecode zur ausführbaren Datei

- Präprozessor
- Compiler
- Linker



Vom Sourcecode zur ausführbaren Datei

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



Beispiel: Eine zweite Übersetzungseinheit

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

Wird durch Datei CMakeLists.txt konfiguriert:

```
cmake_minimum_required(VERSION 3.10)

set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_C_FLAGS "-Wall")
set(CMAKE_CXX_FLAGS "-Wall")

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

Beispiel: CMake

C

Ein erstes C
Programm

Buildprozess

Speicher

Pointer

C++

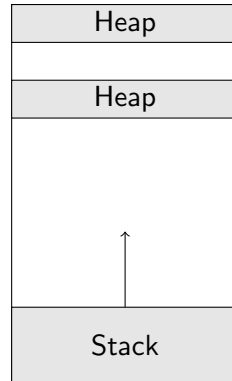
Ein erstes
C++
Programm

Zero-Copy

Speicher

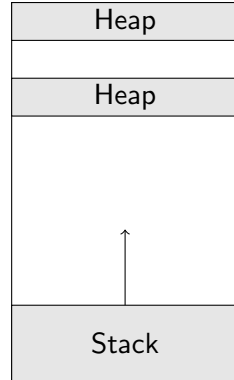
- Hauptspeicher (RAM) wird aus zwei Richtungen vergeben

Stack und Heap



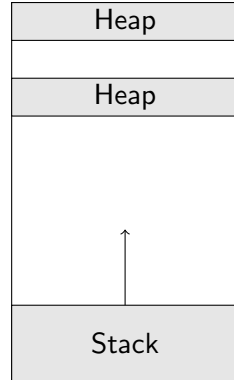
- Hauptspeicher (RAM) wird aus zwei Richtungen vergeben
- Stack wird für Funktion aufgebaut

Stack und Heap



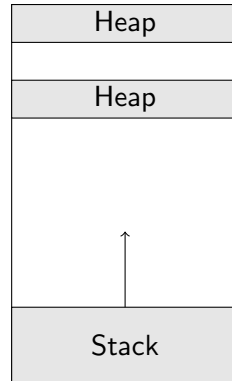
- Hauptspeicher (RAM) wird aus zwei Richtungen vergeben
- Stack wird für Funktion aufgebaut
- Heap für dynamischen Speicher

Stack und Heap



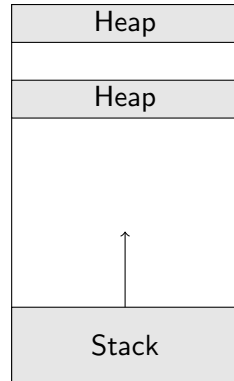
- Hauptspeicher (RAM) wird aus zwei Richtungen vergeben
- Stack wird für Funktion aufgebaut
- Heap für dynamischen Speicher
- Speicher auf dem Heap muss händisch reserviert und freigegeben werden

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

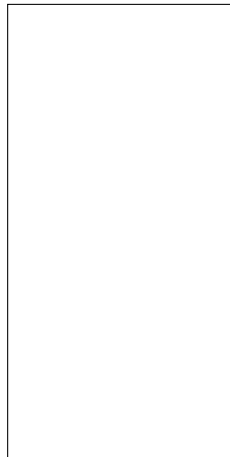
Stack und Heap



Stack

```
int sqr(int arg) {  
    int res = arg * arg;  
    return res;  
}
```

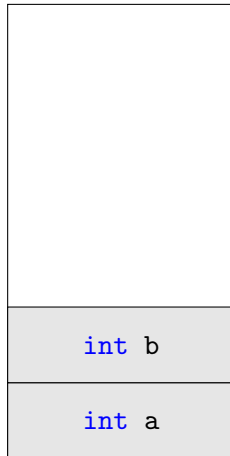
```
int main(void) {  
    int a = 17;  
    int b = sqr(a);  
    return 0;  
}
```



Stack

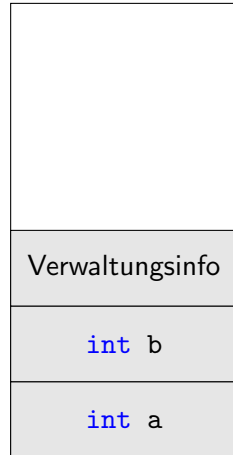
```
int sqr(int arg) {  
    int res = arg * arg;  
    return res;  
}
```

```
int main(void) {  
    int a = 17;  
    int b = sqr(a);  
    return 0;  
}
```



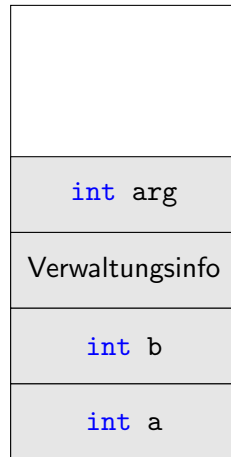
```
int sqr(int arg) {  
    int res = arg * arg;  
    return res;  
}  
  
int main(void) {  
    int a = 17;  
    int b = sqr(a);  
    return 0;  
}
```

Stack



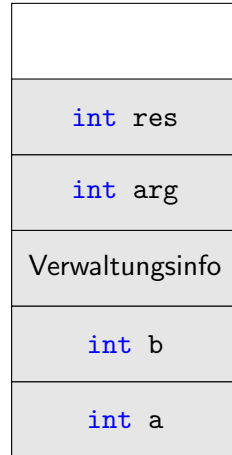

```
int sqr(int arg) {  
    int res = arg * arg;  
    return res;  
}  
  
int main(void) {  
    int a = 17;  
    int b = sqr(a);  
    return 0;  
}
```

Stack



```
int sqr(int arg) {  
    int res = arg * arg;  
    return res;  
}  
  
int main(void) {  
    int a = 17;  
    int b = sqr(a);  
    return 0;  
}
```

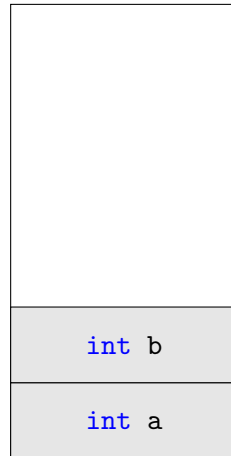
Stack



Stack

```
int sqr(int arg) {  
    int res = arg * arg;  
    return res;  
}
```

```
int main(void) {  
    int a = 17;  
    int b = sqr(a);  
    return 0;  
}
```



Reicht das nicht?

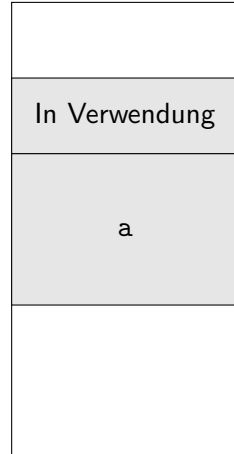
```
int main(void) {  
    int *a = new int[2];  
    int *b = new int[2];  
    delete[] a;  
    return 0;  
}
```

Heap



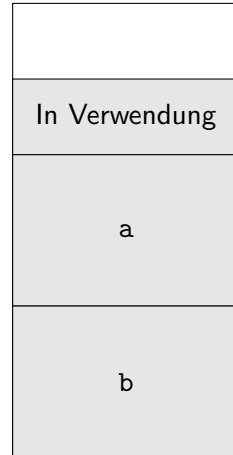
```
int main(void) {  
    int *a = new int[2];  
    int *b = new int[2];  
    delete[] a;  
    return 0;  
}
```

Heap



```
int main(void) {  
    int *a = new int[2];  
    int *b = new int[2];  
    delete[] a;  
    return 0;  
}
```

Heap



```
int main(void) {  
    int *a = new int[2];  
    int *b = new int[2];  
    delete[] a;  
    return 0;  
}
```

Heap



C

Ein erstes C
Programm

Buildprozess

Speicher

Pointer

C++

Ein erstes
C++
Programm

Zero-Copy

Pointer

Pointer

- Pointer

- Pointer
- Angst!

Pointer

Pointer

- Pointer
- Angst!
- Gefährlich!

Pointer

- Pointer
- Angst!
- Gefährlich!
- Böse!

Pointer

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

- Pointer
- Angst!
- Gefährlich!
- Böse!
- Nicht schlimm aber viel Fehlerpotential
- Was ist ein Pointer: eigentlich nur ein `int` den der Compiler separat behandelt.

C

Ein erstes C
Programm

Buildprozess

Speicher

Pointer

C++

Ein erstes
C++
Programm

Zero-Copy

Pointer

- Pointer
- Angst!
- Gefährlich!
- Böse!
- Nicht schlimm aber viel Fehlerpotential
- Was ist ein Pointer: eigentlich nur ein `int` den der Compiler separat behandelt.
- `int b = 17; int *a = &b;`

Pointer

- Pointer
- Angst!
- Gefährlich!
- Böse!
- Nicht schlimm aber viel Fehlerpotential
- Was ist ein Pointer: eigentlich nur ein `int` den der Compiler separat behandelt.
- `int b = 17; int *a = &b;`
- `int *c = new int();`

- Pointer
- Angst!
- Gefährlich!
- Böse!
- Nicht schlimm aber viel Fehlerpotential
- Was ist ein Pointer: eigentlich nur ein `int` den der Compiler separat behandelt.
- `int b = 17; int *a = &b;`
- `int *c = new int();`
- `delete c;`

- Pointer
- Angst!
- Gefährlich!
- Böse!
- Nicht schlimm aber viel Fehlerpotential
- Was ist ein Pointer: eigentlich nur ein `int` den der Compiler separat behandelt.
- `int b = 17; int *a = &b;`
- `int *c = new int();`
- `delete c;`
- Speicherverwaltung gehört nicht in die Anwendungslogik

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  
}
```

C

Ein erstes C
Programm

Buildprozess

Speicher

Pointer

C++

Ein erstes
C++
Programm

Zero-Copy

C++

Was ist C++

- ~~C with classes~~

Was ist C++

- ~~C with classes~~
- Wird in quasi jeder Domäne genutzt

Was ist C++

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

C++ im Vergleich zu Java

- Templates

C++ im Vergleich zu Java

- Templates
- Operatorenüberladung

C++ im Vergleich zu Java

- Templates
- Operatorenüberladung
- Tendentiell weniger tiefe Vererbung

C++ im Vergleich zu Java

- Templates
- Operatorenüberladung
- Tendentiell weniger tiefe Vererbung
- Mehrfachvererbung

C++ im Vergleich zu Java

- Templates
- Operatorenüberladung
- Tendentiell weniger tiefe Vererbung
- Mehrfachvererbung
- Definierte Objektlebenszeit

Ein erstes C++ Programm

Beispiel: Hello World

C

Ein erstes C
Programm

Buildprozess

Speicher

Pointer

C++

Ein erstes
C++
Programm

Zero-Copy

Zero-Copy

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

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);`

- Sprachfeature kein Library-Feature

C

Ein erstes C
Programm

Buildprozess

Speicher

Pointer

C++

Ein erstes
C++
Programm

Zero-Copy

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