Einführung in C++



tum.3D

computer graphics & visualization

- Die Standard-Sprache für die Spieleentwicklung
  - C++ ermöglicht es, sehr effizienten Code zu schreiben
  - Bietet low-level Zugriff auf die Hardware
  - Genaue Verwaltung / Optimierung des Speichers möglich
  - Hochoptimierte Compiler
  - Sehr breite Hardware-Unterstützung: C++ läuft überall
    - Garantiert überall wo Java läuft, da die Java VM in C++ implementiert ist …

- Mächtige Sprache
  - Fokus auf Effizienz
    - Für „Sicherheit“ muss man extra Aufwand treiben
    - Beispiel:
      - Array-Grenzen werden standardmäßig *nicht* überprüft
      - Keine automatische Garbage-Collection
  - Generische Programmierung (*Templates*)
- Wir beschränken uns hier auf die Teile, die es so ähnlich auch in Java gibt

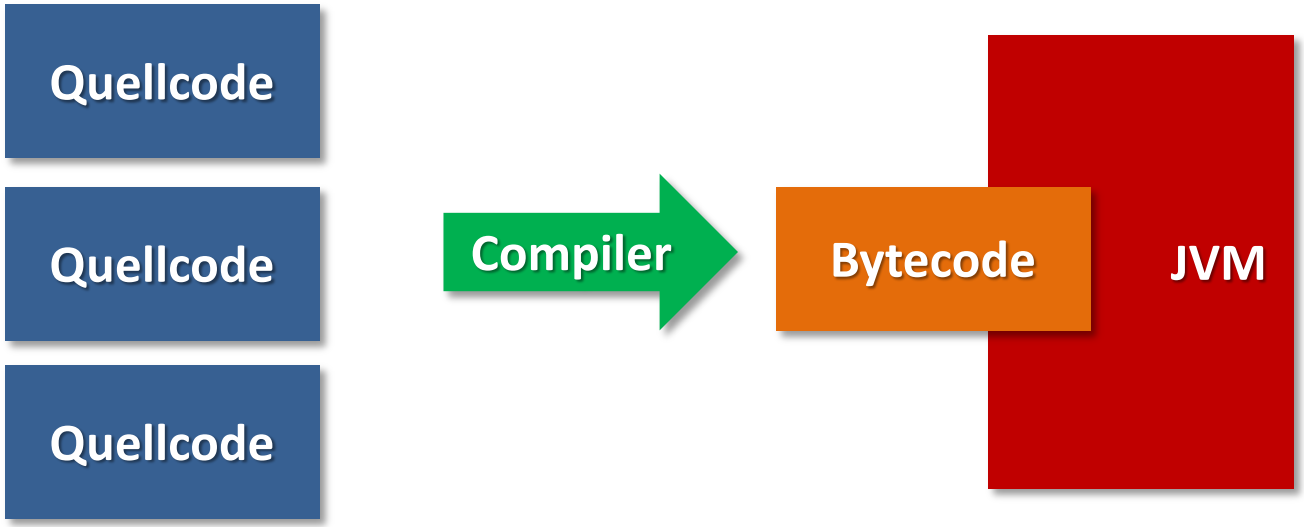
C++

- Grundsätzliche Unterschiede zu Java:
  - Kompilierungsmodell
  - Speicherverwaltung, Destruktoren
  - Klassen/Methoden-Aufrufe
- Wichtige C++ Features, die wir nicht ansprechen
  - Template-Programmierung
  - Mehrfachvererbung / Interfaces
- Wir nehmen im folgenden C++11 an
  - Zum selber ausprobieren Visual Studio 2010+, GCC 4.6+ mit --std=c++0x (bzw. c++11x) oder Clang 3.0 verwenden

C++: Kompilierung

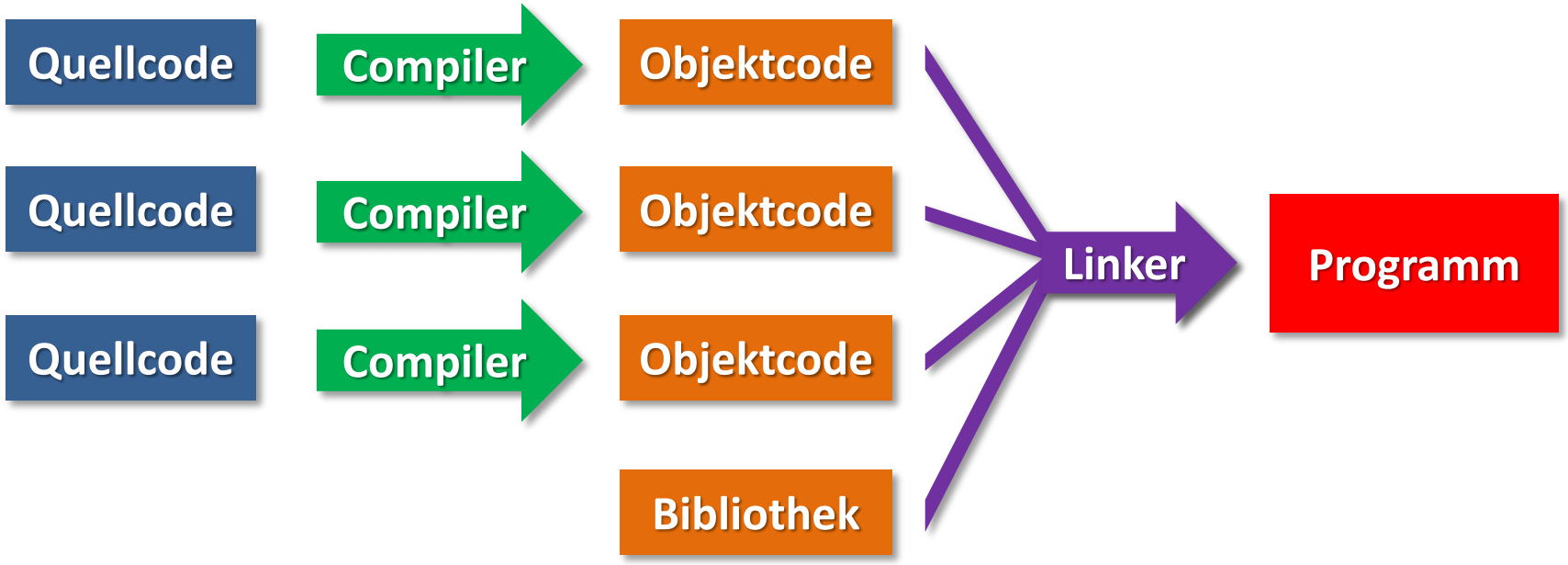
- Java
  - Alles wird in `.class`-Files kompiliert
  - Compiler findet Informationen über andere Klassen „automatisch“
  - `.class`-Files sind direkt ausführbar
- C++
  - Klassen und Methoden werden in Objekt-Dateien kompiliert
  - Informationen über andere Klassen müssen explizit verfügbar sein
  - Objekt-Dateien werden durch den Linker zu einem Programm zusammengebaut

Java



C++

- Der Compiler übersetzt den Quellcode in Objektcode (*.obj)
- Der Linker verknüpft Objektcode und Bibliotheken (*.lib)



Kompilieren von C++

- Der Linker sieht alle generierten Objekt-Dateien
  - Erstellt separate Listen der dort definierten Funktionen
  - Hängt alle Objekt-Dateien aneinander
  - `unresolved external`: Funktion wurde nicht gefunden
- Bibliotheken: „Pakete“ von kompilieren Objekt-Dateien
  - Z.B. die Standardbibliothek
- Am Ende kommt eine ausführbare Datei heraus

Kompilieren in C++

- Jede Methode und Klasse muss vor der ersten Verwendung **deklariert** sein
  - Macht die Parametertypen und Rückgabewerte dem Compiler bekannt

```
// Deklaration
void Foo (int a);

// Aufruf jetzt möglich

void Bar () {
    Foo (23);
}

void Foo (int a) {
    // Definition, eventuell in anderer Datei als Bar()
}
```

Kompilieren in C++

- Trennung von **Deklaration** und **Definition**
  - Typischerweise in seperaten Dateien
  - Deklaration: Header (*.h oder *.hpp)
  - Definition: *.cpp

- Nur .cpp Dateien werden kompiliert!
  - Header werden in .cpp Dateien eingebunden
  - Deklarationen von evtl. externen Funktionen (separate .cpp Datei oder Library) sind dem Compiler ab diesem Zeitpunkt bekannt

Kompilieren in C++

- Präprozessor
  - Textersetzung, verarbeitet jede Datei vor dem Kompilieren
  - Header einbinden: `#include "foo.h"`
    - Fügt den Inhalt von „foo.h“ an dieser Stelle ein
  - Mehrfache Header-Einbindung verhindern: `#pragma once`
    - C++-Standard: Include Guard über `#define` und `#ifndef`
  - Makros definieren:
    - `#define PI 3.0`
    - `#define SUM(x,y) ((x)+(y))`

C++: Das erste Programm

```cpp
#include <iostream>

int main (int argc, char* argv []) {
    // std::cout == (Java) System.out
    std::cout << "Hello World\n";
    return 0;
}
```

- Anders als bei Java
  - Keine Klasse notwendig
  - Hier trifft C auf C++: Keine Strings, sondern `char*` -- dazu später mehr
  - `std::cout` ist ein Objekt aus der Standardbibliothek. Diese wird normalerweise automatisch vom Linker gebunden.

C++: Das erste Programm

```cpp
#include <iostream>

int main (int argc, char* argv []) {
    // std::cout == (Java) System.out
    std::cout << "Hello World\n";
    return 0;
}
```

- Präprozessor-Anweisung
  - „Füge den Inhalt von `iostream.h` hier ein.“
    - Spitze Klammern: Durchsuche vordefinierte Include-Pfade
    - Anführungszeichen: Durchsuche lokale Dateien

C++: Das erste Programm

```cpp
#include <iostream>

int main (int argc, char* argv []) {
    // std::cout == (Java) System.out
    std::cout << "Hello World\n";
    return 0;
}
```

- Main Funktion: Startpunkt des Programms

C++: Das erste Programm

```cpp
#include <iostream>

int main (int argc, char* argv []) {
    // std::cout == (Java) System.out
    std::cout << "Hello World\n";
    return 0;
}
```

- Namespace `std` (Standardbibliothek)
  - Namespaces ermöglichen bessere Organisation
  - Verhindern Mehrdeutigkeiten

C++: Das erste Programm

```
#include <iostream>

int main (int argc, char* argv []) {
    // std::cout == (Java) System.out
    std::cout << "Hello World\n";
    return 0;
}
```

- Operator
  - *Insertion* operator (speziell für Streams)
  - Ähnlich wie +, -, *, /, &, =, == ….

C++: Das erste Programm

```cpp
#include <iostream>

int main (int argc, char* argv []) {
    // std::cout == (Java) System.out
    std::cout << "Hello World\n";
    return 0;
}
```

- String literal
  - Vom Compiler als `const char*` (C-String) behandelt

C++: Das erste Programm

- Vergisst man das `#include <iostream>`, dann kennt der **Compiler** `std::cout` nicht

```
error C2039: 'cout' : is not a member of 'std‘
error C2065: 'cout' : undeclared identifier
```

- Die Standardbibliothek wird automatisch vom Linker benutzt. Verhindert man das, findet der **Linker** `std::cout` nicht

```
error LNK2001: unresolved external symbol "__declspec(dllimport) class
std::basic_ostream<char,struct std::char_traits<char> > std::cout"
(__imp_?cout@std@@3V?$basic_ostream@DU?$char_traits@D@std@@@1@A)
```

C++: Speicherverwaltung

- C++ verwaltet Speicher explizit
  - Anfordern mit `new`
  - Freigeben mit `delete`
  - Jedes `new` benötigt ein `delete`!

- Speicheradressen werden in Zeigern (Pointer) gespeichert
  - `int* pData = new int;`
  - `pData` ist ein Zeiger auf ein Element vom Typ `int`
    - Compiler kennt Typ und Größe der Datenelemente

C++: Speicherverwaltung

- Die Lebensdauer der Daten ist nicht an die Lebensdauer von `pData` gebunden
  - Wenn `pData` „out-of-scope“ geht, ist der Speicher noch belegt („Memory-Leak“)
  - Freigeben mit `delete pData`

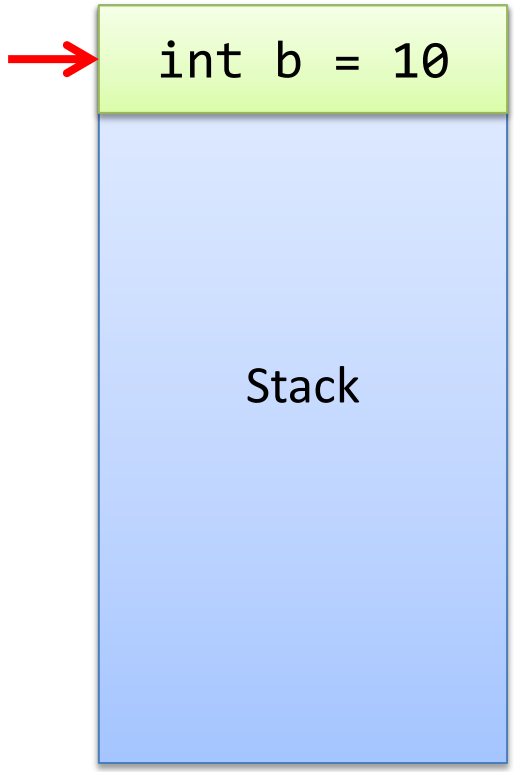
- Anlegen mehrerer Elemente: `new[]`
  - `int pDataArray = new int[255];`
  - pDataArray zeigt auf das erste von 255 Elementen vom Typ int
  - Freigabe: Bei `new[]` mit `delete[]`
    - `Delete[] pDataArray;`

C++: Speichverwaltung

- **Stack**
  - **Stack** wird u.a. für lokal definierte Variablen verwendet
  - Bei Verlassen der scope {} wird der Stack zurückgesetzt

```
int main(int argc, const char* argv[])
{
    int b = 10;

    while (true)
    {
        int a = b + 5;
        break;
    }
}
```

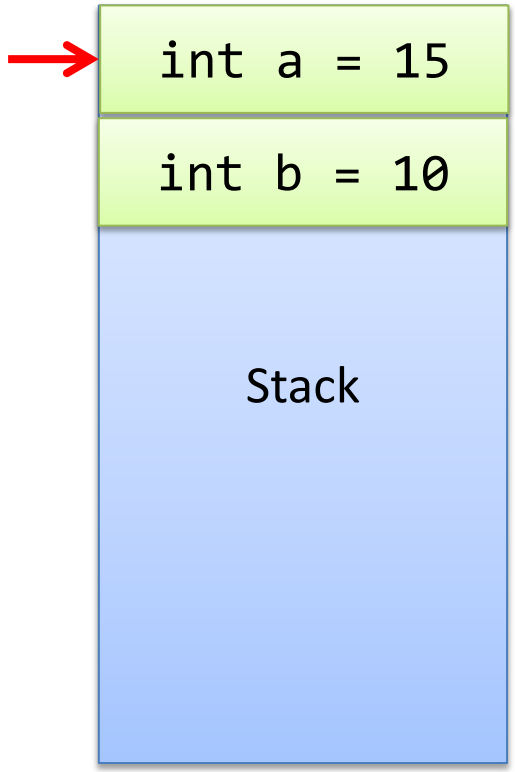


C++: Speichverwaltung

- **Stack**
  - **Stack** wird u.a. für lokal definierte Variablen verwendet
  - Bei Verlassen der scope {} wird der Stack zurückgesetzt

```
int main(int argc, const char* argv[])
{
    int b = 10;

    while (true)
    {
        int a = b + 5;
        break;
    }
}
```

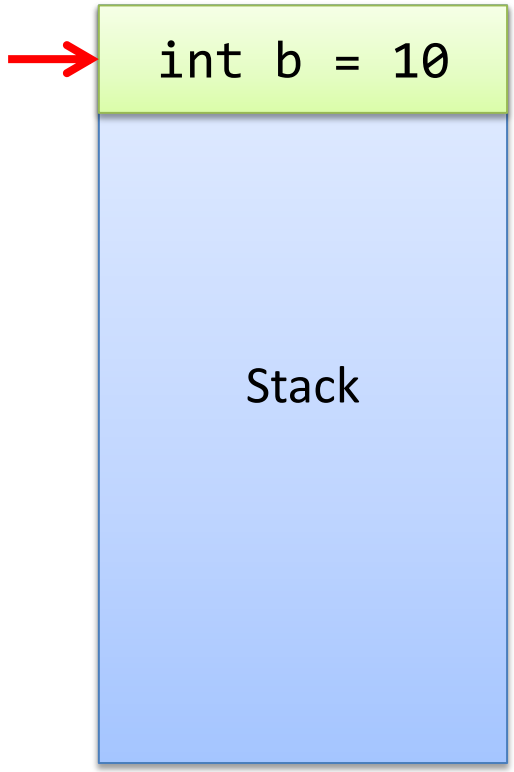


C++: Speichverwaltung

- **Stack**
  - **Stack** wird u.a. für lokal definierte Variablen verwendet
  - Bei Verlassen der scope {} wird der Stack zurückgesetzt

```
int main(int argc, const char* argv[])
{
    int b = 10;

    while (true)
    {
        int a = b + 5;
        break;
    }
}
```



C++: Speichverwaltung

- **Heap**
  - Der Heap bleibt die ganze Anwendung über erhalten
  - `new` liefert eine Adresse aus dem **Heap** und markiert diese (und die Größe des Bereichs) als benutzt
  - `delete` markiert die Adresse wieder als frei

C++: Speicherverwaltung

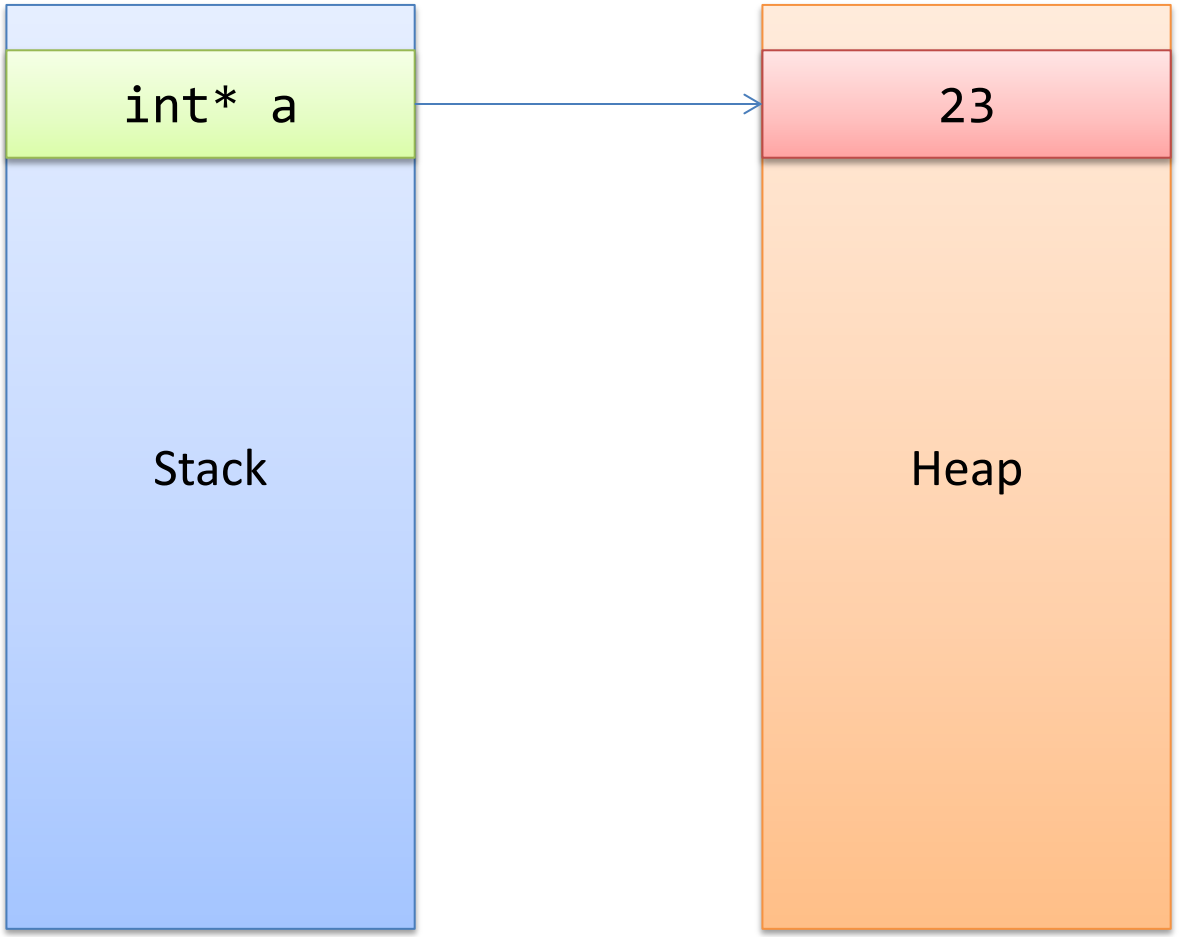
- Jede Variable in C++ hat eine Adresse, diese bekommt man mit &

```
int a; int* ap = &a; // ap zeigt auf a
```

- Zugriff erfolgt durch Dereferenzierung: `*zeiger`
- Greift auf den Wert zu, auf den `zeiger` zeigt

```
int* a = new int;
*a = 23;
```

C++: Speicherverwaltung



C++: Speicherverwaltung

- Statt `(*x).y` kann man auch `x->y` schreiben

```
class C {
public:
    int a;
};

C* c = new C;
c->a = 23;
```

- Statt `*(x + 10)` kann man `x[10]` schreiben
  - x + 10 -> Adresse + 10 * Größe des Datentyps

C++: Speicherverwaltung

- Jedes `new` muss von einem `delete` gefolgt werden
- Wenn möglich: Stack!
  - Nicht immer möglich, z.B. Array mit zur Compilierungszeit unbekannter Größe!
  - Standardbibliothek nutzen!
    - std::vector für arrays
    - std::string für Zeichenketten
    - Advanced: std::unique_ptr<>

```
MyClass a;
a.doSomething();
```

```
int b[100];
```

```
std::vector<int> container;
container.push_back(23);
container.push_back(44);
```

C++: Speicherverwaltung

- C++ kennt Destruktoren
- Analog zum Konstruktor, wird aufgerufen, wenn das Objekt nicht mehr benötigt wird

```
class ClassWithDestructor {
public:
     ~ClassWithDestructor () {
        std::cout << "Dtor";
    }
};

{
    ClassWithDestructor c;
    // Code
    // Hier wird c.~ClassWithDestructor() aufgerufen
}
```

C++: Speicherverwaltung

- Destruktor wird aufgerufen
  - Wenn die Variable „out-of-scope“ geht
  - Wenn `delete` aufgerufen wird
  - Syntax: `~KlassenName`

- Vorsicht bei Pointern in Klassen
  - Beim Kopieren werden nur Pointer kopiert, nicht der Bereich auf dem Heap, auf den der Pointer zeigt
  - Nach Aufruf von `delete` zeigt der Pointer auf ungültigen Speicher

C++: Speicherverwaltung

```
class Foo {
    int* bar;
public:
    Foo() {
        bar = new int[10];
    }
    ~Foo() {
        delete[] bar;
    }
};
```

```
Foo foo1;
{
    Foo foo2 = foo1;
}
// foo2 goes out of scope here!
```

- Wenn foo2 „out of scope“ geht, wird sein Destruktor aufgerufen
  - Der Pointer in foo1 zeigt aber auf denselben Speicherbereich
  - foo1 besitzt also einen Zeiger auf ungültigen Speicher!

C++: Speicherverwaltung

- Null-Zeiger
  - `int* a = nullptr;`
  - Null-Zeiger sind nützlich, um optionale Dinge anzugeben
  - `void Read (const int count, int* targetBuffer, ErrorCode* errorCode = nullptr);`

- Nach dem Freigeben von Speicher: Pointer auf nullptr setzen
  - Ansonsten wird weiterhin auf den freigegebenen Speicher verwiesen

C++: Zeiger und Referenzen

- Java: Immer Call-by-value
  - **Referenzen** werden kopiert!
  - `void swap (int a, int b)` kann so nicht funktionieren

- In C++: Standard Call-by-value
  - Für alle Typen
  - Vorsicht: Es werden **immer** Kopien angelegt
  - Übergeben großer Objekte kostet Zeit

C++: Call-by-reference

- Stattdessen: Zeiger übergeben

```
void swap(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int a, b;
swap(&a, &b);
```

- Das ist allerdings fehleranfällig
  - `a` oder `b` kann nicht gesetzt sein (**`nullptr`**)

C++: Call-by-reference

- Besser: Referenzen

```
void swap(int& a, int& b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int a,b;
swap(a,b);
```

- Referenzen können nicht null sein
  - Optionale Parameter sind also nicht möglich

- Gleiche Performance wie Pointer

C++: Referenzen

- Referenzen machen nur als Funktionsparameter wirklich Sinn
- Referenzen als Membervariable: Möglich, aber fehleranfällig
  - Jetzt könnte eine Referenz auf ein „totes“ Objekt zeigen …
- Empfehlung: Bei Funktionen **alles**, was kein primitiver Typ ist (`float`, `int`), als (const) Referenz übergeben
- Ansonsten Zeiger benutzen
  - Output-Variablen
  - Verknüpfungen in Datenstrukturen

C++: Klassen

- Syntax-Unterschiede:
  - Java:
    ```
    class A extends Foo
    ```
  - **C++:**
    ```
    class A : public Foo
    ```
- Beim Ableiten muss die Sichtbarkeit angegeben werden
  - Standard ist private
  - Einfach immer mit `: public Parent` ableiten
- C++ kennt keine Interfaces

C++: Klassen

- Sichtbarkeiten
  - Keine Sichtbarkeit bei Klassen
  - Innerhalb einer Klasse wird die Sichtbarkeit wird pro Block gesetzt

```
public: // Alles was danach kommt ist public
void A (); int b;

private: // Ab hier private
int m_memberVariable;
```

C++: Klassen

- Beispiel:

```
class Foo {
public:

    int Do (int c)  {
        return a * c + b;
    }

private:
    int a, b;
};
```

```
struct Foo {

    int a, b;

    int Do (int c)  {
        return a * c + b;
    }

};
```

- `struct` und `class` sind quasi äquivalent
  - Einziger Unterschied: `struct` Member standardmäßig `public` (bei `class` `private`)
  - `struct` historisch durch C bedingt
  - Für komplexere Typen immer `class` verwenden!

C++: Klassen

- In Java ist jede Methode virtual

```
public class Foo
{
    void Method () { System.out.println ("foo"); }
}

public class Bar extends Foo
{
    void Method () { System.out.println ("bar"); }
}

Foo f = new Bar ();
f.Method (); // Ausgabe: bar
```

C++: Klassen

```
class Foo {
    void Method () {
        std::cout << "foo";
    }
};

class Bar : public Foo {
    void Method () {
        std::cout << "bar";
    }
};

Foo* f = new Bar();
Bar* b = new Bar();
f->Method(); // Ausgabe: foo
b->Method(); // Ausgabe: bar
```

C++: Klassen

- C++: Statische Bindung
  - Zeiger auf `Foo*`, also `Foo::Method`
  - Zeiger auf `Bar*`, also `Bar::Method`

```
class Foo {
    void Method () {
        std::cout << "foo";
    }
};

class Bar : public Foo {
    void Method () {
        std::cout << "bar";
    }
};

Foo* f = new Bar();
Bar* b = new Bar();
f->Method(); // Ausgabe: foo
b->Method(); // Ausgabe: bar
```

- Dynamische Bindung: `virtual`

```
class Foo {
    virtual void Method ()
    {
        /*...*/
    }
};
```

- Achtung! `Virtual` auch bei Destruktoren notwendig

C++: Klassen

- Interfaces: Abstrakte Klassen
  - Mindestens eine Methode ist als „pure virtual“ deklariert
  - Keine Implementierung
  - **Muss** von abgeleiteten Klassen implementiert werden

```
class IStream {

    virtual int Read (int count, void* targetBuffer)= 0;
    // durch das = 0 wird die Methode pure virtual

};
```

- `IStream` ist nun abstrakt
  - Kann nicht instanziiert werden
  - Virtueller Destruktor sollte noch hinzugefügt werden!

C++: Konvertieren von Typen

- In Java:

```
double bar;
int foo = (int)bar;
```

- Gültige C Syntax,
  - Außerdem in C++: Spezielle cast-keywords
  - hier nicht näher erklärt

- Verhalten wie in Java

C++: Operator overloading

- In C++ kann man Operatoren überladen
- Nicht nur String + String wie in Java, sondern für alles und jeden

```
class Foo {
    Foo operator+(const Foo& a, const Foo& b);
};
```

- Die Standard-I/O-Bibliothek macht davon Gebrauch

```
#include <iostream>
std::cout << "Hello world" << std::endl;
```

C++: Container

- In C++ gibt es eine große, generische Standard-Bibliothek, die u.a. auch Container-Klassen enthält
- Ähnlich den Java-Collections
  - `std::vector<T>`
    - Lineares Array
    - Im Zweifel immer `std::vector` verwenden
    - `#include <vector>`
  - `std::map<K, V>`
    - Binär-Baum
    - Assoziativer Container (vgl. Wörterbuch)
    - `#include <map>`

C++: Container

- `std::set<K>`
  - Set, jeder Eintrag kann nur einmal vorhanden sein
  - `#include <set>`
- `std::unordered_map<K, V>`, `std::unordered_set<K>`
  - Wie Map und Set, jedoch ohne Ordnung
- `std::list, std::stack, std::queue, etc.`

C++ Iteratoren

- Jeder Container besitzt einen Iteratortyp
  - Referenz auf ein Containerelement
  - Kann inkrementiert und dereferenziert werden
  - Weitere Operatoren möglich (z.B. `--`, `+=10`)

```
std::vector<int> a;

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

C++: Strings

- In C ist ein `char*` ein String
  - Muss mit einem Null-Byte terminiert werden
  - Fehleranfällig
- `std::string`
  - `#include <string>`
  - `operator==`, `operator+`
  - Weitere Stringfunktionen (http://www.cplusplus.com/reference/string/)
- `.c_str()` liefert einen gültigen C-String (`char*`)

C++: iostream

- C++ bietet eine Bibliothek für Ein/Ausgabe: `iostream`
- Im Grunde drei Klassen: `istream`, `ostream`, und `iostream`
  - Für Dateien: `ifstream`, `ofstream`, `fstream`

- Standardausgabe/Eingabe über globale Variablen
  - `std::cout`, `std::cin`

- iostreams überladen Operatoren: `>>` und `<<`
  - `std::cout << "Text" << 23 << std::endl;`

C++: iostream

- Die C++ Streams sind sehr mächtig
  - Man kann die Operatoren für eigene Typen definieren
  - Typsicheres Laden/Speichern

- Dafür auch mit Problemen
  - Fehlerbehandlung kompliziert: Entweder über Ausnahmen oder auf `good()`/`bad()` testen.
  - Performance oftmals problematisch

C++: Const

- In C++ kann man Variablen (inkl. Pointer/Referenzen) und Methoden als `const` markieren
- Beispiele:

```
const int a = 10;
```

```
int a = 10;
const int * b = &a;
int const * c = &a;
int * const d = &a;
```

```
int GetWidth () const {
    return width;
 }
```

- Bedeutung:
  - Variable: kann nach Initialisierung nicht mehr geändert werden
  - Methode: kann keine Member-Variablen der Instanz ändern
- Faustregel: Standardmäßig alles als `const` markieren
  - Außer es geht nicht
  - Erleichtert Debugging & Optimierungen

C++: auto

- Seit C++11 gibt es auch „auto“
- Compiler stellt den Typ zur Compile-Zeit basierend auf dem Typ des Ausdrucks
  - Das Programm ist immer noch statisch typisiert!

```
std::vector<int> a;

for (auto it = a.begin(); it != a.end(); ++it) {
    std::cout << *it << std::endl;
}
```

C++: Zusammenfassung

- Ähnlich zu Java
  - Klassen, Methoden, Ableiten …
- Ähnlich zu C
  - Zeiger, manuelle Speicherverwaltung, …
- Mächtige, komplexe Sprache
  - Zu Beginn auf eine Untermenge konzentrieren
  - „Java“ in C++ schreiben statt C
  - Klassen & automatische Speicherverwaltung mittels Container
- "In C++ it's harder to shoot yourself in the foot, but when you do, you blow off your whole leg." — Bjarne Stroustrup

C++ Referenzen

- Referenz
  - http://www.cplusplus.com/reference/
- Online Tutorial
  - http://www.cplusplus.com/doc/tutorial/
- Buch

  Die C++-Programmiersprache / The C++ Programming Language (4th Edition)
  Bjarne Stroustrup