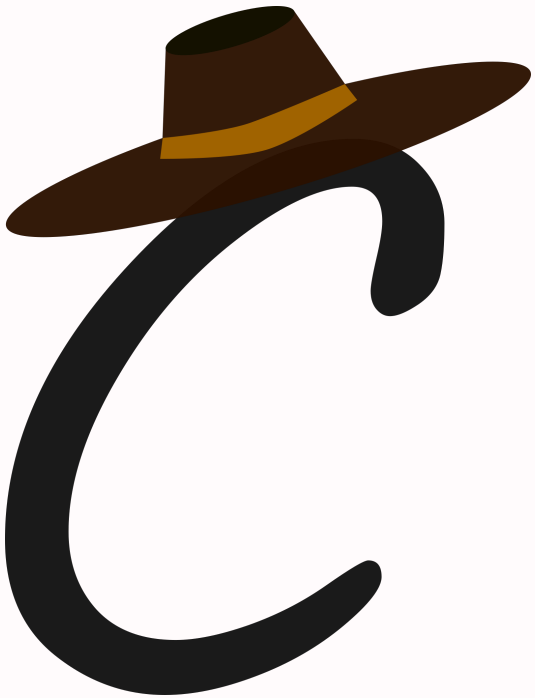
C++ für Java-Programmierer

Allgemeines über C++

Designziele

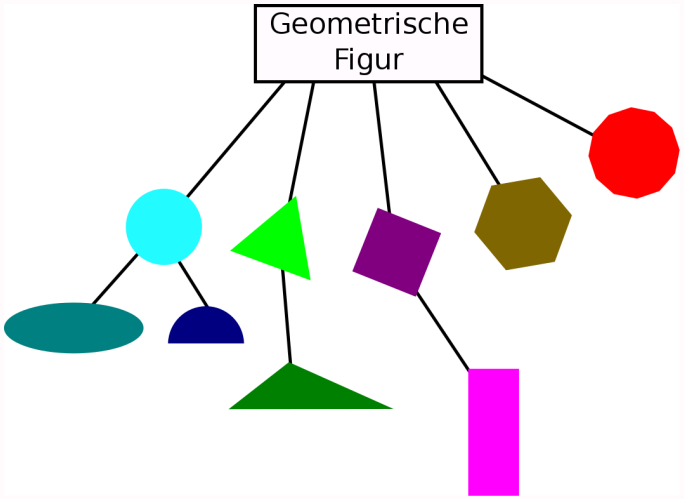


- “ein besseres C”

Designziele

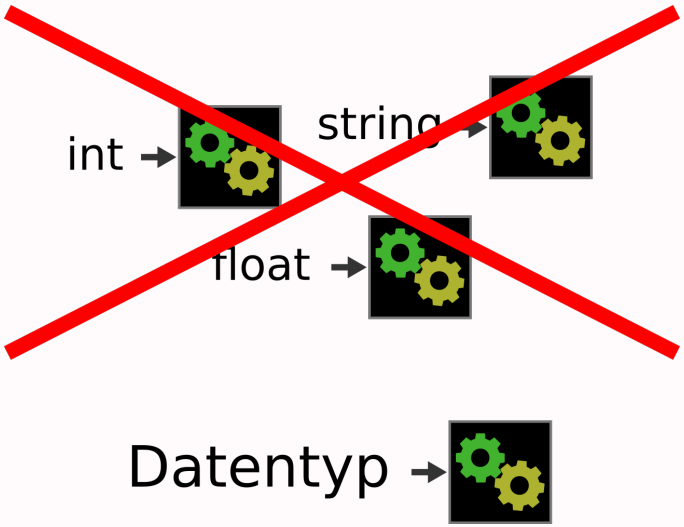
- “ein besseres C”
- Datenabstraktion

Designziele



- “ein besseres C”
- Datenabstraktion
- Objektorientiert

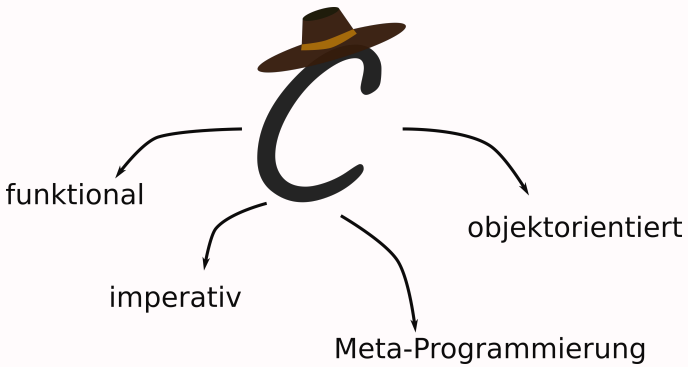
Designziele



[source: Wikimedia]

- “ein besseres C”
- Datenabstraktion
- Objektorientiert
- Generisch

Designziele



- “ein besseres C”
- Datenabstraktion
- Objektorientiert
- Generisch
- Multi-Paradigmen-Sprache

Geschichte (vor 2000)

[source: Bjarne Stroustrup's Homepage]

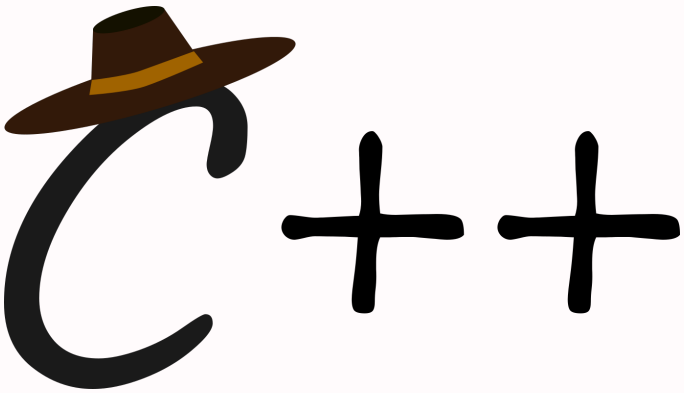
- Bjarne Stroustrup

Geschichte (vor 2000)

[source: Daria]

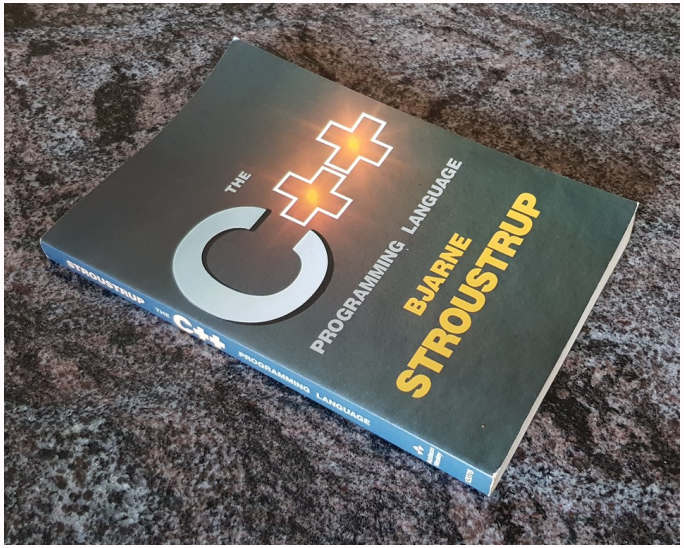
- Bjarne Stroustrup
- 1979: “C with classes”

Geschichte (vor 2000)



- Bjarne Stroustrup
- 1979: “C with classes”
- 1983: C++ als Name gewählt

Geschichte (vor 2000)



[source: Kean Walmsley]

- Bjarne Stroustrup
- 1979: “C with classes”
- 1983: C++ als Name gewählt
- 1985: “TC++PL”

Geschichte (vor 2000)

[source: Adventure Time Wiki]

- Bjarne Stroustrup
- 1979: “C with classes”
- 1983: C++ als Name gewählt
- 1985: “TC++PL”
- 1994: “The STL”

Geschichte (vor 2000)

[source: Standard C++]

- Bjarne Stroustrup
- 1979: “C with classes”
- 1983: C++ als Name gewählt
- 1985: “TC++PL”
- 1994: “The STL”
- 1998: C++98

Wo nutzt man C++?

[source: Wikimedia]

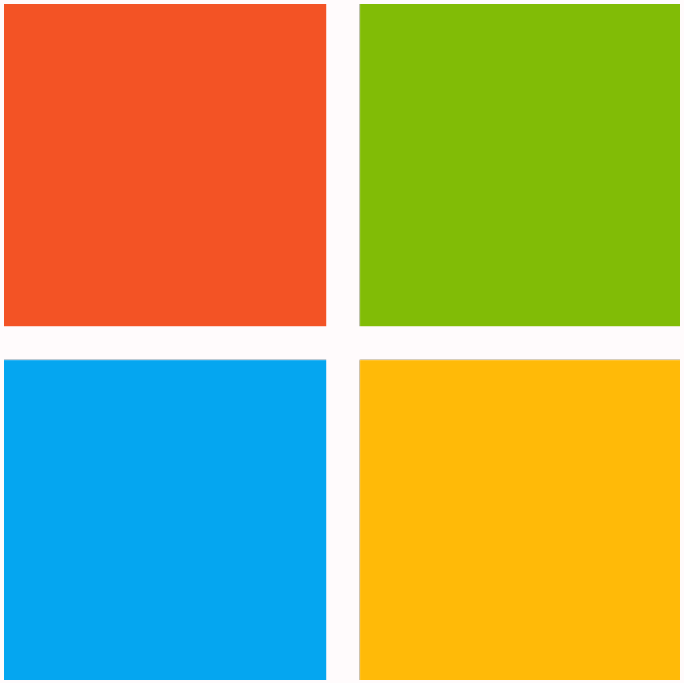
- Adobe: Photoshop, Acrobat, ...

Wo nutzt man C++?

[source: Wikimedia]

- Adobe: Photoshop, Acrobat, ...
- Apple: iOS, macOS, ...

Wo nutzt man C++?



[source: Wikimedia]

- Adobe: Photoshop, Acrobat, ...
- Apple: iOS, macOS, ...
- Microsoft: Terminal, DirectX, C#-Compiler, ...

Wo nutzt man C++?

[source: NASA Jet Propulsion Laboratory]

- Adobe: Photoshop, Acrobat, ...
- Apple: iOS, macOS, ...
- Microsoft: Terminal, DirectX, C#-Compiler, ...
- NASA: Mars-Rover Steuerung, ...

Wo nutzt man C++?

[source: Wikipedia]

- Adobe: Photoshop, Acrobat, ...
- Apple: iOS, macOS, ...
- Microsoft: Terminal, DirectX, C#-Compiler, ...
- NASA: Mars-Rover Steuerung, ...
- Computerspiele: Unreal Engine, Source, Elder Scrolls, ...

Erste Schritte

Kompilieren

[source: Wikimedia]

- “Standard”-Compiler: GCC (GNU Compiler Collection)

Kompilieren

[source: Wikimedia]

- “Standard”-Compiler: GCC (GNU Compiler Collection)
- `C`, `C++`, `Java`, `Fortran`, …

Kompilieren

[source: Wikimedia]

- “Standard”-Compiler: GCC (GNU Compiler Collection)
- `C`, `C++`, `Java`, `Fortran`, …
- Varianten für Windows

Kompilieren

[source: Wikimedia]

- “Standard”-Compiler: GCC (GNU Compiler Collection)
- `C`, `C++`, `Java`, `Fortran`, …
- Varianten für Windows
- GCC ist geschrieben in `C++`

Hello, world!

hello_world.cpp:

```
#include <iostream>

int main(){
  std::cout << "Hello, world!\n";
  return 0;
}
```

Übersetzen in vier Schritten

- Preprocessing: Bearbeitet den Quellcode auf Text-Basis (Kopieren durch `#include` etc.)

Übersetzen in vier Schritten

- Preprocessing: Bearbeitet den Quellcode auf Text-Basis (Kopieren durch `#include` etc.)
- Compiling: Übersetzt Quellcode-Dateien zu Maschinencode

Übersetzen in vier Schritten

- Preprocessing: Bearbeitet den Quellcode auf Text-Basis (Kopieren durch `#include` etc.)
- Compiling: Übersetzt Quellcode-Dateien zu Maschinencode
- Assembling: Erzeugt aus Maschinencode “Object code”

Übersetzen in vier Schritten

- Preprocessing: Bearbeitet den Quellcode auf Text-Basis (Kopieren durch `#include` etc.)
- Compiling: Übersetzt Quellcode-Dateien zu Maschinencode
- Assembling: Erzeugt aus Maschinencode “Object code”
- Linking: Verbindet alle Objekte zu einer einzigen (ausführbaren) Datei

Übersetzen in vier Schritten

- Preprocessing: Bearbeitet den Quellcode auf Text-Basis (Kopieren durch `#include` etc.)
- Compiling: Übersetzt Quellcode-Dateien zu Maschinencode
- Assembling: Erzeugt aus Maschinencode “Object code”
- Linking: Verbindet alle Objekte zu einer einzigen (ausführbaren) Datei

- Manuelles Stoppen nach jedem Schritt möglich

Übersetzen in vier Schritten

hello_world.cpp:

```
#include <iostream>

int main(){
  std::cout << "Hello, world!\n";
  return 0;
}
```

- 6 Zeilen / 83 Byte

Übersetzen in vier Schritten

`hello_world_preprocessed.cpp`:

```
# 1 "hello_world.cpp"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "hello_world.cpp"
# 1 "/usr/include/c++/4.8/iostream" 1 3
# 36 "/usr/include/c++/4.8/iostream" 3
```

⋮

- 17 557 Zeilen / 410 KiB

Übersetzen in vier Schritten

hello_world.s:

```
  .file "hello_world.cpp"
  .local  _ZStL8__ioinit
  .comm _ZStL8__ioinit,1,1
  .section  .rodata
.LC0:
  .string "Hello , world!\n"
  .text
  .globl  main
```

⋮

- 81 Zeilen / 1.69 KiB

Übersetzen in vier Schritten

`hello_world.obj`:

```
// nicht mit einem Texteditor anzeigbar
```

- 2.42 KiB

Übersetzen in vier Schritten

`a.out`:

```
// nicht mit einem Texteditor anzeigbar
```

- 8.98 KiB

GCC

`g++ hello_world.cpp`: kompiliert "`hello_world.cpp`" und erzeugt eine ausführbare Datei "`a.out`".

`-o outfile`	Benennt die ausführbare Datei `outfile` statt `a.out`
`-std=c++14`	Kompiliert unter Nutzung des "2014 ISO C++" Standards – benötigt passende Version des Compilers
`-E`	Stoppt nach Preprocessing
`-S`	Stoppt nach Maschinencode-Generation
`-c`	Stoppt nach Objektcode-Generation

Ein- und Ausgabe

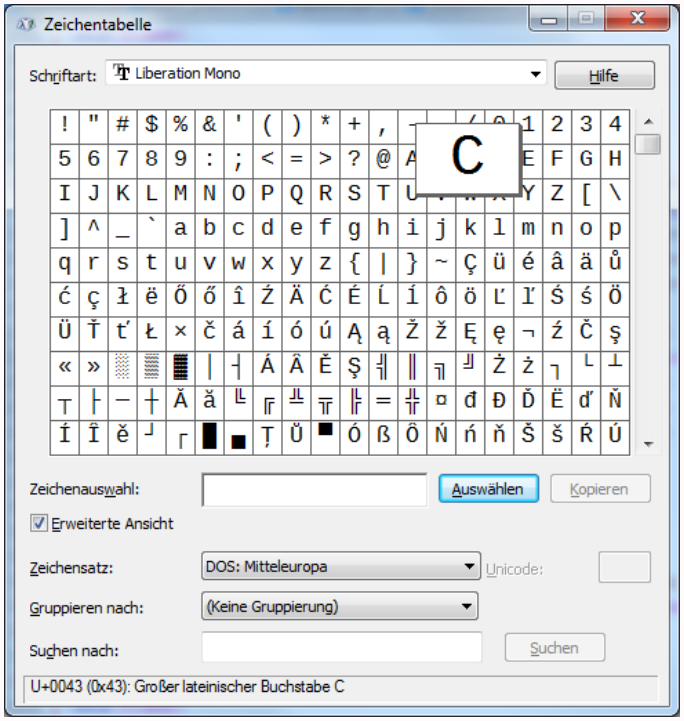
```
#include <iostream>
using namespace std;

int main(){
  cout << "Was ist deine Lieblingsfarbe? ";
  string farbe;
  cin >> farbe;
  cout << farbe +
      " ist eine schoene Farbe!" << endl;

  return 0;
}
```

Datentypen

[source: Wikimedia]

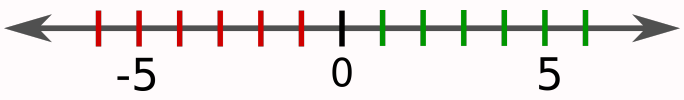
- `bool`

Datentypen



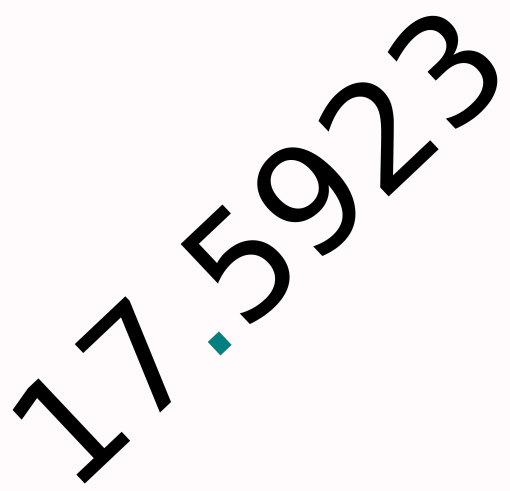
- `bool`
- `char`

Datentypen



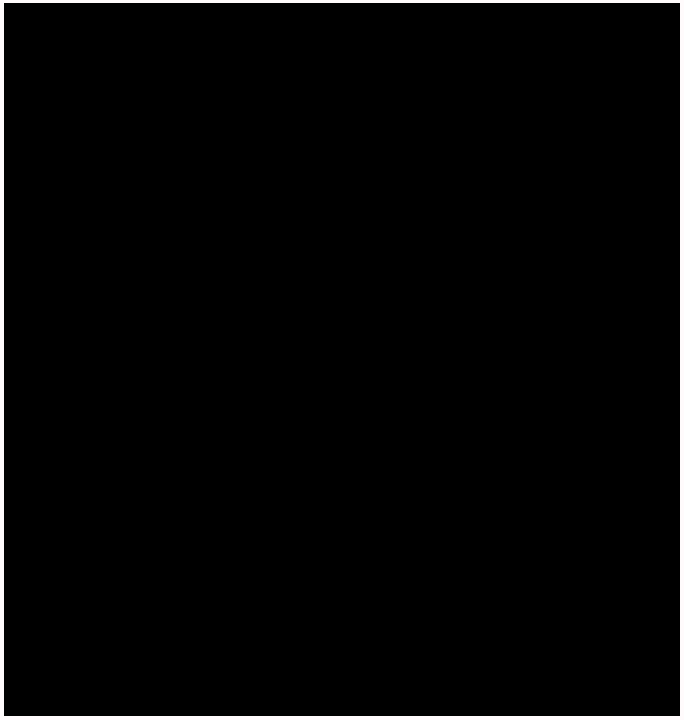
- `bool`
- `char`
- `int` (inkl. Varianten)

Datentypen



- `bool`
- `char`
- `int` (inkl. Varianten)
- `float` (und `double`)

Datentypen



- `bool`
- `char`
- `int` (inkl. Varianten)
- `float` (und `double`)
- `void`

Operatoren allgemein

- Arithmetik (`+`, `-`, `*`, `/`); Potenzierung (`^`, `**`) existiert nicht

Operatoren allgemein

- Arithmetik (`+`, `-`, `*`, `/`); Potenzierung (`^`, `**`) existiert nicht
- Modulo (`%`)

Operatoren allgemein

- Arithmetik (`+`, `-`, `*`, `/`); Potenzierung (`^`, `**`) existiert nicht
- Modulo (`%`)
- Gleichheit (`==`)

Operatoren allgemein

- Arithmetik (`+`, `-`, `*`, `/`); Potenzierung (`^`, `**`) existiert nicht
- Modulo (`%`)
- Gleichheit (`==`)
- “Ungleichheit” (`!=`)

Operatoren allgemein

- Arithmetik (`+`, `-`, `*`, `/`); Potenzierung (`^`, `**`) existiert nicht
- Modulo (`%`)
- Gleichheit (`==`)
- “Ungleichheit” (`!=`)
- Vergleiche (`<`, `>`, `<=`, `>=`)

Operatoren allgemein

- Arithmetik (`+`, `-`, `*`, `/`); Potenzierung (`^`, `**`) existiert nicht
- Modulo (`%`)
- Gleichheit (`==`)
- “Ungleichheit” (`!=`)
- Vergleiche (`<`, `>`, `<=`, `>=`)
- Logik (`&&`, `||`, `!`)

Weitere Operatoren

- Bitweise (>>, <<, &, |, ^)

Weitere Operatoren

- Bitweise (`>>`, `<<`, `&`, `|`, `^`)
- “ternary conditional” (`? :`)

Weitere Operatoren

- Bitweise (`>>`, `<<`, `&`, `|`, `^`)
- “ternary conditional” (`?:`); `(a > b) ? a : b` liefert Maximum aus $\{\texttt{a}, \texttt{b}\}$

Weitere Operatoren

- Bitweise (`>>`, `<<`, `&`, `|`, `^`)
- “ternary conditional” (`? :`)
- “three-way comparison” (`<=>`)[1]

[1] Seit C++20

Weitere Operatoren

- Bitweise (`>>`, `<<`, `&`, `|`, `^`)
- “ternary conditional” (`?:`)
- “three-way comparison” (`<=>`)[1]; `a <=> b` liefert negative Zahl falls `a < b`, `0`, falls `a = b` und positive Zahl, falls `a > b`

[1] Seit C++20

Weitere Operatoren

- Bitweise (`>>`, `<<`, `&`, `|`, `^`)
- “ternary conditional” (`?:`)
- “three-way comparison” (`<=>`)[1]
- “Sequencing” (`,`)

[1] Seit C++20

Weitere Operatoren

- Bitweise (`>>`, `<<`, `&`, `|`, `^`)
- “ternary conditional” (`?:`)
- “three-way comparison” (`<=>`)[1]
- “Sequencing” (`,`); `a = (b=3, b+2)` setzt `b` auf 3 und danach `a` auf `b+2`

[1] Seit C++20

Weitere Operatoren

- Bitweise (`>>`, `<<`, `&`, `|`, `^`)
- “ternary conditional” (`?:`)
- “three-way comparison” (`<=>`)[1]
- “Sequencing” (`,`)
- Sonstige (`( )` (Casting), `::` (Scope), `[ ]` (Zugriff), `sizeof`, `new`, `delete`, …)

[1] Seit C++20

Kurzschreibweisen

C++ unterstützt zusammengesetzte Operatoren:

```
int z = 22;

z += 5; // z = z + 5; (z == 27)
z -= 7; // z = z - 7; (z == 20)
z *= 2; // z = z * 2; (z == 40)
z /= 4; // z = z / 4; (z == 10)
```

Kurzschreibweisen

C++ unterstützt zusammengesetzte Operatoren:

```
int z = 22;

z += 5; // z = z + 5; (z == 27)
z -= 7; // z = z - 7; (z == 20)
z *= 2; // z = z * 2; (z == 40)
z /= 4; // z = z / 4; (z == 10)
```

Auch für Modulo und bitweise Operationen (%=, <<=, ...)

Inkrement und Dekrement

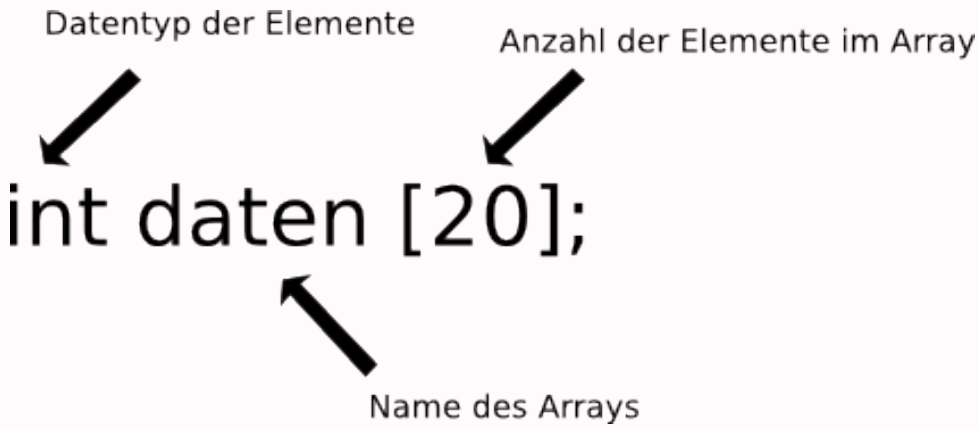
C++ unterstützt Inkrement- und Dekrement-Operatoren – jeweils in der Präfix- und der Postfix-Variante.

```
int i = 1, j = 1;
int a, b, c, d;

a = i++; // a = i; i = i+1; (a == 1, i == 2)
b = i--; // b = i; i = i-1; (b == 2, i == 1)
c = ++j; // j = j+1; c = j; (c == 2, j == 2)
d = --j; // j = j-1; d = j; (c == 1, j == 1)
```

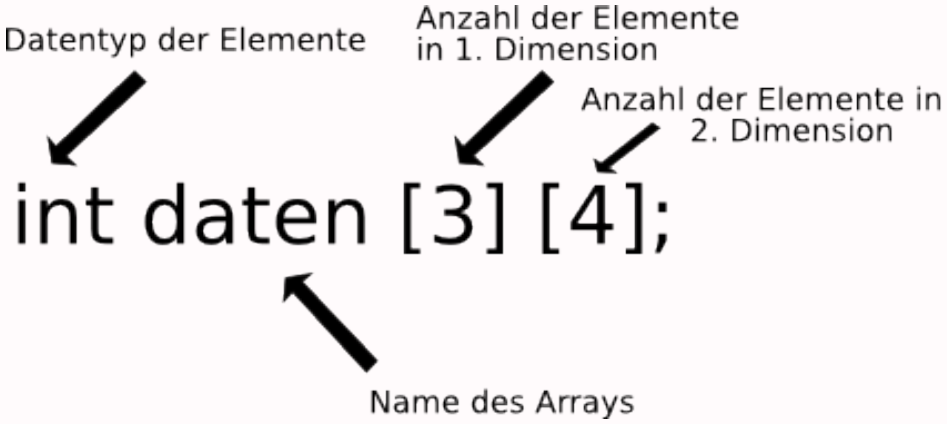
Arrays

- Sammlung von Elementen gleichen Typs
- kein eigener Typ, kein Objekt
- statische Länge
- “gültige” Indizes: 0 bis Länge -1
- keine Laufzeitüberprüfung auf gültigen Index

Initialisierung von Arrays



Initialisierung von Arrays



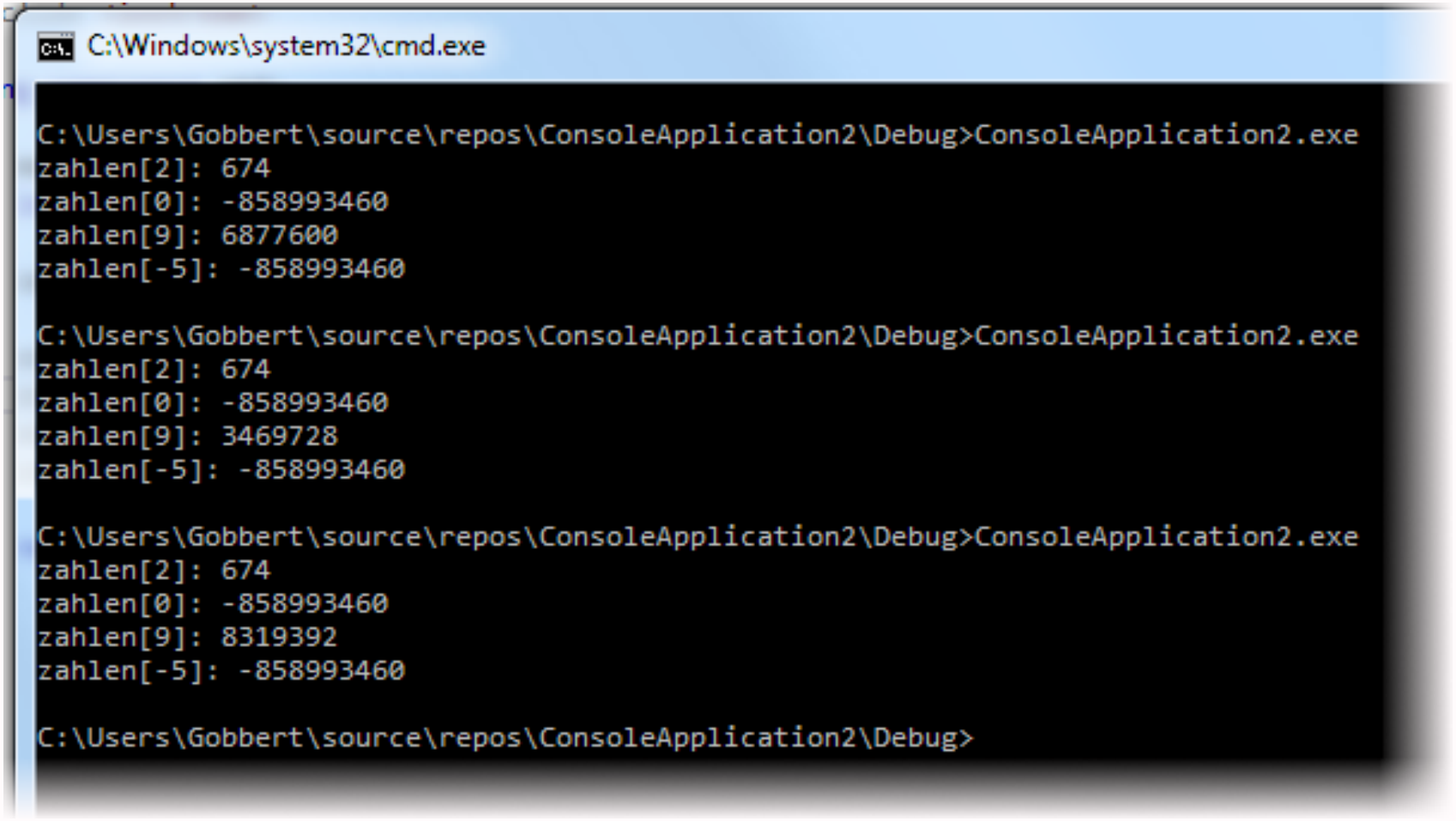
Array-Beispiel

```
int zahlen[5]; // Array mit Positionen 0 bis 4

zahlen[2] = 674; // weist dem dritten Element den
    Wert 674 zu
cout << zahlen[2] << endl;

// das Element wurde nicht initialisiert
cout << zahlen[0] << endl;
// Indizes liegen ausserhalb der Grenzen
cout << zahlen[9] << endl;
cout << zahlen[-5] << endl;
```

Array-Beispiel

```
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/GUT-Cpp $ ./arrays2.o
zahlen[2]: 674
zahlen[0]: 1
zahlen[9]: 32767
zahlen[-5]: 32652
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/GUT-Cpp $ ./arrays2.o
zahlen[2]: 674
zahlen[0]: 1
zahlen[9]: 32764
zahlen[-5]: 32582
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/GUT-Cpp $ ./arrays2.o
zahlen[2]: 674
zahlen[0]: 1
zahlen[9]: 32764
zahlen[-5]: 32759
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/GUT-Cpp $
```

Array-Beispiel



Eigenschaften von Strings

- Variable Länge
- Konkatenation mit “`+`”
- “Array” von Zeichen
- Vergleichsoperationen anwendbar (`==`, `<`, `<=`, …)
- Member-Funktionen (`substr()`, `find()`, …)

String-Beispiel

```
string s, r;

cout << "Bitte ein Wort eingeben: ";
cin >> s;

for(int i = s.length()-1; i >= 0; i--){
  r += s[i];
}

if(s == r){
  cout << s + " ist ein Palindrom!" << endl;
}else{
  cout << s + " ist kein Palindrom!" << endl;
}
```

String-Beispiel

```
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ g++ -o palindrom.o palindrom.cpp
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ ./palindrom.o
Bitte ein Wort eingeben: Palindrom
Palindrom ist kein Palindrom!
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ ./palindrom.o
Bitte ein Wort eingeben: reliefpfeiler
reliefpfeiler ist ein Palindrom!
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $
```

Eigenschaften von Funktionen

- Funktionen überladen
- Default-Parameter
- call-by-*reference*: eigentliches Objekt übergeben
- call-by-*value*: Kopie eines Objekts/Werts übergeben

Überladungs-Beispiel

```
int summe(int a, int b, int c){
  return a + b + c;
}
int summe(int a, int b){
  return a + b;
}

int main(){

  cout << summe(1,2,3) << endl; // 1 + 2 + 3 = 6
  cout << summe(1,2) << endl;   // 1 + 2 = 3

  return 0;
}
```

Default-Parameter-Beispiel

```
int summe(int a, int b, int c = 0){
  return a + b + c;
}

int main(){
  cout << summe(1, 2, 3) << endl; // 1+2+3 = 6
  cout << summe(1, 2) << endl; // 1+2+0 = 3
  cout << summe(1) << endl; // error: "too few
      arguments in function call"

  return 0;
}
```

Parameterübergabe-Beispiel

```
void by_value(int i, int j){
  i += j;
  cout << i << endl;
}

void by_reference(int& i, int& j){
  i += j;
  cout << i << endl;
}
```

Parameterübergabe-Beispiel

```
int main(){
  int i = 5, j = 3;

  by_value(i, j);
  cout << i << endl;

  by_reference(i, j);
  cout << i << endl;

  return 0;
}
```

Parameterübergabe-Beispiel

```
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ g++ fkt_parameter.cpp -o fkt_parameter
.o
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ ./fkt_parameter.o
8
5
8
8
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $
```

Weitere Unterschiede zu Java

“Kleinstes” Programm

hello_world.cpp:

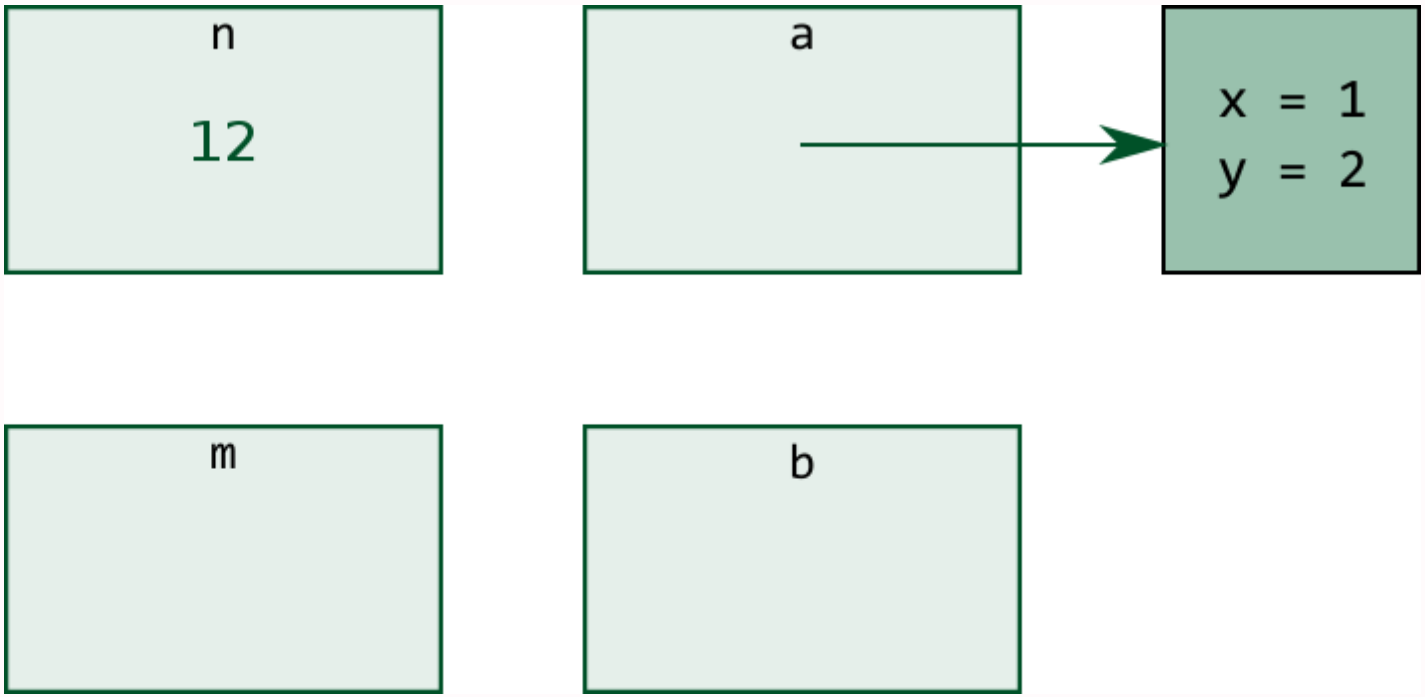
```cpp
int main(){
  std::cout << "Hello,
      world!\n";
  return 0;
}
```

hello_world.java:

```java
class hello_world{
  public static void
      main(String[]
      args){
    System.out.println(
        "Hello, world!"
        );
  }

}
```

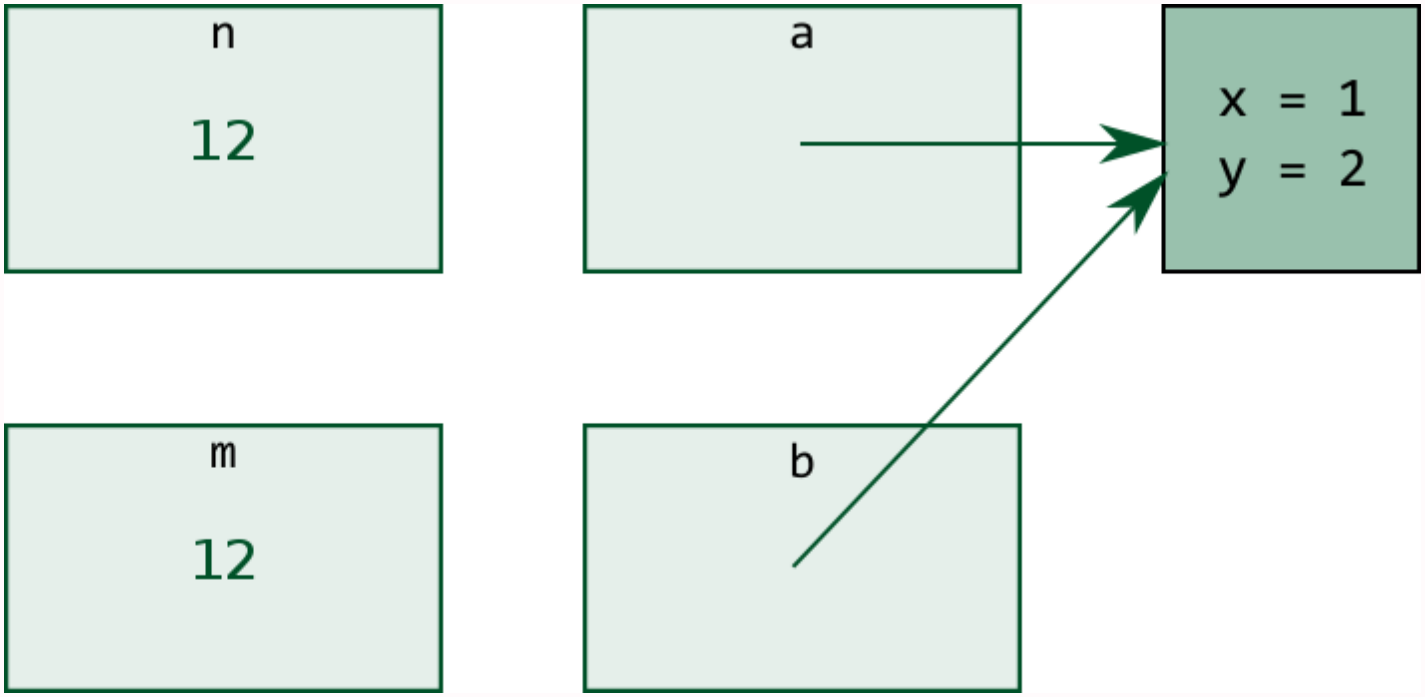
Objekte in Java

```
int n = 12;
int m;
point a = new point(1,2);
point b;
```



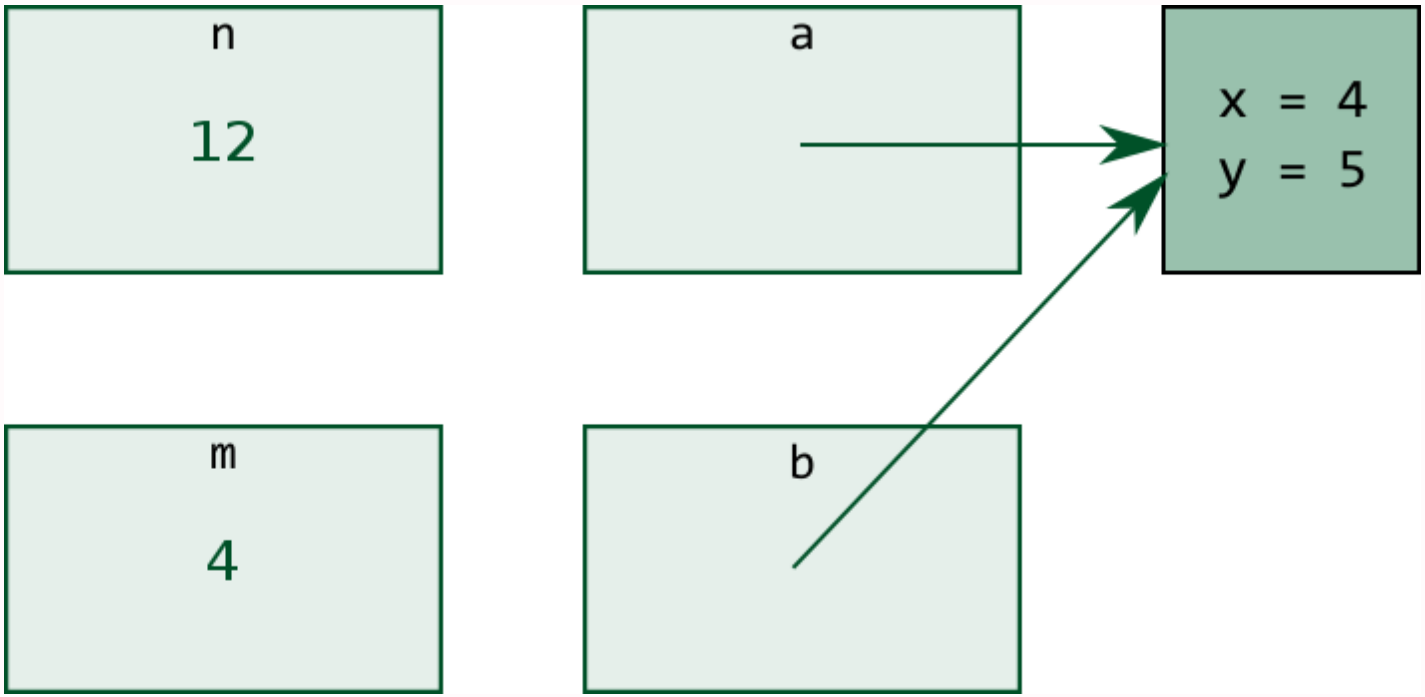
Objekte in Java

```
m = n;
b = a;
```



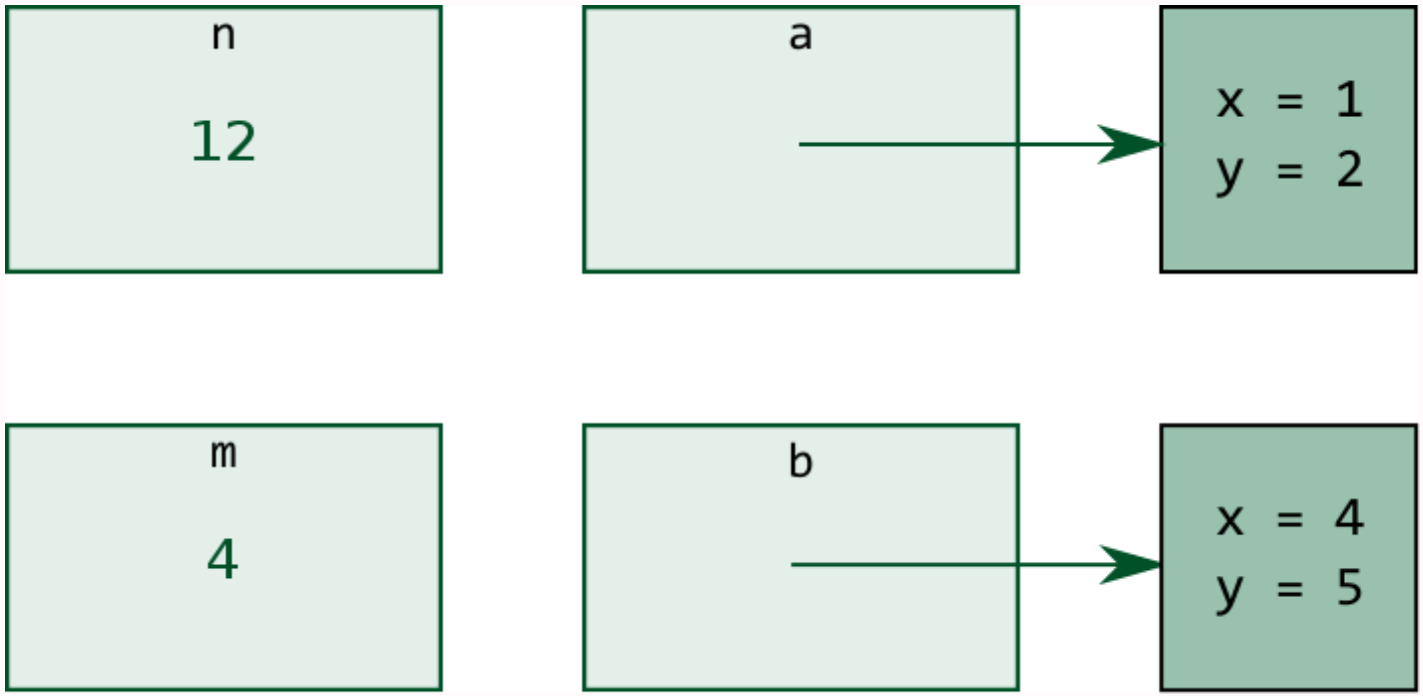
Objekte in Java

```
m = 4;
b.x = 4;
b.y = 5;
```



Objekte in C++

Gleiche Anweisungen wie vorhin:



Objekte in Java und C++ im Vergleich

Java:

```
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ javac objekte.java
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ java objekte
n = 12
m = "error: variable m might not have been initialized"
a = point@75ba3523 (1, 2)
b = "error: variable b might not have been initialized"

n = 12
m = 4
a = point@75ba3523 (4, 5)
b = point@75ba3523 (4, 5)
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $
```

C++:

```
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ g++ objekte.cpp -o objekte.o
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ ./objekte.o
n = 12
m = 0
a = 0x7ffde69e3f80 (1, 2)
b = 0x7ffde69e3f90 (4197440, 0)

n = 12
m = 4
a = 0x7ffde69e3f80 (1, 2)
b = 0x7ffde69e3f90 (4, 5)
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $
```

Objekte in C++

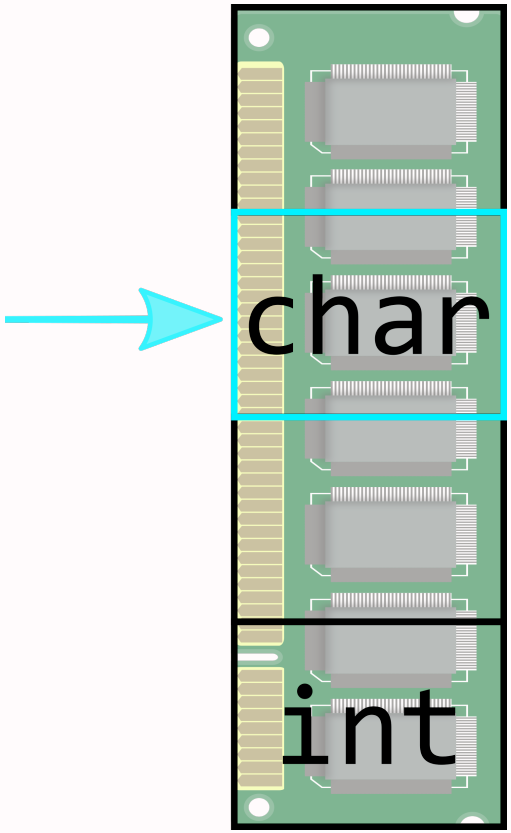
- Objektdeklaration erzeugt Objekt (ohne **new**)
- Objekt-Variablen enthalten Werte, keine Referenzen
- Keine Standard-Initialisierung
- Keine automatische “Garbage Collection”

Weitere Unterschiede

C++	Java
“write once compile anywhere”	“write once run anywhere”
nativer, ausführbarer Code	JVM
Überladen von Methoden und Operatoren	Überladen von Methoden
Objekte sind “values”	Objekte sind Referenzen

Pointer

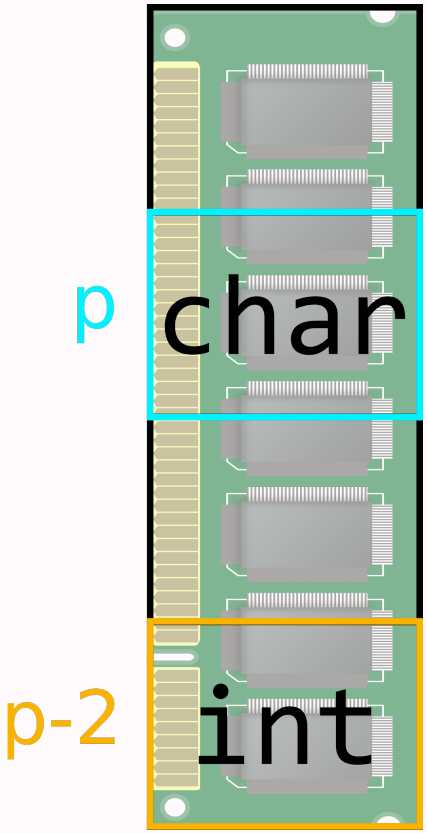
Pointer in C++



[source: Wikimedia]

- Adressen von Variablen

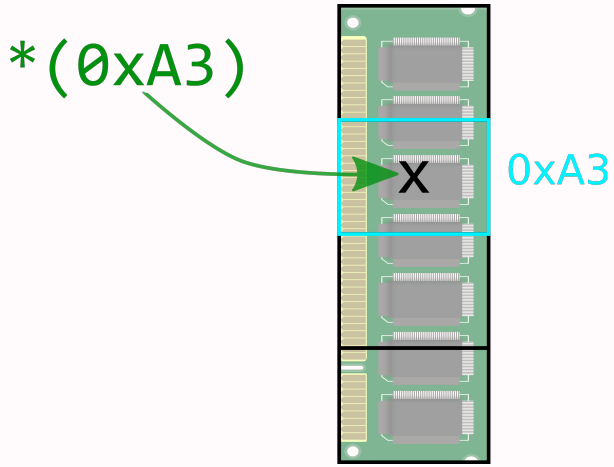
Pointer in C++



[source: Wikimedia]

- Adressen von Variablen
- “Pointerarithmetik”

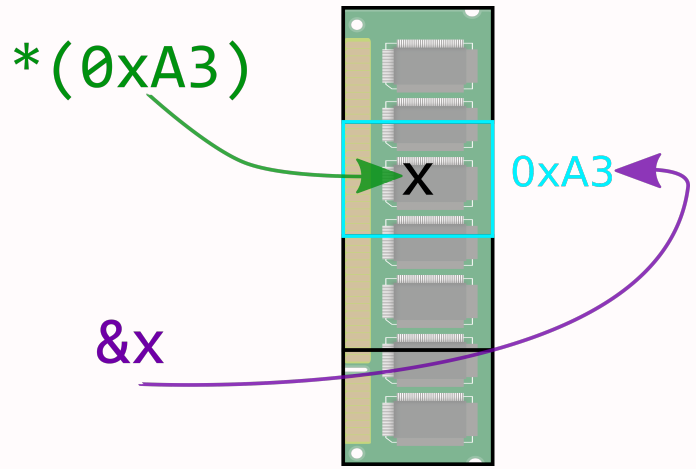
Pointer in C++



[source: Wikimedia]

- Adressen von Variablen
- “Pointerarithmetik”
- `*`: Adresse $\rightarrow$ Variable

Pointer in C++



[source: Wikimedia]

- Adressen von Variablen
- “Pointerarithmetik”
- *: Adresse → Variable
- &: Variable → Adresse

Pointer-Beispiele

```
int  x; // Integer
int* p; // Pointer auf Integer

 p = &x; // p = Adress-of(x)
*p = 1;  // Entspricht x = 1
*p = *p + 1; // Entspricht x = x + 1
```

Pointer-Beispiele

```
int  x; // Integer
int* p; // Pointer auf Integer

 p = &x; // p = Adress-of(x)
*p = 1;  // Entspricht x = 1
*p = *p + 1; // Entspricht x = x + 1
```

```
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ g++ pointer1.cpp -o pointer1.o
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ ./pointer1.o
 x: 2
*p: 2
&x: 0x7ffd7dda1244
&p: 0x7ffd7dda1248
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $
```

Pointer-Beispiele

```
int x;   // Integer
int y;   // Integer
int* px; // Pointer auf Integer
int* py; // Pointer auf Integer

px = &x; // px = Adress-of(x)
*px = 1; // entspricht x = 1
py = px+1; // ?
*py = 2;   // ?
```

Pointer-Beispiele

```
int x;   // Integer
int y;   // Integer
int* px; // Pointer auf Integer
int* py; // Pointer auf Integer

px = &x; // px = Adress-of(x)
*px = 1; // entspricht x = 1
py = px+1; // ?
*py = 2;   // ?
```

```
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ g++ pointer2.cpp -o pointer2.o
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ ./pointer2.o
x: 1
y: 2
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $
```

Pointer-Beispiele

Linux:

```
&x: 0x7ffe6c423f68      &y: 0x7ffe6c423f6c
px: 0x7ffe6c423f68      py: 0x7ffe6c423f6c

        => &y == py
```

Windows:

```
&x: 0042FD94    &y: 0042FD88
px: 0042FD94    py: 0042FD98

        => &y != py
```

- C++ überlässt Speicherlayout dem Betriebssystem

Pointer-Beispiele

```
void tausche(int* a, int* b){
  int t;
  t = *a;
  *a = *b;
  *b = t;
}
```

Pointer-Beispiele

```
void tausche(int* a, int* b){
  int t;
  t = *a;
  *a = *b;
  *b = t;
}
```

```
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ g++ tauschen.cpp -o tauschen.o
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ ./tauschen.o
Vorher: x: 2; y: 8
tausche(&x, &y);
Nachher: x: 8; y: 2
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $
```

Pointer und Arrays

```
int a[n];
```

- Array ist Pointer auf erstes Element
- C++ garantiert, dass `a+0` bis `a+n-1` nicht überschrieben werden
- `a == &(a[0])`

Pointer und Arrays

```
int a[n];
```

- Array ist Pointer auf erstes Element
- C++ garantiert, dass `a+0` bis `a+n-1` nicht überschrieben werden
- `a == &(a[0])`
- `a[i] == *(a + i)`

Pointer und Arrays

```
int a[n];
```

- Array ist Pointer auf erstes Element
- C++ garantiert, dass `a+0` bis `a+n-1` nicht überschrieben werden
- `a == &(a[0])`
- `a[i] == *(a + i) == *(i + a)`

Pointer und Arrays

```
int a[n];
```

- Array ist Pointer auf erstes Element
- C++ garantiert, dass `a+0` bis `a+n-1` nicht überschrieben werden
- `a == &(a[0])`
- `a[i] == *(a + i) == *(i + a) == i[a]`

Objektorientiertes Programmieren

Klassen und Objekte

Klasse: Datenstruktur mit Daten und darauf arbeitenden Funktionen

Objekt: Individuelles Exemplar einer Klasse

$\Rightarrow$ Kapselung von Datenstrukturen

Klassen-Beispiel

```
class Point{
public:
    Point();
    Point(int new_x, int new_y);
    void move(int dx, int dy);
    void print();

private:
    int x;
    int y;
};
```

Klassen-Beispiel

```
class Point{
public:
    Point();
    Point(int new_x, int new_y);
    void move(int dx, int dy);
    void print();

private:
    int x;
    int y;
};
```

- Keine Funktionsdefinitionen?

Klassen-Beispiel

```
Point::Point(){ x = 0; y = 0; }
Point::Point(int new_x, int new_y){ x = new_x; y =
    new_y; }

void Point::move(int dx, int dy){
  x = x + dx;
  y = y + dy;
}

void Point::print(){
  cout << "(" << x << ", " << y << ")";
}
```

Klassen-Beispiel

Statt

```
Point::Point(){ x = 0; y = 0; }
Point::Point(int new_x, int new_y){ x = new_x; y =
    new_y; }
```

auch Konstruktoren mit Initialisierungslisten:

```
Point::Point(): x(0), y(0) {}
Point::Point(int new_x, int new_y): x(new_x), y(
    new_y) {}
```

Zugriff auf Daten und Funktionen

- Objekte werden wie Variablen deklariert:

```
Point a(2, 6); // oder Point a;
```

- Daten eines Objekts werden durch den Operator `.` (Punkt) angesprochen:

```
a.move(2,-2);
```

- Daten eines Objekt-Pointers werden durch den Operator `->` (Pfeil) angesprochen:

```
a_pointer->print();
```

- `p->f()` ist Kurzschreibweise für `(*p).f()`

Zugriffsmodifikatoren

- `public`: jede Klasse
- `private`: nur die Klasse selbst
- `protected`: die Klasse selbst und all ihre Kinder/Kindeskinder

Zugriffsmodifikatoren

- `public`: jede Klasse
- `private`: nur die Klasse selbst
- `protected`: die Klasse selbst und all ihre Kinder/Kindeskinder

- Zugriff ist Klassen-basiert!

Zugriffsmodifikatoren

```
class Base{
  public:
    Base(): x(0){};
    void print(){ cout << "x: " << x << endl; };
    void inc(Base& b){ (b.x)++; };
  private:
    int x;
};
```

Zugriffsmodifikatoren

```
class Base{
  public:
    Base(): x(0){};
    void print(){ cout << "x: " << x << endl; };
    void inc(Base& b){ (b.x)++; };
  private:
    int x;
};
```

```
  Base a, b;
  b.x++; // 'int Base::x' is private
  a.inc(b);
  b.print();
```

Zugriffsmodifikatoren

```
class Base{
  public:
    Base(): x(0){};
    void print(){ cout << "x: " << x << endl; };
    void inc(Base& b){ (b.x)++; };
  private:
    int x;
};
```

```
  Base a, b;
  b.x++; // 'int Base::x' is private
  a.inc(b);
  b.print();
```

- `b.print()` → `x: 1`

Verbünde (structs)

- Sammlung von Variablen beliebigen Typs

Verbünde (structs)

- Sammlung von Variablen beliebigen Typs
- Kann Arrays und Pointer auf Verbünde enthalten

Verbünde (structs)

- Sammlung von Variablen beliebigen Typs
- Kann Arrays und Pointer auf Verbünde enthalten
- C++: Kann Methoden enthalten

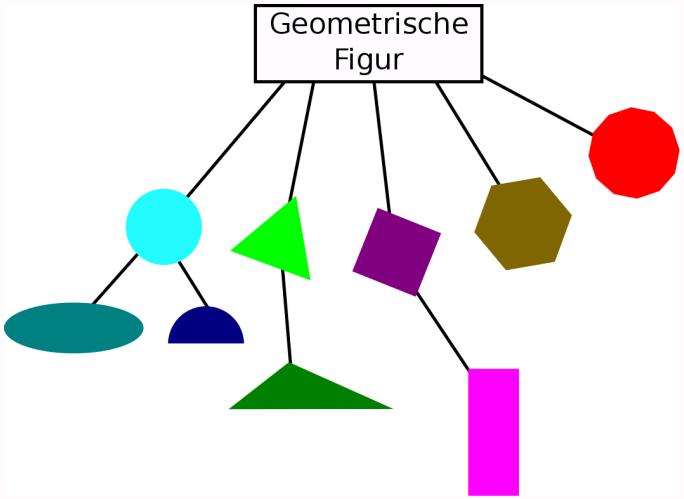
Verbünde (structs)

- Sammlung von Variablen beliebigen Typs
- Kann Arrays und Pointer auf Verbünde enthalten
- C++: Kann Methoden enthalten

$\Rightarrow$ In C++ sind Klassen und Verbünde prinzipiell[2] gleich!

[2] Standard-Zugriffsmodifikator in Klassen ist `private` und in Verbünden `public`

Vererbung



- Vererbung erweitert Klassen
- “base class” vererbt Variablen und Methoden an “derived classes”
- Ähnliche Klassen können gleiche Variablen und Methoden aus einer gemeinsamen Oberklasse beziehen

Vererbungs-Beispiel

```
class Polygon{
protected:
  double w, h;
public:
  void set(double a, double b){
    w = a;
    h = b;
  }
};
```

Vererbungs-Beispiel

```
class Rechteck: public Polygon{
public:
  double flaeche(){
    return w * h;
  }
};

class Dreieck: public Polygon{
public:
  double flaeche(){
    return w * h / 2;
  }
};
```

Vererbungs-Beispiel

```
Dreieck d;
Rechteck r;

d.set(2,4); // set() von Polygon geerbt
r.set(5,3);

cout << "Dreieck: " << d.flaeche() << endl; //
    gibt 4 aus
cout << "Rechteck: " << r.flaeche() << endl; //
    gibt 15 aus
```

Konstruktoren

- Konstruktor einer Klasse wird aufgerufen, wenn ein Objekt initialisiert wird

Konstruktoren

- Konstruktor einer Klasse wird aufgerufen, wenn ein Objekt initialisiert wird
- Bevor der Konstruktor einer “derived class” aufgerufen wird, wird der Konstruktor der “base class” aufgerufen

Konstruktoren

- Konstruktor einer Klasse wird aufgerufen, wenn ein Objekt initialisiert wird
- Bevor der Konstruktor einer “derived class” aufgerufen wird, wird der Konstruktor der “base class” aufgerufen
- Initialisierungslisten können benutzt werden um Argumente an den Konstruktor der “base class” zu übergeben

Konstruktor-Beispiel

```
class Polygon{
protected:
  double w, h;
public:
  Polygon(){ cout << "Polygon-Konstruktor
      ausgeführt." << endl; }
  Polygon(string arg){ cout << "Polygon-Konstruktor
       ausgeführt mit Argument: " << arg << endl; }
};
```

Konstruktor-Beispiel

```
class Rechteck: public Polygon{
public:
  Rechteck(){ cout << "Rechteck-Konstruktor
      ausgeführt." << endl; }
};

class Dreieck: public Polygon{
public:
  Dreieck() : Polygon("Hallo!"){ cout << "Dreieck-
      Konstruktor ausgeführt." << endl; };
};
```

Konstruktor-Beispiel

```
int main(){
  Rechteck r;
  Dreieck d;

  return 0;
}
```

Konstruktor-Beispiel

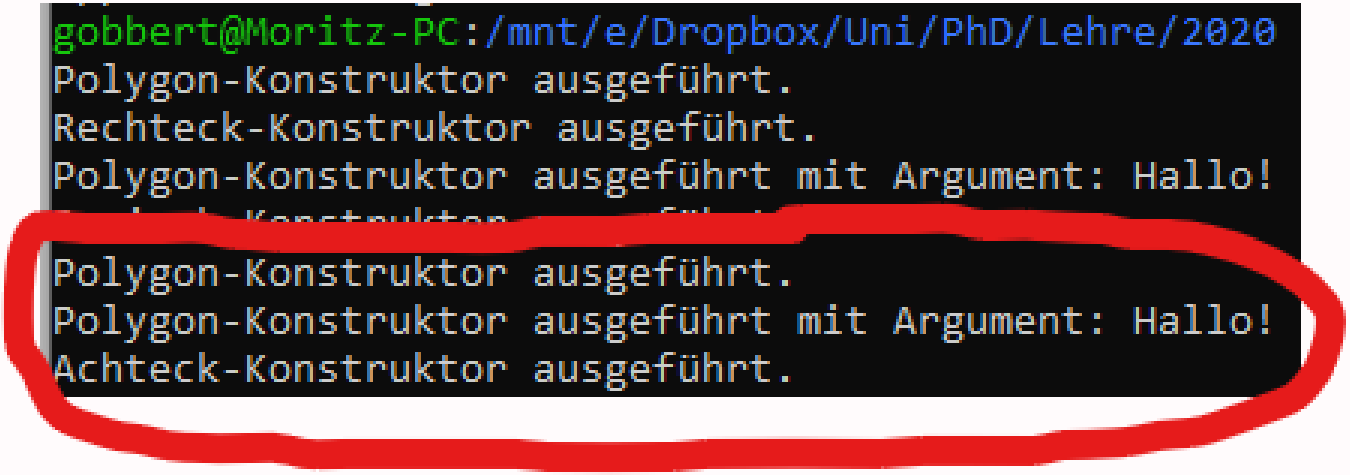
```
gobbert@Moritz-PC:/mnt/e/Dropbox/Uni/PhD/Lehre/2020 (
Polygon-Konstruktor ausgeführt.
Rechteck-Konstruktor ausgeführt.
Polygon-Konstruktor ausgeführt mit Argument: Hallo!
Dreieck-Konstruktor ausgeführt.
gobbert@Moritz-PC:/mnt/e/Dropbox/Uni/PhD/Lehre/2020 (
```

Konstruktor-Beispiel

- Unterschied zu Java:

```
class Achteck: public Polygon{
public:
  Achteck(){
    Polygon("Hallo!");
    cout << "Achteck-Konstruktor ausgeführt." <<
        endl;
  };
};
```

Konstruktor-Beispiel



Befreundete Klassen

[source: Hermann Kern, 1904]

- Befreundete Klassen haben Zugriff auf die privaten Variablen und Methoden ihrer Freunde
- Sowohl ganze Klassen als auch einzelne Methoden können Freunde einer Klasse sein
- Widerspricht dem Prinzip der Datenkapselung

Freundschafts-Beispiel

```
class Box{
private:
  string inhalt;
  int position;
  friend class Regal;
public:
  Box(string in){ inhalt = in; position = 0; }
  void print(){ cout << inhalt << " an Position "
      << position << endl; }
};
```

Freundschafts-Beispiel

```
class Regal{
public:
  void verschiebe_box(Box& b, int pos){
    b.position = pos; }
  void aendere_inhalt(Box& b, string neu){
    b.inhalt = neu; }
};
```

Freundschafts-Beispiel

```
Box kiste("Erdbeeren");
Regal obst;

kiste.print(); // Erdbeeren an Position 0
obst.verschiebe_box(kiste, 7);
obst.aendere_inhalt(kiste, "Orangen");
kiste.print(); // Orangen an Position 7
```

Freundschafts-Beispiel 2

```
class Box{
private:
  string inhalt;
  int position;
  friend void verschiebe_box(Box&, int);
public:
  Box(string in){ inhalt = in; position = 0; }
  void print(){ cout << inhalt << " an Position "
      << position << endl; }
};
```

Freundschafts-Beispiel 2

```
void verschiebe_box(Box& b, int pos){
  b.position = pos;
}
int main(){
  Box kiste("Erdbeeren");

  kiste.print(); // Erdbeeren an Position 0
  verschiebe_box(kiste, 7);
  kiste.print(); // Erdbeeren an Position 7

  return 0;
}
```

Freundschafts-Beispiel 2

```
void aendere_inhalt(Box& b, string neu){
  b.inhalt = neu;
}
```

Freundschafts-Beispiel 2

```
void aendere_inhalt(Box& b, string neu){
  b.inhalt = neu;
}
```

```
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $ g++ freundschaft_methoden.cpp -o freun
dschaft_methoden.o
freundschaft_methoden.cpp: In function ‘void aendere_inhalt(Box&, std::string)’:
freundschaft_methoden.cpp:6:9: error: ‘std::string Box::inhalt’ is private
  string inhalt;
         ^
freundschaft_methoden.cpp:19:4: error: within this context
  b.inhalt = neu;
    ^
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/Kleines Studienprojekt $
```

Header-Dateien

- Aufteilung von Klassen-Definition und -Implementierung
- Definition in Header-Datei (`*.h`)
- Implementierung in üblicher “implementation”-Datei (`*.cpp`)

Header-Dateien

- Aufteilung von Klassen-Definition und -Implementierung
- Definition in Header-Datei (`*.h`)
- Implementierung in üblicher “implementation”-Datei (`*.cpp`)
- Einbinden der Header-Datei mit Schlüsselwort `#include`

Header-Beispiel

katze.h

```
#include <iostream>

class Katze{
  private:
    std::string name; // Name der Katze
    int energie; // Zahl zwischen 0 und 60
  public:
    Katze(std::string); // Konstruktor
    void schlafen(int); // erhoeht energie
    void spielen(int); // verringert energie
    std::string status(); // beschreibt energie
};
```

Header-Beispiel

katze.cpp

```
#include "katze.h"
using namespace std;

Katze::Katze(string n = "Kitty"){
  name = n;
  energie = 10;
}

void Katze::schlafen(int dauer){
  // ...
}
// ...
```

Header-Beispiel

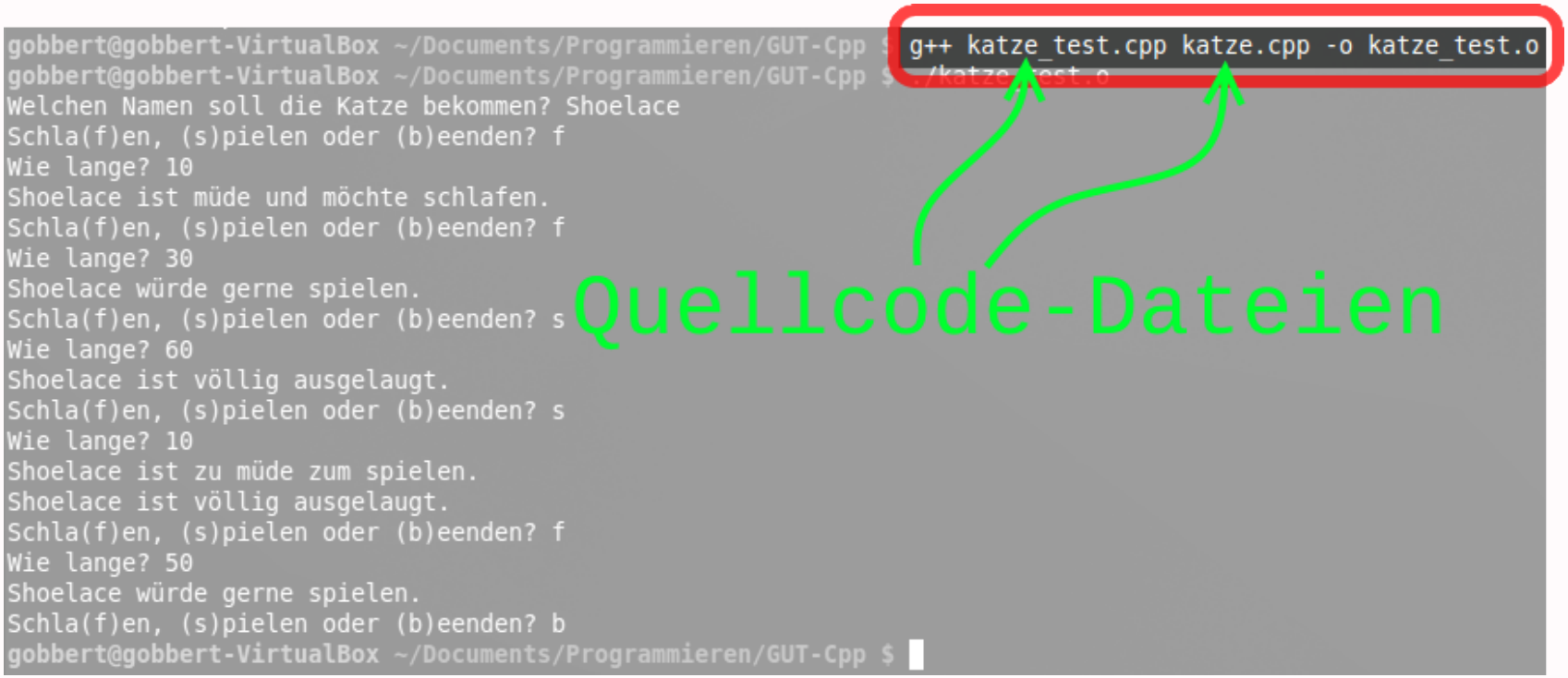
katze_test.cpp

```cpp
#include "katze.h"
using namespace std;

int main(){
  // Variableninitialisierung
  cout << "Wie heisst die Katze? ";
  string name;
  cin >> name;
  Katze k(name);
  string input;
  int dauer;
  // ...
}
```

Header-Beispiel

```
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/GUT-Cpp $ g++ katze_test.cpp katze.cpp -o katze_test.o
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/GUT-Cpp $ ./katze_test.o
Welchen Namen soll die Katze bekommen? Shoelace
Schla(f)en, (s)pielen oder (b)eenden? f
Wie lange? 10
Shoelace ist müde und möchte schlafen.
Schla(f)en, (s)pielen oder (b)eenden? f
Wie lange? 30
Shoelace würde gerne spielen.
Schla(f)en, (s)pielen oder (b)eenden? s
Wie lange? 60
Shoelace ist völlig ausgelaugt.
Schla(f)en, (s)pielen oder (b)eenden? s
Wie lange? 10
Shoelace ist zu müde zum spielen.
Shoelace ist völlig ausgelaugt.
Schla(f)en, (s)pielen oder (b)eenden? f
Wie lange? 50
Shoelace würde gerne spielen.
Schla(f)en, (s)pielen oder (b)eenden? b
gobbert@gobbert-VirtualBox ~/Documents/Programmieren/GUT-Cpp $
```

Header-Beispiel



Include Guard

grandfather.h

```
class foo {
    int member;
};
```

father.h

```
#include "grandfather.h"
```

child.cpp

```
#include "grandfather.h"
#include "father.h"
```

Include Guard

grandfather.h

```
class foo {
    int member;
};
```

father.h

```
#include "grandfather.h"
```

child.cpp

```
#include "grandfather.h"
#include "father.h"
```

$\Rightarrow$ verletzt “one definition”-Regel (`foo` doppelt definiert)

Include Guard (fort.)

`grandfather.h` (`father.h` und `child.cpp` unverändert)

```
#ifndef GRANDFATHER_H
#define GRANDFATHER_H

class foo {
    int member;
};

#endif
```

Include Guard (fort.)

`grandfather.h` (`father.h` und `child.cpp` unverändert)

```
#ifndef GRANDFATHER_H
#define GRANDFATHER_H

class foo {
    int member;
};

#endif
```

- Erstes `#include ''grandfather.h''` setzt `GRANDFATHER_H`; zweites `#include ''grandfather.h''` tut nichts

Operator-Überladung

Allgemein

Motivation: Klassen für Benutzer intuitiver gestalten, bspw. “+” für eigene Klassen ermöglichen

Allgemein

Motivation: Klassen für Benutzer intuitiver gestalten, bspw. "`+`" für eigene Klassen ermöglichen

- Fast alle Operatoren können Überladen werden. Ausnahmen: `::` (Scope), `?:` (tern. conditional), `.` (Klassenzugriff) und `sizeof`

Allgemein

Motivation: Klassen für Benutzer intuitiver gestalten, bspw. "`+`" für eigene Klassen ermöglichen

- Fast alle Operatoren können Überladen werden. Ausnahmen: `::` (Scope), `?:` (tern. conditional), `.` (Klassenzugriff) und `sizeof`
- Es können keine neuen Operatoren (bspw. `**`) erstellt werden

Allgemein

Motivation: Klassen für Benutzer intuitiver gestalten, bspw. "`+`" für eigene Klassen ermöglichen

- Fast alle Operatoren können Überladen werden. Ausnahmen: `::` (Scope), `?:` (tern. conditional), `.` (Klassenzugriff) und `sizeof`
- Es können keine neuen Operatoren (bspw. `**`) erstellt werden
- Operatorrangfolge ("precedence") oder Stelligkeit der Operatoren kann nicht geändert werden

Allgemein

Motivation: Klassen für Benutzer intuitiver gestalten, bspw. "`+`" für eigene Klassen ermöglichen

- Fast alle Operatoren können Überladen werden. Ausnahmen: `::` (Scope), `?:` (tern. conditional), `.` (Klassenzugriff) und `sizeof`
- Es können keine neuen Operatoren (bspw. `**`) erstellt werden
- Operatorrangfolge ("precedence") oder Stelligkeit der Operatoren kann nicht geändert werden
- Funktionen können innerhalb oder außerhalb der zugeh. Klasse definiert werden

Allgemein

Motivation: Klassen für Benutzer intuitiver gestalten, bspw. "`+`" für eigene Klassen ermöglichen

- Fast alle Operatoren können Überladen werden. Ausnahmen: `::` (Scope), `?:` (tern. conditional), `.` (Klassenzugriff) und `sizeof`
- Es können keine neuen Operatoren (bspw. `**`) erstellt werden
- Operatorrangfolge ("precedence") oder Stelligkeit der Operatoren kann nicht geändert werden
- Funktionen können innerhalb oder außerhalb der zugeh. Klasse definiert werden
- (Konventionen/Eigenheiten versch. Operatoren bei Gelegenheit nachlesen...)

Beispiel

`vector_2d` ist ein zweidimensionaler Vektor (bestehend aus x– und y–Komponente). Die Funktionen sind innerhalb der Klasse definiert.

```
vector_2d operator+(const vector_2d &other){
  return vector_2d(x + other.x, y + other.y);
}
// Skalarprodukt
int operator*(const vector_2d &other){
  return (x*other.x + y*other.y);
}
```

Beispiel

`vector_2d` ist ein zweidimensionaler Vektor (bestehend aus x– und y–Komponente). Die Funktionen sind innerhalb der Klasse definiert.

```
    bool operator==(const vector_2d &other){
      return (x == other.x && y == other.y);
    }
```

Beispiel

`vector_2d` ist ein zweidimensionaler Vektor (bestehend aus x– und y–Komponente).

```
ostream& operator<<(ostream& os, const vector_2d& v
    ){
  os << "(" << v.x << "," << v.y << ")";
  return os;
}
```

Beispiel

```
vector_2d v1(1,2); vector_2d v2(2,1); vector_2d
    v3(3,3);

vector_2d v_sum = v1+v2;
int skalar = v1*v2;
cout << "v1+v2: " << v_sum << endl;
cout << "v1*v2: " << skalar << endl;
cout << "v1 == v2: " << (v1==v2) << endl;
cout << "v3 == v_sum: " << (v3==v_sum) << endl;
```

Beispiel

```
gobbert@Moritz-PC:/mnt/e/Dropbox/Uni/PhD/Lehre/2020 (Somm
v1: (1,2), v2: (2,1), v3: (3,3), v_sum: (3,3)
v1+v2: (3,3)
v1*v2: 4
v1 == v2: 0
v3 == v_sum: 1
gobbert@Moritz-PC:/mnt/e/Dropbox/Uni/PhD/Lehre/2020 (Somm
```

Generisches Programmieren

Templates

Motivation: Gleichartige Funktionen (oder Klassen), die sich im Typ unterscheiden, in einer Definition zusammenfassen

```
void tausche(int* a, int* b){
  int t = *a; *a = *b; *b = t;
}
void tausche(double* a, double* b){
  double t = *a; *a = *b; *b = t;
}
void tausche(string* a, string* b){
  string t = *a; *a = *b; *b = t;
}
```

Templates

Motivation: Gleichartige Funktionen (oder Klassen), die sich im Typ unterscheiden, in einer Definition zusammenfassen

```
template <typename T>
void tausche(T* a, T* b){
  T t = *a; *a = *b; *b = t;
}
```

- Templates sind Schablonen für Funktions- und Klassendefinitionen

Funktions-Templates im Compiler

Wenn im Programm die Funktion aufgerufen wird, definiert der Compiler automatisch die zum Typ passende Funktion:

```
double x, y;
tausche(x, y);
```

definiert die Funktion

```
void tausche(double* a, double* b){ /* ... */ }
```

Klassen-Templates

- Schablone für Klassen
- Formale Parameter sind Platzhalter für die (in die Schablone einzusetzenden) Daten
- Formale Parameter werden bei Generierung durch konkrete Daten ersetzt

Klassen-Templates (Beispiel)

```
template <typename E>
class Point{
public:
    Point(E new_x, E new_y);
    void move(E dx, E dy);
    void print();

private:
    E x;
    E y;
};
```

Klassen-Templates (Beispiel)

```
template <typename E>
Point<E>::Point(E new_x, E new_y){ x = new_x; y =
    new_y; }

template <typename E>
void Point<E>::move(E dx, E dy){
  x = x + dx;
  y = y + dy;
}
template <typename E>
void Point<E>::print(){
  cout << "(" << x << ", " << y << ")";
}
```

Klassen-Templates (Beispiel)

```
Point<double> a(1.2, 0.7);
a.move(0.4, 1.2);
a.print(); // a = (1.6, 1.9)

Point<string> b("ab", "cd");
b.move("xy", "xz");
b.print(); // b = (abxy, cdxz)

Point<bool> c(false, false);
c.move(true, false);
c.print(); // c = (true, false)
```

C++ für Java-Programmierer