

# Grundlagenschulung C/C++

#5 Hello, World of C++

---

Thorsten Gedicke

07.03.2022

Elektronische Fahrwerksysteme GmbH (EFS)



# Hello, World of C++

---

Erinnerung: Unser aus C bekanntes

Hello-World Beispiel: hello.c

```
1 #include <stdio.h>
2
3 int main(void) {
4     puts("Hello, World!");
5     return 0;
6 }
```

Erinnerung: Unser aus C bekanntes  
Hello-World Beispiel: hello.c

```
1 #include <stdio.h>
2
3 int main(void) {
4     puts("Hello, World!");
5     return 0;
6 }
```

Um unseren Code statt als C als C++ zu  
bauen, ändern wir den Compileraufruf:

Vorher (C): gcc -std=c89

Jetzt (C++): g++ -std=c++98

Unser HelloWorld-Code aus dem C-Teil  
lässt sich problemlos als C++ bauen.

Erinnerung: Unser aus C bekanntes  
Hello-World Beispiel: hello.c

```
1 #include <stdio.h>
2
3 int main(void) {
4     puts("Hello, World!");
5     return 0;
6 }
```

Um unseren Code statt als C als C++ zu  
bauen, ändern wir den Compileraufruf:

Vorher (C): gcc -std=c89

Jetzt (C++): g++ -std=c++98

Unser HelloWorld-Code aus dem C-Teil  
lässt sich problemlos als C++ bauen.

```
> g++ -std=c++98 -pedantic -Wall -o hello hello.c
> hello
Hello, World!
```

Natürlich ist unser Code trotzdem noch kein „echter“ C++ Code. Wenn wir C++ Code schreiben, sollten wir zunächst die Dateiendungen anpassen:

Natürlich ist unser Code trotzdem noch kein „echter“ C++ Code. Wenn wir C++ Code schreiben, sollten wir zunächst die Dateiendungen anpassen:

*Implementation:* **.cpp**, .cc, .cxx, .C

Natürlich ist unser Code trotzdem noch kein „echter“ C++ Code. Wenn wir C++ Code schreiben, sollten wir zunächst die Dateiendungen anpassen:

*Implementation:* **.cpp**, .cc, .cxx, .C

*Header:* **.h**, .hpp, .hh, .hxx



Natürlich ist unser Code trotzdem noch kein „echter“ C++ Code. Wenn wir C++ Code schreiben, sollten wir zunächst die Dateiendungen anpassen:

*Implementation:* **.cpp**, .cc, .cxx, .C

*Header:* **.h**, .hpp, .hh, .hxx

Also: main.c → main.cpp

# Leere Parameterliste

Für C++ bedeutet eine leere Parameterliste, dass die Funktion keine Argumente entgegennimmt. Wir verzichten in diesen Fällen also nun auf `void`.

Vorher: `int main(void)`

Jetzt: `int main()`

# Leere Parameterliste

Für C++ bedeutet eine leere Parameterliste, dass die Funktion keine Argumente entgegennimmt. Wir verzichten in diesen Fällen also nun auf `void`.

Vorher: `int main(void)`

Jetzt: `int main()`

Inhalt von `hello.cpp`:

```
1 #include <stdio.h>
2
3 int main() {
4     puts("Hello, World!");
5     return 0;
6 }
```

# Leere Parameterliste

Für C++ bedeutet eine leere Parameterliste, dass die Funktion keine Argumente entgegennimmt. Wir verzichten in diesen Fällen also nun auf `void`.

Vorher: `int main(void)`

Jetzt: `int main()`

Inhalt von `hello.cpp`:

```
1 #include <stdio.h>
2
3 int main() {
4     puts("Hello, World!");
5     return 0;
6 }
```

```
> g++ -std=c++98 -pedantic -Wall -o hello hello.cpp
> hello
Hello, World!
```

# Namespaces

---

Bezeichner von Datentypen und Funktionen müssen eindeutig sein, da es sonst zu einer Namenskollision kommt (Einnerung: *One-Definition-Rule, ODR*).

Bezeichner von Datentypen und Funktionen müssen eindeutig sein, da es sonst zu einer Namenskollision kommt (Einnerung: *One-Definition-Rule, ODR*).

In C kann das besonders dann problematisch werden, wenn Code von verschiedenen Quellen kombiniert wird. Präventiv kann das nur durch die Wahl langer, unansehnlicher Namen gelöst werden: `struct Punkt` → `struct EFSVorentwicklungspunkt2D`.

Bezeichner von Datentypen und Funktionen müssen eindeutig sein, da es sonst zu einer Namenskollision kommt (Einnerung: *One-Definition-Rule, ODR*).

In C kann das besonders dann problematisch werden, wenn Code von verschiedenen Quellen kombiniert wird. Präventiv kann das nur durch die Wahl langer, unansehnlicher Namen gelöst werden: `struct Punkt` → `struct EFSVorentwicklungspunkt2D`.

C++ entschärft dieses Problem durch *Namespaces*:

- Namen gehören zu Namespaces (wie Dateien zu Verzeichnissen).



Bezeichner von Datentypen und Funktionen müssen eindeutig sein, da es sonst zu einer Namenskollision kommt (Einnerung: *One-Definition-Rule, ODR*).

In C kann das besonders dann problematisch werden, wenn Code von verschiedenen Quellen kombiniert wird. Präventiv kann das nur durch die Wahl langer, unansehnlicher Namen gelöst werden: `struct Punkt` → `struct EFSVorentwicklungspunkt2D`.

C++ entschärft dieses Problem durch *Namespaces*:

- Namen gehören zu Namespaces (wie Dateien zu Verzeichnissen).
- Auflösung von Namen relativ zum aktuellen Namespace (wie bei Dateipfaden).

Bezeichner von Datentypen und Funktionen müssen eindeutig sein, da es sonst zu einer Namenskollision kommt (Einnerung: *One-Definition-Rule, ODR*).

In C kann das besonders dann problematisch werden, wenn Code von verschiedenen Quellen kombiniert wird. Präventiv kann das nur durch die Wahl langer, unansehnlicher Namen gelöst werden: `struct Punkt` → `struct EFSVorentwicklungspunkt2D`.

C++ entschärft dieses Problem durch *Namespaces*:

- Namen gehören zu Namespaces (wie Dateien zu Verzeichnissen).
- Auflösung von Namen relativ zum aktuellen Namespace (wie bei Dateipfaden).
- Namensangabe ist relativ oder absolut möglich (wie bei Dateipfaden).

Bezeichner von Datentypen und Funktionen müssen eindeutig sein, da es sonst zu einer Namenskollision kommt (Einnerung: *One-Definition-Rule, ODR*).

In C kann das besonders dann problematisch werden, wenn Code von verschiedenen Quellen kombiniert wird. Präventiv kann das nur durch die Wahl langer, unansehnlicher Namen gelöst werden: `struct Punkt` → `struct EFSVorentwicklungsPunkt2D`.

C++ entschärft dieses Problem durch *Namespaces*:

- Namen gehören zu Namespaces (wie Dateien zu Verzeichnissen).
- Auflösung von Namen relativ zum aktuellen Namespace (wie bei Dateipfaden).
- Namensangabe ist relativ oder absolut möglich (wie bei Dateipfaden).
- Trennzeichen ist `::` (vgl. Pfadtrennzeichen `\` für Windows und `/` für Unix)

Wir *öffnen* einen Namespace mit der  
Syntax `namespace xyz { }`,  
wobei xyz ein beliebiger Name ist.

Wir *öffnen* einen Namespace mit der  
Syntax `namespace xyz { }`,  
wobei xyz ein beliebiger Name ist.

Innerhalb der `{ }` deklarieren/definieren  
wir Funktionen, Typen und Objekte wie  
gewohnt. Diese sind auch füreinander  
sichtbar.

Wir *öffnen* einen Namespace mit der Syntax `namespace xyz { }`, wobei xyz ein beliebiger Name ist.

Innerhalb der `{ }` deklarieren/definieren wir Funktionen, Typen und Objekte wie gewohnt. Diese sind auch füreinander sichtbar.

Außerhalb des Namespace müssen wir mit `::` „hinein navigieren“.

Wir *öffnen* einen Namespace mit der Syntax `namespace xyz { }`, wobei xyz ein beliebiger Name ist.

Innerhalb der `{ }` deklarieren/definieren wir Funktionen, Typen und Objekte wie gewohnt. Diese sind auch füreinander sichtbar.

Außerhalb des Namespace müssen wir mit `::` „hinein navigieren“.

```
1 #include <stdio.h>
2
3 namespace schulung {
4
5 void foo() {
6     puts("foo");
7 }
8
9 void foobar() {
10    foo();
11    puts("bar");
12 }
13
14 } // Ende Namespace "schulung"
15
16 int main() {
17     schulung::foobar();
18     return 0;
19 }
```

Namespaces können wie Verzeichnisse geschachtelt werden.



Namespaces können wie Verzeichnisse geschachtelt werden.

Namen werden auf der gleichen Ebene *oder auf höheren Ebenen* gesucht.

Namespaces können wie Verzeichnisse geschachtelt werden.

Namen werden auf der gleichen Ebene *oder auf höheren Ebenen* gesucht.

Die Funktion puts und alles, was wir aus C kennen, liegen auf der höchsten Ebene, dem *globalen Namespace*.

Namespaces können wie Verzeichnisse geschachtelt werden.

Namen werden auf der gleichen Ebene *oder auf höheren Ebenen* gesucht.

Die Funktion puts und alles, was wir aus C kennen, liegen auf der höchsten Ebene, dem *globalen Namespace*.

```
1 #include <stdio.h>
2
3 namespace schulung {
4
5     void foo() {
6         puts("foo");
7     }
8
9     namespace deeper {
10
11         void foobar() {
12             foo();
13             puts("bar");
14         }
15
16     } // Ende Namespace "deeper"
17
18     void baz() {
19         deeper::foobar();
20     }
21
22 } // Ende Namespace "schulung"
23
24 int main() {
25     schulung::baz();
26     schulung::deeper::foobar();
27     return 0;
28 }
```

Durch Vorangestelltes :: kann ein Name absolut angegeben und auf den globalen Namespace verwiesen werden. Das ist relativ selten nötig.

Durch Vorangestelltes `::` kann ein Name absolut angegeben und auf den globalen Namespace verwiesen werden. Das ist relativ selten nötig.

```
1 #include <stdio.h>
2
3 namespace schulung {
4
5 void foo() {
6     ::puts("foo");
7 }
8
9 namespace deeper {
10
11 void foobar() {
12     ::schulung::foo();
13     ::puts("bar");
14 }
15
16 } // Ende Namespace "deeper"
17
18 void baz() {
19     ::schulung::deeper::foobar();
20 }
21
22 } // Ende Namespace "schulung"
23
24 int main() {
25     ::schulung::baz();
26     ::schulung::deeper::foobar();
27     return 0;
28 }
```

Namespaces verhindern Kollisionen und erlauben eindeutige Bezeichnung, auch wenn Elemente lokal gleich heißen.

**Beispiel:**

Zwei mal `foo`, kein Problem.

Namespaces verhindern Kollisionen und erlauben eindeutige Bezeichnung, auch wenn Elemente lokal gleich heißen.

### Beispiel:

Zwei mal foo, kein Problem.

```
1  #include <stdio.h>
2
3  namespace schulung {
4
5  void foo() {
6      puts("foo");
7  }
8
9  namespace deeper {
10
11  void foo() {
12      schulung::foo();
13      puts("bar");
14  }
15
16  } // Ende Namespace "deeper"
17
18  void baz() {
19      foo(); // ::schulung::foo
20      deeper::foo();
21  }
22
23  } // Ende Namespace "schulung"
24
25  int main() {
26      schulung::baz();
27      return 0;
28  }
```

In Headern deklarieren wir Funktionen innerhalb von Namespaces.  
Für die dazugehörige Definition können wir den Namen auflösen:



In Headern deklarieren wir Funktionen innerhalb von Namespaces.  
Für die dazugehörige Definition können wir den Namen auflösen:

```
1  /*** foobar.h ***/
2
3  #ifndef SCHULUNG_FOOBAR_H
4  #define SCHULUNG_FOOBAR_H
5
6  namespace schulung {
7
8  void foo();
9  void baz();
10
11 namespace deeper {
12
13 void foobar();
14
15 } // Ende Namespace "deeper"
16 } // Ende Namespace "schulung"
17
18 #endif
```

In Headern deklarieren wir Funktionen innerhalb von Namespaces.  
Für die dazugehörige Definition können wir den Namen auflösen:

```
1  /*** foobar.h ***/
2
3  #ifndef SCHULUNG_FOOBAR_H
4  #define SCHULUNG_FOOBAR_H
5
6  namespace schulung {
7
8  void foo();
9  void baz();
10
11 namespace deeper {
12
13 void foobar();
14
15 } // Ende Namespace "deeper"
16 } // Ende Namespace "schulung"
17
18 #endif
```

```
1  /*** foobar.cpp ***/
2
3  #include "foobar.h"
4  #include <stdio.h>
5
6  void schulung::foo() {
7      puts("foo");
8  }
9
10 void schulung::baz() {
11     deeper::foobar();
12 }
13
14 void schulung::deeper::foobar() {
15     foo();
16     puts("bar");
17 }
```

In Headern deklarieren wir Funktionen innerhalb von Namespaces.  
Für die dazugehörige Definition können wir den Namen auflösen:

```
1  /*** foobar.h ***/
2
3  #ifndef SCHULUNG_FOOBAR_H
4  #define SCHULUNG_FOOBAR_H
5
6  namespace schulung {
7
8  void foo();
9  void baz();
10
11 namespace deeper {
12
13 void foobar();
14
15 } // Ende Namespace "deeper"
16 } // Ende Namespace "schulung"
17
18 #endif
```

```
1  /*** foobar.cpp ***/
2
3  #include "foobar.h"
4  #include <stdio.h>
5
6  void schulung::foo() {
7      puts("foo");
8  }
9
10 void schulung::baz() {
11     deeper::foobar();
12 }
13
14 void schulung::deeper::foobar() {
15     foo();
16     puts("bar");
17 }
```

```
1  /*** main.cpp ***/
2
3  #include "foobar.h"
4
5  int main() {
6      schulung::baz();
7      schulung::deeper::foobar();
8      return 0;
9  }
```

## Regel:

Es gibt so gut wie nie einen Grund, den globalen Namespace zu „verschmutzen“.  
Wir sollten stets in einem sinnvoll gewählten Namespace agieren:

- Projektname
- Logische Projektstruktur

## Regel:

Es gibt so gut wie nie einen Grund, den globalen Namespace zu „verschmutzen“.

Wir sollten stets in einem sinnvoll gewählten Namespace agieren:

- Projektname
- Logische Projektstruktur

## Ausnahme:

In den Beispielen der Schulungsfolien werden wir, um den Code kompakt zu halten, im globalen Namespace bleiben.

# Header der C++ Standardbibliothek

---

Header der C++ Standardbibliothek verzichten auf eine Dateiendung. Alle Elemente, die von C++ Standardheadern bereitgestellt werden, befinden sich im Namespace `std`.

Header der C++ Standardbibliothek verzichten auf eine Dateiendung. Alle Elemente, die von C++ Standardheadern bereitgestellt werden, befinden sich im Namespace `std`.

C++ übernimmt die Header der C Standardbibliothek nach folgendem Muster:

`<xyz.h> → <xyz>`



Header der C++ Standardbibliothek verzichten auf eine Dateiendung. Alle Elemente, die von C++ Standardheadern bereitgestellt werden, befinden sich im Namespace std.

C++ übernimmt die Header der C Standardbibliothek nach folgendem Muster:

`<xyz.h> → <cxyz>`

Vorher: `#include<stdio.h> ⇒ puts, printf`

Jetzt: `#include<cstdio> ⇒ std::puts, std::printf`

Header der C++ Standardbibliothek verzichten auf eine Dateierendung. Alle Elemente, die von C++ Standardheadern bereitgestellt werden, befinden sich im Namespace std.

C++ übernimmt die Header der C Standardbibliothek nach folgendem Muster:

`<xyz.h> → <cxyz>`

Vorher: `#include<stdio.h> ⇒ puts, printf`

Jetzt: `#include<cstdio> ⇒ std::puts, std::printf`

**Regel:** Die Header der Form `<xyz.h>` werden von C++ nur zur C-Kompatibilität unterstützt und sollten von C++ Code nicht verwendet werden.

Unser Hello-World-Programm sieht damit nun so aus:

```
1 #include <stdio>
2
3 int main() {
4     std::puts("Hello, World!");
5     return 0;
6 }
```

Unser Hello-World-Programm sieht damit nun so aus:

```
1 #include <cstdio>
2
3 int main() {
4     std::puts("Hello, World!");
5     return 0;
6 }
```

Allerdings ist die Ausgabe mittels puts oder printf immer noch C-Stil und kein typisches C++. Um die bevorzugte Variante für C++ zu verstehen, müssen wir mit neuen Konzepten etwas ausholen...

## struct, union und enum in C++

---

Erinnerung: In C mussten wir die Definition eines `struct/union/enum` XYZ stets mit `typedef` kombinieren, wenn wir XYZ alleine als Name des Datentyps nutzen wollten:

1		<code>struct</code>		1		<code>typedef struct</code>	
2		Point		2		Point	
3		{		3		{	
4		double		4		double	
5		x;		5		x;	
6		double		6		y;	
7		y;		7		} Point	
8		};		8		;	
9		// Nutzbar als		9		// Nutzbar als	
10		struct		10		Point	
11		Point		11		p;	
12		p;					

vs.

Erinnerung: In C mussten wir die Definition eines `struct/union/enum` XYZ stets mit `typedef` kombinieren, wenn wir XYZ alleine als Name des Datentyps nutzen wollten:

1	<code>struct</code>		1	<code>typedef struct</code>	<code>Point</code>	{
2	<code>double</code>		2	<code>double</code>	<code>x</code> ;	
3	<code>double</code>		3	<code>double</code>	<code>y</code> ;	
4	};		4	}	<code>Point</code>	;
5	<code>// Nutzbar als</code>		5	<code>// Nutzbar als</code>		
6	<code>struct</code>		6	<code>Point</code>	<code>p</code> ;	

vs.

In C++ ist das `typedef struct` Idiom überflüssig. Wir schreiben einfach:

```
1 struct Point {
2     double x;
3     double y;
4 };
5 // Nutzbar als
6 Point p;
```

Erinnerung: In C mussten wir die Definition eines `struct/union/enum` XYZ stets mit `typedef` kombinieren, wenn wir XYZ alleine als Name des Datentyps nutzen wollten:

1	<code>struct</code>		1	<code>typedef struct</code>	<code>Point</code>	{
2	<code>double</code>		2	<code>double</code>	<code>x</code> ;	
3	<code>double</code>		3	<code>double</code>	<code>y</code> ;	
4	};		4	}	<code>Point</code>	;
5	<code>// Nutzbar als</code>		5	<code>// Nutzbar als</code>		
6	<code>struct</code>		6	<code>Point</code>	<code>p</code> ;	

vs.

In C++ ist das `typedef struct` Idiom überflüssig. Wir schreiben einfach:

```
1 struct Point {
2     double x;
3     double y;
4 };
5 // Nutzbar als
6 Point p;
```

💡 Gilt genau so für `union` und `enum`



## Referenzen

---

Erinnerung: In C können wir Verweise auf Variablen als Pointer anlegen. Ein Pointer  $p$  symbolisiert selber nicht den Wert, auf den er zeigt, sondern dessen Speicheradresse.

- $p = \&x$  setzt Zeiger neu, statt referenzierten Wert zu verändern
- $p + 1$  erzeugt verschobenen Zeiger, statt mit referenziertem Wert zu addieren.

Erinnerung: In C können wir Verweise auf Variablen als Pointer anlegen. Ein Pointer  $p$  symbolisiert selber nicht den Wert, auf den er zeigt, sondern dessen Speicheradresse.

- $p = \&x$  setzt Zeiger neu, statt referenzierten Wert zu verändern
- $p + 1$  erzeugt verschobenen Zeiger, statt mit referenziertem Wert zu addieren.

Der dereferenzierte Pointer ( $*p$ ) steht stellvertretend für den Wert, auf den er zeigt:

- $*p = x$  verändert Wert, auf den  $p$  zeigt.
- $*p + 1$  addiert 1 auf den Wert, auf den  $p$  zeigt.

Erinnerung: In C können wir Verweise auf Variablen als Pointer anlegen. Ein Pointer  $p$  symbolisiert selber nicht den Wert, auf den er zeigt, sondern dessen Speicheradresse.

- $p = \&x$  setzt Zeiger neu, statt referenzierten Wert zu verändern
- $p + 1$  erzeugt verschobenen Zeiger, statt mit referenziertem Wert zu addieren.

Der dereferenzierte Pointer ( $*p$ ) steht stellvertretend für den Wert, auf den er zeigt:

- $*p = x$  verändert Wert, auf den  $p$  zeigt.
- $*p + 1$  addiert 1 auf den Wert, auf den  $p$  zeigt.

Beobachtung: Manchmal, z.B. für Call-by-Reference Parameter, brauchen wir eigentlich nur einen „Stellvertreter“ (■), aber keinen verstellbaren Zeiger (●) ...

In manchen Fällen wollen wir nur anzeigen, dass wir mit einer bestehenden Variable arbeiten wollen, statt eine Kopie zu machen, z.B. bei Funktionsparametern. C++ bietet dafür zusätzlich zu Pointern *Referenzen* an. Referenzen werden wie folgt angelegt:

In manchen Fällen wollen wir nur anzeigen, dass wir mit einer bestehenden Variable arbeiten wollen, statt eine Kopie zu machen, z.B. bei Funktionsparametern. C++ bietet dafür zusätzlich zu Pointern *Referenzen* an. Referenzen werden wie folgt angelegt:

```
1 | int tony_stark = 7;
2 | int& iron_man = tony_stark; // iron_man hat Typ int&, lese "Referenz auf int"
3 | iron_man = iron_man + 1;    // int& ist wie int-Variable nutzbar
4 | printf("%d", tony_stark);  // "8", tony_stark und iron_man sind identisch
```

In manchen Fällen wollen wir nur anzeigen, dass wir mit einer bestehenden Variable arbeiten wollen, statt eine Kopie zu machen, z.B. bei Funktionsparametern. C++ bietet dafür zusätzlich zu Pointern *Referenzen* an. Referenzen werden wie folgt angelegt:

```
1 int tony_stark = 7;
2 int& iron_man = tony_stark; // iron_man hat Typ int&, lese "Referenz auf int"
3 iron_man = iron_man + 1;    // int& ist wie int-Variable nutzbar
4 printf("%d", tony_stark);   // "8", tony_stark und iron_man sind identisch
```

- Referenzen müssen mit einem Ziel initialisiert werden.
- Einmal gesetzte Referenzen können nicht „umgesetzt“ werden.
- Referenzen können nicht „auf nichts“ zeigen (analog zu NULL).

Const-Referenzen forcieren Read-Only Zugriff (wie Pointer-auf-Const).

```
1 | int foo = 7;  
2 | const int& bar = foo;  
3 | foo = 42;           // OK, foo immer noch schreibbar  
4 | printf("%d", bar); // "42"  
5 | bar = 23;           // FEHLER: Zugriffe mit bar sind read-only.
```



Const-Referenzen forcieren Read-Only Zugriff (wie Pointer-auf-Const).

```
1 | int foo = 7;  
2 | const int& bar = foo;  
3 | foo = 42;           // OK, foo immer noch schreibbar  
4 | printf("%d", bar); // "42"  
5 | bar = 23;           // FEHLER: Zugriffe mit bar sind read-only.
```

Indem wir Funktions-Parameter nicht als Pointer, sondern als Referenzen deklarieren, erreichen wir Call-by-Reference Semantik ohne Umweg über die Speicheradresse. Beim Aufruf der Funktion entfällt dann der Address-of Operator (&x). Vergleiche bekannte Lösung mit Pointer und neue Lösung mit Referenzen...

Erinnerung (Teil 4):

Dieses **C Programm** nutzt Pointer um Call-by-Reference umzusetzen.

```
1  /** coords.c */
2  #include "coords.h"
3  #include <stdio.h>
4  #include <math.h>
5
6  void print_coords(const Coords* v) {
7      printf("Coords (%.2f, %.2f)\n", v->x, v->y);
8  }
9
10 Coords make_unit(const Coords* v) {
11     double l = sqrt(pow(v->x, 2) + pow(v->y, 2));
12     Coords out;
13     out.x = v->x / l;
14     out.y = v->y / l;
15     return out;
16 }
```

```
1  /** coords.h */
2  #ifndef COORDS_H
3  #define COORDS_H
4
5  typedef struct Coords {
6      double x;
7      double y;
8  } Coords;
9
10 void print_coords(const Coords* v);
11 Coords make_unit(const Coords* v);
12 #endif
```

```
1  /** coord_demo.c */
2
3  #include "coords.h"
4
5  int main(void)
6  {
7      Coords a = {1.2, 2.4};
8      Coords b = make_unit(&a);
9      print_coords(&a); // (1.20, 2.40)
10     print_coords(&b); // (0.45, 0.89)
11     return 0;
12 }
```

In der **C++ Variante** nutzen wir  
Referenzen: Pointer-Dereferenzierung (->)  
und Address-of Operator (&) entfallen.

```
1  /** coords.cpp */
2  #include "coords.h"
3  #include <cstdio>
4  #include <cmath>
5
6  void print(const Coords& v) {
7      std::printf("Coords (%.2f, %.2f)\n", v.x, v.y);
8  }
9
10 Coords make_unit(const Coords& v) {
11     double l = std::sqrt(v.x*v.x + v.y*v.y);
12     Coords out;
13     out.x = v.x / l;
14     out.y = v.y / l;
15     return out;
16 }
```

```
1  /** coords.h */
2  #ifndef COORDS_H
3  #define COORDS_H
4
5  struct Coords {
6      double x;
7      double y;
8  };
9
10 void print(const Coords& v);
11 Coords make_unit(const Coords& v);
12 #endif
```

```
1  /** coord_demo.cpp */
2
3  #include "coords.h"
4
5  int main()
6  {
7      Coords a = {1.2, 2.4};
8      Coords b = make_unit(a);
9      print(a); // (1.20, 2.40)
10     print(b); // (0.45, 0.89)
11     return 0;
12 }
```

Ohne `const` kann das Argument verändert werden.

Der Aufrufer kann nicht unterscheiden, ob eine Kopie oder eine Referenz übergeben wird.


Ohne `const` kann das Argument verändert werden.

Der Aufrufer kann nicht unterscheiden, ob eine Kopie oder eine Referenz übergeben wird.

```
1  #include "coords.h"
2
3  void negate(Coords& v) {
4      v.x = -v.x;
5      v.y = -v.y;
6  }
7
8  int main(void)
9  {
10     Coords a = {1.2, 2.4};
11     print(a); // (1.20, 2.40)
12     negate(a);
13     print(a); // (-1.20, -2.40)
14     return 0;
15 }
```

Ohne `const` kann das Argument verändert werden.

Der Aufrufer kann nicht unterscheiden, ob eine Kopie oder eine Referenz übergeben wird.

 Klare Dokumentation erforderlich!

```
1  #include "coords.h"
2
3  void negate(Coords& v) {
4      v.x = -v.x;
5      v.y = -v.y;
6  }
7
8  int main(void)
9  {
10     Coords a = {1.2, 2.4};
11     print(a); // (1.20, 2.40)
12     negate(a);
13     print(a); // (-1.20, -2.40)
14     return 0;
15 }
```

# Richtlinie: Referenz oder Pointer?

Nutze Referenzen für

- Funktionsparameter, die nicht optional sind

# Richtlinie: Referenz oder Pointer?

Nutze Referenzen für

- Funktionsparameter, die nicht optional sind
- lokale Aliase zur Abkürzung, z.B.

```
1 | Coords& v = viele.verschachtelte.strukturen.langer_member_name;  
2 | // arbeite mit v weiter
```



# Richtlinie: Referenz oder Pointer?

Nutze Referenzen für

- Funktionsparameter, die nicht optional sind
- lokale Aliase zur Abkürzung, z.B.

```
1 | Coords& v = viele.verschachtelte.strukturen.langer_member_name;  
2 | // arbeite mit v weiter
```

Nutze Pointer für

- optionale Funktionsparameter (NULL als erlaubter Wert, ⚠ Prüfen vor Zugriff!)

# Richtlinie: Referenz oder Pointer?

## Nutze Referenzen für

- Funktionsparameter, die nicht optional sind
- lokale Aliase zur Abkürzung, z.B.

```
1 | Coords& v = viele.verschachtelte.strukturen.langer_member_name;  
2 | // arbeite mit v weiter
```

## Nutze Pointer für

- optionale Funktionsparameter (NULL als erlaubter Wert, ⚠️ Prüfen vor Zugriff!)
- Member in Datenstrukturen

# Richtlinie: Referenz oder Pointer?

## Nutze Referenzen für

- Funktionsparameter, die nicht optional sind
- lokale Aliase zur Abkürzung, z.B.

```
1 | Coords& v = viele.verschachtelte.strukturen.langer_member_name;  
2 | // arbeite mit v weiter
```

## Nutze Pointer für

- optionale Funktionsparameter (NULL als erlaubter Wert, ⚠ Prüfen vor Zugriff!)
- Member in Datenstrukturen
- Arrays, Pointer-Arithmetik, umstellbare Zeiger



Das Referenz-Symbol & in der Deklaration `int& x` hat die gleiche Syntax wie das Pointer-Symbol `*` und bindet an den *Namen*.



Das Referenz-Symbol & in der Deklaration `int& x` hat die gleiche Syntax wie das Pointer-Symbol `*` und bindet an den *Namen*.

```
1 | int a = 42;  
2 | int& x = a, y = a;
```

Hier ist nur `x` eine Referenz auf `a` (Typ `int&`).  
`y` hat Typ `int` und ist eine unabhängige Kopie.



Das Referenz-Symbol & in der Deklaration `int& x` hat die gleiche Syntax wie das Pointer-Symbol `*` und bindet an den *Namen*.

```
1 | int a = 42;  
2 | int& x = a, y = a;
```

Hier ist nur `x` eine Referenz auf `a` (Typ `int&`).  
`y` hat Typ `int` und ist eine unabhängige Kopie.

⇒ Mehrfach-Deklarationen mit Pointern *oder Referenzen* niemals in einer Zeile!

# Overloading

---

Funktionen werden in C und C++ unterschiedlich gehandhabt:

- C identifiziert Funktionen nur anhand des Namens.
- C++ identifiziert Funktionen anhand des Namens *und der Parameter-Liste*.



Funktionen werden in C und C++ unterschiedlich gehandhabt:

- C identifiziert Funktionen nur anhand des Namens.
- C++ identifiziert Funktionen anhand des Namens *und der Parameter-Liste*.

Darum können in C++ mehrere Funktionen mit gleichem Namen und unterschiedlichen Parametern angelegt werden. Wir sprechen dann von *Overloading* bzw. sagen die Funktion ist *überladen*.

---

Funktionen werden in C und C++ unterschiedlich gehandhabt:

- C identifiziert Funktionen nur anhand des Namens.
- C++ identifiziert Funktionen anhand des Namens *und der Parameter-Liste*.

Darum können in C++ mehrere Funktionen mit gleichem Namen und unterschiedlichen Parametern angelegt werden. Wir sprechen dann von *Overloading* bzw. sagen die Funktion ist *überladen*.

Beim Aufruf der Funktion findet *Overload-Resolution* statt: Der Compiler wählt die Variante, die „besser passt“<sup>1</sup>. Speziell gilt: Es gewinnt die Variante, die *ohne implizite Konvertierung der Argumente* auskommt.

---

<sup>1</sup>Ausführliches, komplexes Regelwerk:

[https://en.cppreference.com/w/cpp/language/overload\\_resolution](https://en.cppreference.com/w/cpp/language/overload_resolution)

```
1 #include <stdio>
2
3 void my_print(int i) {
4     std::printf("Int %d\n", i);
5 }
6
7 void my_print(double f) {
8     std::printf("Double %.2f\n", f);
9 }
10
11 int main() {
12     my_print(42);    // Int 42
13     my_print(3.14); // Double 3.14
14     short s = 7;
15     my_print(s);    // Int 7 (short->int passt besser als short->double)
16     float f = 9.81;
17     my_print(f);    // Double 9.81 (float->double passt besser als float->int)
18     return 0;
19 }
```

```
1 #include <cstdio>
2
3 void my_print(int i)    { std::printf("Int %d\n", i); }
4 void my_print(double f) { std::printf("Double %.2f\n", f); }
5
6 int main() {           // Erinnerung: Literal 42u bedeutet unsigned int
7     my_print((double) 42u); // Double 42.00
8     my_print(42u);        // FEHLER: Kein "bester" Overload.
9     return 0;
10 }
```

```

1 #include <stdio>
2
3 void my_print(int i)    { std::printf("Int %d\n", i); }
4 void my_print(double f) { std::printf("Double %.2f\n", f); }
5
6 int main() {           // Erinnerung: Literal 42u bedeutet unsigned int
7     my_print((double) 42u); // Double 42.00
8     my_print(42u);        // FEHLER: Kein "bester" Overload.
9     return 0;
10 }

```

test.cpp: In function 'int main()':

test.cpp:13:15: error: call of overloaded 'my\_print(unsigned int)' is ambiguous

```

    my_print(42u);           // FEHLER: Kein "bester" Overload.
        ^

```

test.cpp:3:6: note: candidate: void my\_print(int)

```

    void my_print(int i) {
        ^

```

test.cpp:7:6: note: candidate: void my\_print(double)

```

    void my_print(double f) {
        ^

```

# Operator Overloading

---

In C++ können wir die vorhandenen Operatoren für die Nutzung mit eigenen Datentypen überladen. Wir definieren dazu eine Funktion, deren Name aus dem Wort `operator` und dem Symbol des jeweiligen Operators zusammengesetzt ist.

**Beispiel:** Definiere `+` als Vektor-Addition für unseren Typ `Coords`.

In C++ können wir die vorhandenen Operatoren für die Nutzung mit eigenen Datentypen überladen. Wir definieren dazu eine Funktion, deren Name aus dem Wort `operator` und dem Symbol des jeweiligen Operators zusammengesetzt ist.

**Beispiel:** Definiere `+` als Vektor-Addition für unseren Typ `Coords`.

Um den Operator `a+b` zu überladen, definieren wir die Funktion `operator+(a, b)` und übergeben die Parameter `a` und `b` als Referenzen.



In C++ können wir die vorhandenen Operatoren für die Nutzung mit eigenen Datentypen überladen. Wir definieren dazu eine Funktion, deren Name aus dem Wort `operator` und dem Symbol des jeweiligen Operators zusammengesetzt ist.

**Beispiel:** Definiere `+` als Vektor-Addition für unseren Typ `Coords`.

Um den Operator `a+b` zu überladen, definieren wir die Funktion `operator+(a, b)` und übergeben die Parameter `a` und `b` als Referenzen.

Im speziellen Fall deklarieren wir die Funktion also als

`Coords operator+(const Coords& a, const Coords& b);`

```
1  #include "coords.h"
2
3  Coords operator+(const Coords& a, const Coords& b) {
4      Coords o;
5      o.x = a.x + b.x;
6      o.y = a.y + b.y;
7      return o;
8  }
9
10 int main() {
11     Coords v1 = {1.2, 2.3};
12     Coords v2 = {3.4, 4.5};
13     print(v1 + v2); // Coords (4.60, 6.80)
14     return 0;
15 }
```

💡 Das Ergebnis von  $v1 + v2$  wird in keiner Variablen gespeichert. Es ist ein *temporärer Wert*.

```
1  #include "coords.h"
2
3  Coords operator+(const Coords& a, const Coords& b) {
4      Coords o;
5      o.x = a.x + b.x;
6      o.y = a.y + b.y;
7      return o;
8  }
9
10 int main() {
11     Coords v1 = {1.2, 2.3};
12     Coords v2 = {3.4, 4.5};
13     print(v1 + v2); // Coords (4.60, 6.80)
14     return 0;
15 }
```

💡 Das Ergebnis von  $v1 + v2$  wird in keiner Variablen gespeichert. Es ist ein *temporärer Wert*.

💡 Einen Temporären Wert kann `print` deshalb nutzen, weil es eine `const` Referenz entgegennimmt (statt einer Referenz ohne `const` oder Pointer).

```
1  #include "coords.h"
2
3  Coords operator+(const Coords& a, const Coords& b) {
4      Coords o;
5      o.x = a.x + b.x;
6      o.y = a.y + b.y;
7      return o;
8  }
9
10 int main() {
11     Coords v1 = {1.2, 2.3};
12     Coords v2 = {3.4, 4.5};
13     print(v1 + v2); // Coords (4.60, 6.80)
14     return 0;
15 }
```

## C++ I/O Streams

---

Von C kennen wir den Header `<cstdio>` (`<stdio.h>`), der Funktionalität für Ein- und Ausgaben in Form von Funktionen wie `puts`, `printf` und Datenstrom-Objekten wie `stdin` bereitstellt. Dieser ist weiter nutzbar, aber „not the C++ way of doing things“.

Von C kennen wir den Header `<cstdio>` (`<stdio.h>`), der Funktionalität für Ein- und Ausgaben in Form von Funktionen wie `puts`, `printf` und Datenstrom-Objekten wie `stdin` bereitstellt. Dieser ist weiter nutzbar, aber „not the C++ way of doing things“.

C++ stellt eigene Funktionalität für Ein- und Ausgaben bereit, die einige Vorteile bietet und von C++ Code genutzt werden sollte. Der Header `<iostream>` stellt dazu verschiedene Stream-Objekte bereit, unterteilt in Eingänge (`istream`) und Ausgänge (`ostream`):

Von C kennen wir den Header `<cstdio>` (`<stdio.h>`), der Funktionalität für Ein- und Ausgaben in Form von Funktionen wie `puts`, `printf` und Datenstrom-Objekten wie `stdin` bereitstellt. Dieser ist weiter nutzbar, aber „not the C++ way of doing things“.

C++ stellt eigene Funktionalität für Ein- und Ausgaben bereit, die einige Vorteile bietet und von C++ Code genutzt werden sollte. Der Header `<iostream>` stellt dazu verschiedene Stream-Objekte bereit, unterteilt in Eingänge (`istream`) und Ausgänge (`ostream`):

**`cout`** Standard `ostream` für Ausgaben auf die Kommandozeile



Von C kennen wir den Header `<cstdio>` (`<stdio.h>`), der Funktionalität für Ein- und Ausgaben in Form von Funktionen wie `puts`, `printf` und Datenstrom-Objekten wie `stdin` bereitstellt. Dieser ist weiter nutzbar, aber „not the C++ way of doing things“.

C++ stellt eigene Funktionalität für Ein- und Ausgaben bereit, die einige Vorteile bietet und von C++ Code genutzt werden sollte. Der Header `<iostream>` stellt dazu verschiedene Stream-Objekte bereit, unterteilt in Eingänge (`istream`) und Ausgänge (`ostream`):

**cout** Standard `ostream` für Ausgaben auf die Kommandozeile

**cerr** Standard `ostream` für Fehlermeldungen auf die Kommandozeile

Von C kennen wir den Header `<cstdio>` (`<stdio.h>`), der Funktionalität für Ein- und Ausgaben in Form von Funktionen wie `puts`, `printf` und Datenstrom-Objekten wie `stdin` bereitstellt. Dieser ist weiter nutzbar, aber „not the C++ way of doing things“.

C++ stellt eigene Funktionalität für Ein- und Ausgaben bereit, die einige Vorteile bietet und von C++ Code genutzt werden sollte. Der Header `<iostream>` stellt dazu verschiedene Stream-Objekte bereit, unterteilt in Eingänge (`istream`) und Ausgänge (`ostream`):

**cout** Standard `ostream` für Ausgaben auf die Kommandozeile

**cerr** Standard `ostream` für Fehlermeldungen auf die Kommandozeile

**cin** Standard `istream` für Eingaben auf der Kommandozeile

Für ostream Objekte wie cout ist der Operator << überladen, um Basisdatentypen als Text formatiert auszugeben. Wir können schreiben:

```
std::cout << "Hello, World!\n";
```

Für ostream Objekte wie cout ist der Operator << überladen, um Basisdatentypen als Text formatiert auszugeben. Wir können schreiben:

```
std::cout << "Hello, World!\n";
```

Diese Benutzung des << Operators hat keinerlei Bezug zu dessen ursprünglicher Definition als arithmetischer Operator (bitweiser Links-Shift<sup>2</sup>). Sie ist jedoch so verbreitet, dass vereinzelt << auch als „Stream-Operator“ bezeichnet wird.

---

<sup>2</sup>[https://en.cppreference.com/w/cpp/language/operator\\_arithmetic](https://en.cppreference.com/w/cpp/language/operator_arithmetic)

# Hello, World (finale Form)

```
1 #include <iostream>
2
3 int main() {
4     std::cout << "Hello, World of C++!\n";
5     return 0;
6 }
```

Wir können mehrere << Operatoren verketten:

```
1 | std::cout << "Mehrere " << "Ausgaben " << "nacheinander.\n";  
2 | // Ausgabe: Mehrere Ausgaben nacheinander.
```

Wir können mehrere << Operatoren verketten:

```
1 | std::cout << "Mehrere " << "Ausgaben " << "nacheinander.\n";  
2 | // Ausgabe: Mehrere Ausgaben nacheinander.
```

Da mit Overloads alle Basisdatentypen abgedeckt sind, entfallen Format-Strings:

```
1 | int a = 7;  
2 | double b = 10.0;  
3 | std::cout << a << " geteilt durch " << b << " ergibt " << a/b << ".\n";  
4 | // Ausgabe: 7 geteilt durch 10 ergibt 0.7.
```

Die Streams `cout` und `cerr` sind *buffered*, d.h. Ausgaben werden aus Gründen der Performance zunächst zwischengespeichert und später zu einem vom System bestimmten Zeitpunkt gesammelt geschrieben. Möchte man z.B. das Verhalten des Programms analysieren, kann es wichtig sein, dass Ausgaben unmittelbar geschrieben werden. Einen solchen *Flush* des Buffers löst der Stream-Manipulator `endl` aus:



Die Streams `cout` und `cerr` sind *buffered*, d.h. Ausgaben werden aus Gründen der Performance zunächst zwischengespeichert und später zu einem vom System bestimmten Zeitpunkt gesammelt geschrieben. Möchte man z.B. das Verhalten des Programms analysieren, kann es wichtig sein, dass Ausgaben unmittelbar geschrieben werden. Einen solchen *Flush* des Buffers löst der Stream-Manipulator `endl` aus:

`endl` schreibt einen Zeilenumbruch und flusht den Stream-Buffer.

Die Streams `cout` und `cerr` sind *buffered*, d.h. Ausgaben werden aus Gründen der Performance zunächst zwischengespeichert und später zu einem vom System bestimmten Zeitpunkt gesammelt geschrieben. Möchte man z.B. das Verhalten des Programms analysieren, kann es wichtig sein, dass Ausgaben unmittelbar geschrieben werden. Einen solchen *Flush* des Buffers löst der Stream-Manipulator `endl` aus:

`endl` schreibt einen Zeilenumbruch und flusht den Stream-Buffer.

### Beispiel:

```
1 | std::cout << "Hello, World!" << std::endl;  
2 | // Effekt wie "Hello, World!\n", aber wird sofort geschrieben.
```

Mit printf-Formatstrings konnten wir z.B. Gleitkommazahlen genau formatieren. Für C++ Streams wird die spezifische Formatierung der Ausgabe durch Stream-Manipulatoren aus dem Header `<iomanip>`<sup>3</sup> ermöglicht:

---

<sup>3</sup><https://en.cppreference.com/w/cpp/io/manip>

Mit printf-Formatstrings konnten wir z.B. Gleitkommazahlen genau formatieren. Für C++ Streams wird die spezifische Formatierung der Ausgabe durch Stream-Manipulatoren aus dem Header `<iomanip>`<sup>3</sup> ermöglicht:

```
1 #include <iostream>
2 #include <iomanip>
3
4 int main()
5 {
6     std::cout << 1 / 3.0 << std::endl;
7     // 0.333333
8     std::cout << std::setprecision(2) << 1 / 3.0 << std::endl;
9     // 0.33
10 }
```

---

<sup>3</sup><https://en.cppreference.com/w/cpp/io/manip>

Das Lesen von `istreams` wie `cin` funktioniert, als Spiegelung der Syntax für Ausgaben, mit dem Operator `>>` (eigentlich: Bitweiser Rechts-Shift).

Das Lesen von istreams wie cin funktioniert, als Spiegelung der Syntax für Ausgaben, mit dem Operator >> (eigentlich: Bitweiser Rechts-Shift).

Die Overloads von `operator>>` ermöglichen das direkte Interpretieren von Eingaben als Datentypen.

# Lesen von cin

Das Lesen von istreams wie cin funktioniert, als Spiegelung der Syntax für Ausgaben, mit dem Operator >> (eigentlich: Bitweiser Rechts-Shift).

Die Overloads von `operator>>` ermöglichen das direkte Interpretieren von Eingaben als Datentypen.

```
1 #include <iostream>
2
3 int main() {
4     int a = 0;
5     double b = 0.0;
6     std::cout << "Ein int und ein double bitte: ";
7     std::cin >> a >> b;
8     std::cout << "Ich habe gelesen: " << a
9         << " und " << b << std::endl;
10    return 0;
11 }
```

```
> Ein int und ein double bitte: 7 3.14
Ich habe gelesen: 7 und 3.14
```

- Wie schon mit `atoi` betrachten wir an dieser Stelle keine, für echte Programme wichtige, Behandlung von Falscheingaben<sup>4</sup>. Diese ist mit dem gezeigten Verfahren prinzipiell möglich und erfordert zum Teil Konzepte, die wir noch kennen lernen werden.

---

<sup>4</sup>Weitere Informationen z.B. hier: <https://isocpp.org/wiki/faq/input-output>



# Anmerkungen zum Einlesen

- Wie schon mit `atoi` betrachten wir an dieser Stelle keine, für echte Programme wichtige, Behandlung von Falscheingaben<sup>4</sup>. Diese ist mit dem gezeigten Verfahren prinzipiell möglich und erfordert zum Teil Konzepte, die wir noch kennen lernen werden.
- Ein wichtiges Thema bei Ein- und Ausgaben ist auch die sog. *Locale*<sup>5</sup>, also die Einstellung, die landesspezifisch entscheidet ob z.B. der Dezimaltrenner der Punkt oder das Komma ist.

---

<sup>4</sup>Weitere Informationen z.B. hier: <https://isocpp.org/wiki/faq/input-output>

<sup>5</sup>Kurzeinstieg zu Locales:

[https://www.boost.org/doc/libs/1\\_48\\_0/libs/locale/doc/html/std\\_locales.html](https://www.boost.org/doc/libs/1_48_0/libs/locale/doc/html/std_locales.html)

# Anmerkungen zum Einlesen

- Wie schon mit `atoi` betrachten wir an dieser Stelle keine, für echte Programme wichtige, Behandlung von Falscheingaben<sup>4</sup>. Diese ist mit dem gezeigten Verfahren prinzipiell möglich und erfordert zum Teil Konzepte, die wir noch kennen lernen werden.
- Ein wichtiges Thema bei Ein- und Ausgaben ist auch die sog. *Locale*<sup>5</sup>, also die Einstellung, die landesspezifisch entscheidet ob z.B. der Dezimaltrenner der Punkt oder das Komma ist.
- C++ nutzt eine eigene Repräsentation von Zeichenketten: `std::string`. Diese werden wir noch kennen lernen.

---

<sup>4</sup>Weitere Informationen z.B. hier: <https://isocpp.org/wiki/faq/input-output>

<sup>5</sup>Kurzeinstieg zu Locales:

[https://www.boost.org/doc/libs/1\\_48\\_0/libs/locale/doc/html/std\\_locales.html](https://www.boost.org/doc/libs/1_48_0/libs/locale/doc/html/std_locales.html)

## using und Namespaces

---

Wir stellen fest, dass wir seit Einführung der Namespaces in manchen Situationen häufig wiederholt `std` (oder einen anderen Namespace) tippen müssen, so wie hier:

```
1 | int main() {  
2 |     std::cout << std::setprecision(2) << 1/3.0 << std::endl;  
3 |     return 0;  
4 | }
```

Wir stellen fest, dass wir seit Einführung der Namespaces in manchen Situationen häufig wiederholt `std` (oder einen anderen Namespace) tippen müssen, so wie hier:

```
1 int main() {  
2     std::cout << std::setprecision(2) << 1/3.0 << std::endl;  
3     return 0;  
4 }
```

Die `using` Deklaration kann hier helfen, den Code etwas kompakter zu gestalten. Die erste Form macht einzelne Elemente eines Namespace im aktuellen Scope sichtbar:

```
1 int main() {  
2     using std::cout;  
3     using std::setprecision;  
4     using std::endl;  
5     cout << setprecision(2) << 1/3.0 << endl;  
6     return 0;  
7 }
```

Die zweite Form, `using namespace`, ist deutlich aggressiver und macht *alle* Elemente des betreffenden Namespace sichtbar:

```
1 | int main() {  
2 |     using namespace std;  
3 |     cout << setprecision(2) << 1/3.0 << endl;  
4 |     return 0;  
5 | }
```

- + Praktisch
- Erhöhte Gefahr von Namenskonflikten

Wir können `using` auch außerhalb einer Funktion nutzen. Die sichtbargemachten Elemente werden dann im gesamten aktuellen Namespace sichtbar:

```
1 using namespace std;
2
3 int main() {
4     cout << setprecision(2) << 1/3.0 << endl;
5     return 0;
6 }
```

**⚠ Niemals sollte in einem Header per `using` ein gesamter Namespace (insbesondere nicht `std`) global sichtbar gemacht werden. Die Wirkung mitsamt Konflikt-Risiko betreffe jeden Nutzer des Headers.**

Bevorzuge, mit absteigender Präferenz, ...

1. Verzicht auf `using` und explizite Benennung über `::`



Bevorzuge, mit absteigender Präferenz, ...

1. Verzicht auf `using` und explizite Benennung über `::`
2. Sichtbarmachen einzelner Elemente *innerhalb einer Funktion*,  
z.B. mit `using std::cout`

Bevorzuge, mit absteigender Präferenz, ...

1. Verzicht auf `using` und explizite Benennung über `::`
2. Sichtbarmachen einzelner Elemente *innerhalb einer Funktion*,  
z.B. mit `using std::cout`
3. Sichtbarmachen eines ganzen Namespace *innerhalb einer Funktion*,  
z.B. mit `using namespace std`

Bevorzuge, mit absteigender Präferenz, ...

1. Verzicht auf `using` und explizite Benennung über `::`
2. Sichtbarmachen einzelner Elemente *innerhalb einer Funktion*,  
z.B. mit `using std::cout`
3. Sichtbarmachen eines ganzen Namespace *innerhalb einer Funktion*,  
z.B. mit `using namespace std`
4. (2), außerhalb einer Funktion aber *nur innerhalb eines .cpp Files*

Bevorzuge, mit absteigender Präferenz, ...

1. Verzicht auf `using` und explizite Benennung über `::`
2. Sichtbarmachen einzelner Elemente *innerhalb einer Funktion*,  
z.B. mit `using std::cout`
3. Sichtbarmachen eines ganzen Namespace *innerhalb einer Funktion*,  
z.B. mit `using namespace std`
4. (2), außerhalb einer Funktion aber *nur innerhalb eines .cpp Files*
5. (3), außerhalb einer Funktion aber *nur innerhalb eines .cpp Files*

Hinweis: `bool` in C++

---

In C++ sind der Typ `bool` und die Literale `true` und `false` seit dem ersten Standard ohne nötiges Einbinden eines Headers vorhanden und sollen für Wahrheitswerte genutzt werden.

Die Interpretation von Zahlen als Wahrheitswerte (0 entspricht `false`, alles andere `true`) bleibt aus C erhalten und ist, insbesondere zur Überprüfung von Pointern auf `NULL`, häufig zu sehen.

Ende