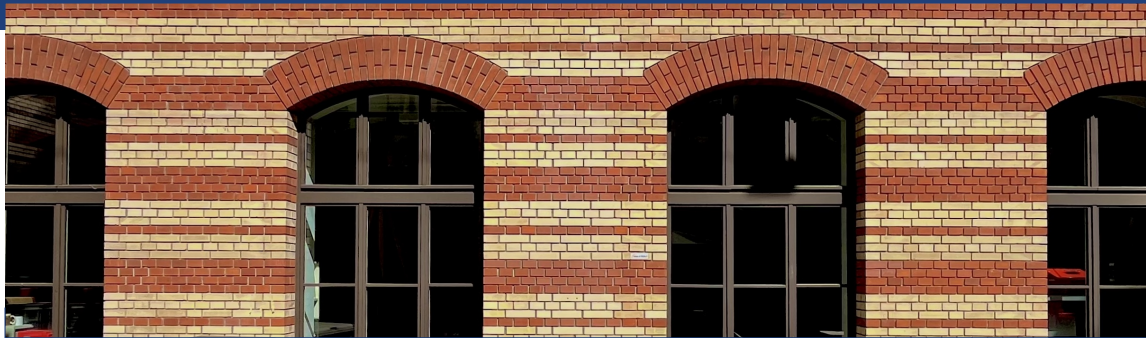




Informatik

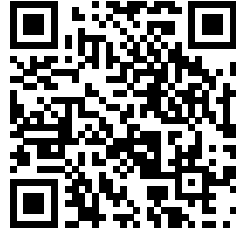
Referenzen, Vektoren, Matrizen

Adel Gavranović

ETH Zürich — 2026

Überblick

Follow-up: Header und .cpp-Dateien
Referenzen
Vektoren und Referenzen
Mehr zu Referenzen
Multidimensionale Vektoren



`adelgavranovic.ch`

 Webpage

 Mail

1. Follow-up

Follow-up aus letzter Übungsstunde

Follow-up aus letzter Übungsstunde

- Eure Frage zu den Headers: `main.cpp` bindet nur die `.h`-Datei ein. Woher kommt dann der Code aus der `.cpp`-Datei?

2. Follow-up: Header und .cpp-Dateien

Eure Frage

myCoolFunctions.h

```
#pragma once  
  
int someFunction(int x);
```

myCoolFunctions.cpp

```
#include "myCoolFunctions.h"  
  
int someFunction(int x) {  
    return x + 1;  
}
```

main.cpp

```
#include "myCoolFunctions.h"  
  
int main() {  
    return someFunction(42);  
}
```

main.cpp bindet nur myCoolFunctions.h ein, nie myCoolFunctions.cpp (wie letzte Woche myAwesomeFunctions.h). Woher weiss der Compiler, wo der Code von someFunction steht?

Eure Frage

myCoolFunctions.h

```
#pragma once  
  
int someFunction(int x);
```

myCoolFunctions.cpp

```
#include "myCoolFunctions.h"  
  
int someFunction(int x) {  
    return x + 1;  
}
```

main.cpp

```
#include "myCoolFunctions.h"  
  
int main() {  
    return someFunction(42);  
}
```

main.cpp bindet nur myCoolFunctions.h ein, nie myCoolFunctions.cpp (wie letzte Woche myAwesomeFunctions.h). Woher weiss der Compiler, wo der Code von someFunction steht?

Der Compiler weiss es nicht, und er muss es auch nicht wissen. Der *Linker* findet den Code, und zwar nur, weil ihr ihm sagt, welche Dateien er zusammenfügen soll.

Schritt 1: Der Präprozessor kopiert den Header hinein

Das sieht der Compiler von
main.cpp:

```
// pasted from myCoolFunctions.h
int someFunction(int x);

int main() {
    return someFunction(42);
}
```

Schritt 1: Der Präprozessor kopiert den Header hinein

Das sieht der Compiler von
main.cpp:

```
// pasted from myCoolFunctions.h
int someFunction(int x);

int main() {
    return someFunction(42);
}
```

- `#include "myCoolFunctions.h"`
ist reines Ersetzen von Text

Schritt 1: Der Präprozessor kopiert den Header hinein

Das sieht der Compiler von
main.cpp:

```
// pasted from myCoolFunctions.h
int someFunction(int x);

int main() {
    return someFunction(42);
}
```

- `#include "myCoolFunctions.h"`
ist reines Ersetzen von Text
- Vor dem Kompilieren kopiert der
Präprozessor den Inhalt des
Headers an genau diese Stelle

Schritt 1: Der Präprozessor kopiert den Header hinein

Das sieht der Compiler von `main.cpp`:

```
// pasted from myCoolFunctions.h
int someFunction(int x);

int main() {
    return someFunction(42);
}
```

- `#include "myCoolFunctions.h"` ist reines Ersetzen von Text
- Vor dem Kompilieren kopiert der Präprozessor den Inhalt des Headers an genau diese Stelle
- Übrig bleibt nur die *Deklaration*, kein Funktionsrumpf

Selbst anschauen: `g++ -E main.cpp`. Bei `<iostream>` kommen so über 90 000 Zeilen heraus (auf meinem Mac).

Schritt 2: Eine .cpp-Datei nach der anderen

Schritt 2: Eine .cpp-Datei nach der anderen

- Jede .cpp-Datei wird allein kompiliert, ohne die anderen zu sehen. So eine Datei heisst *Übersetzungseinheit* (translation unit)

Schritt 2: Eine .cpp-Datei nach der anderen

- Jede .cpp-Datei wird allein kompiliert, ohne die anderen zu sehen. So eine Datei heisst *Übersetzungseinheit* (translation unit)
- Für den Aufruf reicht die Deklaration: Name, Parametertypen und Rückgabetyt. Den Funktionsrumpf braucht der Compiler nicht

Schritt 2: Eine .cpp-Datei nach der anderen

- Jede .cpp-Datei wird allein kompiliert, ohne die anderen zu sehen. So eine Datei heisst *Übersetzungseinheit* (translation unit)
- Für den Aufruf reicht die Deklaration: Name, Parametertypen und Rückgabotyp. Den Funktionsrumpf braucht der Compiler nicht

`main.cpp` → `main.o`

Maschinencode für `main`, plus eine
Notiz: “ruft `someFunction(int)` auf,
das ist woanders definiert”

Diese Notiz heisst *un aufgelöstes
Symbol*

Schritt 2: Eine .cpp-Datei nach der anderen

- Jede .cpp-Datei wird allein kompiliert, ohne die anderen zu sehen. So eine Datei heisst *Übersetzungseinheit* (translation unit)
- Für den Aufruf reicht die Deklaration: Name, Parametertypen und Rückgabetyp. Den Funktionsrumpf braucht der Compiler nicht

`main.cpp` → `main.o`

Maschinencode für `main`, plus eine Notiz: “ruft `someFunction(int)` auf, das ist woanders definiert”

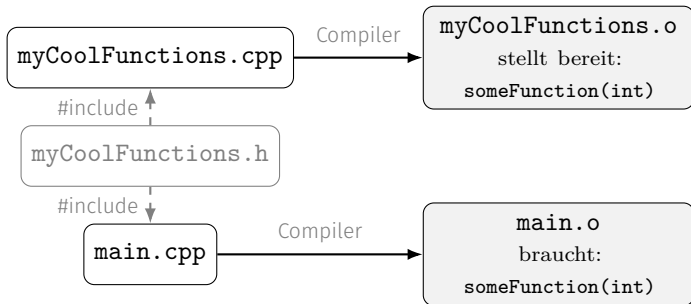
Diese Notiz heisst *unaufgelöstes Symbol*

`myCoolFunctions.cpp` →

`myCoolFunctions.o`

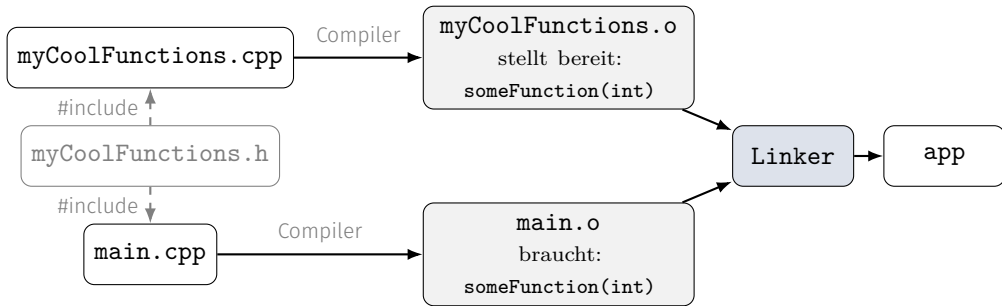
Enthält die eigentliche Definition und sagt: “ich stelle `someFunction(int)` bereit”

Schritt 3: Der Linker verbindet die Teile



Der Linker nimmt alle Objektdaten, die ihr ihm gebt. Er ordnet jedem unaufgelösten Symbol eine Definition zu und setzt die echten Adressen ein. Heraus kommt *ein* ausführbares Programm.

Schritt 3: Der Linker verbindet die Teile



Der Linker nimmt alle Objektdaten, die ihr ihm gibt. Er ordnet jedem unaufgelösten Symbol eine Definition zu und setzt die echten Adressen ein. Heraus kommt *ein* ausführbares Programm.

Der Knackpunkt: Ihr müsst beide Dateien angeben

Der Knackpunkt: Ihr müsst beide Dateien angeben

- Der Compiler sucht `myCoolFunctions.cpp` nie, nur weil er `myCoolFunctions.h` gesehen hat

Der Knackpunkt: Ihr müsst beide Dateien angeben

- Der Compiler sucht `myCoolFunctions.cpp` nie, nur weil er `myCoolFunctions.h` gesehen hat
- Dass beide gleich heissen, ist reine Konvention für uns Menschen

Der Knackpunkt: Ihr müsst beide Dateien angeben

- Der Compiler sucht `myCoolFunctions.cpp` nie, nur weil er `myCoolFunctions.h` gesehen hat
- Dass beide gleich heissen, ist reine Konvention für uns Menschen

```
# all in one go
g++ main.cpp myCoolFunctions.cpp -o app

# or compile separately, then link
g++ -c main.cpp          # -> main.o
g++ -c myCoolFunctions.cpp # -> myCoolFunctions.o
g++ main.o myCoolFunctions.o -o app
```

Auf **code expert** passiert das für euch: es nimmt alle `.cpp`-Dateien des Projekts.

Datei vergessen? Dann meldet sich der Linker

Ohne `myCoolFunctions.cpp` kompiliert `main.cpp` problemlos. Erst der Linker scheitert:

```
$ g++ main.cpp -o app  
undefined reference to `someFunction(int)'
```

So bei GCC (**code expert**). Mein Mac schreibt: `Undefined symbols ...`
`"someFunction(int)"`

Datei vergessen? Dann meldet sich der Linker

Ohne `myCoolFunctions.cpp` kompiliert `main.cpp` problemlos. Erst der Linker scheitert:

```
$ g++ main.cpp -o app  
undefined reference to `someFunction(int)'
```

So bei GCC (**code expert**). Mein Mac schreibt: `Undefined symbols ...
"someFunction(int)"`

💡 Diese Meldung heisst fast immer: Eine Datei oder Bibliothek wurde nicht mitgelinkt.

Warum nicht einfach die .cpp-Datei einbinden?

Warum nicht einfach die .cpp-Datei einbinden?

- Geht tatsächlich, solange `main.cpp` die einzige Datei ist: `#include` kopiert jede Datei hinein

Warum nicht einfach die .cpp-Datei einbinden?

- Geht tatsächlich, solange `main.cpp` die einzige Datei ist: `#include` kopiert jede Datei hinein
- Wird `myCoolFunctions.cpp` aber auch kompiliert (oder von einer zweiten Datei eingebunden), ist `someFunction` zweimal definiert:

```
$ g++ main.cpp myCoolFunctions.cpp      # main.cpp has #include "myCoolFunctions.cpp"
duplicate symbol 'someFunction(int)'      (GCC: multiple definition of ...)
```

Warum nicht einfach die .cpp-Datei einbinden?

- Geht tatsächlich, solange `main.cpp` die einzige Datei ist: `#include` kopiert jede Datei hinein
- Wird `myCoolFunctions.cpp` aber auch kompiliert (oder von einer zweiten Datei eingebunden), ist `someFunction` zweimal definiert:

```
$ g++ main.cpp myCoolFunctions.cpp      # main.cpp has #include "myCoolFunctions.cpp"
duplicate symbol 'someFunction(int)'      (GCC: multiple definition of ...)
```

- Die Regel: Deklarationen dürfen beliebig oft vorkommen, die Definition genau einmal im ganzen Programm

Warum nicht einfach die .cpp-Datei einbinden?

- Geht tatsächlich, solange `main.cpp` die einzige Datei ist: `#include` kopiert jede Datei hinein
- Wird `myCoolFunctions.cpp` aber auch kompiliert (oder von einer zweiten Datei eingebunden), ist `someFunction` zweimal definiert:

```
$ g++ main.cpp myCoolFunctions.cpp      # main.cpp has #include "myCoolFunctions.cpp"
duplicate symbol 'someFunction(int)'      (GCC: multiple definition of ...)
```

- Die Regel: Deklarationen dürfen beliebig oft vorkommen, die Definition genau einmal im ganzen Programm
- `#pragma once` hilft hier nicht: es verhindert nur, dass ein Header zweimal in *dieselbe* .cpp-Datei kopiert wird

Warum nicht einfach die .cpp-Datei einbinden?

- Geht tatsächlich, solange `main.cpp` die einzige Datei ist: `#include` kopiert jede Datei hinein
- Wird `myCoolFunctions.cpp` aber auch kompiliert (oder von einer zweiten Datei eingebunden), ist `someFunction` zweimal definiert:

```
$ g++ main.cpp myCoolFunctions.cpp      # main.cpp has #include "myCoolFunctions.cpp"
duplicate symbol 'someFunction(int)'      (GCC: multiple definition of ...)
```

- Die Regel: Deklarationen dürfen beliebig oft vorkommen, die Definition genau einmal im ganzen Programm
- `#pragma once` hilft hier nicht: es verhindert nur, dass ein Header zweimal in *dieselbe* .cpp-Datei kopiert wird

💡 Bindet .h-Dateien ein, kompiliert .cpp-Dateien. Nie `#include "...cpp"`.

Und in grösseren Projekten?

Und in grösseren Projekten?

- Ein Build-System wie Make oder CMake führt die Liste der Quelldateien und ruft diese Befehle für euch auf

Und in grösseren Projekten?

- Ein Build-System wie Make oder CMake führt die Liste der Quelldateien und ruft diese Befehle für euch auf
- Bei vorkompilierten Bibliotheken liegen die Definitionen in einer `.a`-, `.so`- oder `.lib`-Datei. Dem Linker zeigt man sie mit `-L/pfad` und `-lname`

Und in grösseren Projekten?

- Ein Build-System wie Make oder CMake führt die Liste der Quelldateien und ruft diese Befehle für euch auf
- Bei vorkompilierten Bibliotheken liegen die Definitionen in einer `.a`-, `.so`- oder `.lib`-Datei. Dem Linker zeigt man sie mit `-L/pfad` und `-lname`
- Die Standardbibliothek (`std::cout` & Co.) ist so eine Bibliothek. `g++` linkt sie automatisch dazu, darum müsst ihr dafür nie etwas angeben

Alles nochmals in Ruhe, als Video (Englisch):  Low Level: "why do header files even exist?"

Fragen/Unklarheiten?

3. Feedback zu **code** expert

Allgemeines bezüglich **code expert**

Allgemeines bezüglich **code expert**

- Versteckte Testfälle sind jetzt aktiv, ab den Übungen von Woche 6 (seit gestern online)

Allgemeines bezüglich **code expert**

- Versteckte Testfälle sind jetzt aktiv, ab den Übungen von Woche 6 (seit gestern online)
 - Übungen mit “[Hidden tests]” werden nach der Abgabefrist mit *neuen* Tests neu bewertet

Allgemeines bezüglich **code expert**

- Versteckte Testfälle sind jetzt aktiv, ab den Übungen von Woche 6 (seit gestern online)
 - Übungen mit “[Hidden tests]” werden nach der Abgabefrist mit *neuen* Tests neu bewertet
 - Wer erst nach dem Neubewerten abgibt, wird nicht mehr neu bewertet

Allgemeines bezüglich **code expert**

- Versteckte Testfälle sind jetzt aktiv, ab den Übungen von Woche 6 (seit gestern online)
 - Übungen mit “[Hidden tests]” werden nach der Abgabefrist mit *neuen* Tests neu bewertet
 - Wer erst nach dem Neubewerten abgibt, wird nicht mehr neu bewertet
 - Mein Feedback zu diesen Übungen kommt deshalb erst nach dem Neubewerten

Allgemeines bezüglich **code expert**

- Versteckte Testfälle sind jetzt aktiv, ab den Übungen von Woche 6 (seit gestern online)
 - Übungen mit “[Hidden tests]” werden nach der Abgabefrist mit *neuen* Tests neu bewertet
 - Wer erst nach dem Neubewerten abgibt, wird nicht mehr neu bewertet
 - Mein Feedback zu diesen Übungen kommt deshalb erst nach dem Neubewerten
- Challenge Exercise 1 ist online: Abgabe bis 9. November

Allgemeines bezüglich **code expert**

- Versteckte Testfälle sind jetzt aktiv, ab den Übungen von Woche 6 (seit gestern online)
 - Übungen mit “[Hidden tests]” werden nach der Abgabefrist mit *neuen* Tests neu bewertet
 - Wer erst nach dem Neubewerten abgibt, wird nicht mehr neu bewertet
 - Mein Feedback zu diesen Übungen kommt deshalb erst nach dem Neubewerten
- Challenge Exercise 1 ist online: Abgabe bis 9. November
- Bei einer Aufgabe steckengeblieben? Kommt ins Study Center (jeden Mittwoch, Infos auf der Study-Center-Webseite)!  Study Center

Allgemeines bezüglich **code expert**

- Versteckte Testfälle sind jetzt aktiv, ab den Übungen von Woche 6 (seit gestern online)
 - Übungen mit “[Hidden tests]” werden nach der Abgabefrist mit *neuen* Tests neu bewertet
 - Wer erst nach dem Neubewerten abgibt, wird nicht mehr neu bewertet
 - Mein Feedback zu diesen Übungen kommt deshalb erst nach dem Neubewerten
- Challenge Exercise 1 ist online: Abgabe bis 9. November
- Bei einer Aufgabe steckengeblieben? Kommt ins Study Center (jeden Mittwoch, Infos auf der Study-Center-Webseite)!  Study Center

💡 Gebt nur aus, was die Aufgabe verlangt. Zusätzliche Ausgaben können die alten Tests zufällig bestehen und die neuen nicht.

Fragen zu den Übungen der letzten Woche?

Fragen bezüglich **code expert** eurerseits?

4. Lernziele

Ziele

Ziele

- ☐ Programme mit Referenzen tracen und schreiben können
- ☐ Programme schreiben können, die Vektoren erstellen, verändern und durchlaufen
- ☐ Programme schreiben können, die zweidimensionale Vektoren bearbeiten

5. Zusammenfassung

Getting on the same page

Getting on the same page

- Was wurde diese Woche behandelt?

6. Referenzen

Referenzen

Eine Referenz ist ein *Alias*: ein zweiter Name für eine Variable, die es schon gibt.

Referenzen

Eine Referenz ist ein *Alias*: ein zweiter Name für eine Variable, die es schon gibt.

```
int a = 3;  
int& b = a; // b is another name for a
```

Referenzen

Eine Referenz ist ein *Alias*: ein zweiter Name für eine Variable, die es schon gibt.

```
int a = 3;  
int& b = a; // b is another name for a
```

- Es gibt danach nur *eine* Variable, mit zwei Namen. Was ihr mit **b** macht, passiert mit **a**

Referenzen

Eine Referenz ist ein *Alias*: ein zweiter Name für eine Variable, die es schon gibt.

```
int a = 3;  
int& b = a; // b is another name for a
```

- Es gibt danach nur *eine* Variable, mit zwei Namen. Was ihr mit **b** macht, passiert mit **a**
- Eine Referenz muss sofort initialisiert werden, und zwar mit einer Variablen (L-Wert) vom gleichen Typ

Referenzen

Eine Referenz ist ein *Alias*: ein zweiter Name für eine Variable, die es schon gibt.

```
int a = 3;  
int& b = a; // b is another name for a
```

- Es gibt danach nur *eine* Variable, mit zwei Namen. Was ihr mit `b` macht, passiert mit `a`
- Eine Referenz muss sofort initialisiert werden, und zwar mit einer Variablen (L-Wert) vom gleichen Typ
- `int& c = 3;` geht also nicht: `3` hat keine Adresse

Referenzen tracen

```
int a = 3;  
int& b = a;  
b = 7;  
std::cout << a;
```

main

a	3
---	---

Ausgabe:

Referenzen tracen

```
int a = 3;  
int& b = a;  
b = 7;  
std::cout << a;
```

main

a	3
b	Alias für a

Ausgabe:

b bekommt keinen eigenen Wert, nur einen Verweis auf a.

Referenzen tracen

```
int a = 3;  
int& b = a;  
b = 7;  
std::cout << a;
```

main

a	7
b	Alias für a

Ausgabe:

Schreiben über `b` ändert den Wert von `a`.

Referenzen tracen

```
int a = 3;  
int& b = a;  
b = 7;  
std::cout << a;
```

main

a	7
b	Alias für a

Ausgabe: 7

Pass by value

```
void foo(int i) {  
    i = 5;  
}  
  
int main() {  
    int i = 4;  
    foo(i);  
    std::cout << i << std::endl;  
}
```

main

i	4
---	---

Ausgabe:

Pass by value

```
void foo(int i) {  
    i = 5;  
}  
  
int main() {  
    int i = 4;  
    foo(i);  
    std::cout << i << std::endl;  
}
```

main

i	4
---	---

foo

i	4
---	---

Ausgabe:

Der Aufruf bekommt seine eigene Tabelle. Der Wert von *i* wird hineinkopiert.

Pass by value

```
void foo(int i) {  
    i = 5;  
}  
  
int main() {  
    int i = 4;  
    foo(i);  
    std::cout << i << std::endl;  
}
```

main

i	4
---	---

foo

i	5
---	---

Ausgabe:

foo ändert nur seine eigene Kopie.

Pass by value

```
void foo(int i) {  
    i = 5;  
}  
  
int main() {  
    int i = 4;  
    foo(i);  
    std::cout << i << std::endl;  
}
```

main

i	4
---	---

foo (fertig)

i	5
---	---

Ausgabe: 4

Ohne & heisst das *pass by value*: die Funktion arbeitet mit Kopien. So haben wir es bisher immer gemacht.

Pass by reference

```
void foo(int& i) {  
    i = 5;  
}  
  
int main() {  
    int i = 4;  
    foo(i);  
    std::cout << i << std::endl;  
}
```

main

i	4
---	---

Ausgabe:

Pass by reference

```
void foo(int& i) {  
    i = 5;  
}  
  
int main() {  
    int i = 4;  
    foo(i);  
    std::cout << i << std::endl;  
}
```

main

i	4
---	---

foo

i	Alias für i (main)
---	--------------------

Ausgabe:

Diesmal wird nichts kopiert: i in foo ist ein zweiter Name für i in main.

Pass by reference

```
void foo(int& i) {  
    i = 5;  
}  
  
int main() {  
    int i = 4;  
    foo(i);  
    std::cout << i << std::endl;  
}
```

main

i	5
---	---

foo

i	Alias für i (main)
---	--------------------

Ausgabe:

Die Änderung landet in der Tabelle von `main`.

Pass by reference

```
void foo(int& i) {  
    i = 5;  
}  
  
int main() {  
    int i = 4;  
    foo(i);  
    std::cout << i << std::endl;  
}
```

main

i	5
---	---

foo (fertig)

i	Alias für i (main)
---	--------------------

Ausgabe: 5

Mit & heisst das *pass by reference*: der Parameter wird ein Alias für das Argument.

Aufgabe “Referenz oder Kopie?”

```
int foo(int& a, int b) { // (a)
int foo(int a, int b) { // (b)
int foo(int a, int& b) { // (c)
    a += b;
    return a;
}

int main() {
    int a = 0;
    int b = 1;
    for (int i = 0; i < 5; ++i) {
        b = foo(a, b);
        std::cout << b << " ";
    }
    return 0;
}
```

Aufgabe

Was gibt das Programm aus, wenn foo die Signatur (a), (b) oder (c) hat?

5–8 Minuten alleine. Danach stellt jemand seine Lösung vor.

Prüfung Sommer 2015, Aufgabe 9

Lösung (a): `int foo(int& a, int b)`

```
int foo(int& a, int b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

a in foo ist ein Alias: a += b
ändert a in main.

i	a	b	Ausgabe
---	---	---	---------

Lösung (a): `int foo(int& a, int b)`

```
int foo(int& a, int b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

i	a	b	Ausgabe
0	1	1	1

a in foo ist ein Alias: a += b
ändert a in main.

Lösung (a): `int foo(int& a, int b)`

```
int foo(int& a, int b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

a in foo ist ein Alias: a += b
ändert a in main.

i	a	b	Ausgabe
0	1	1	1
1	2	2	1 2

Lösung (a): `int foo(int& a, int b)`

```
int foo(int& a, int b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

a in foo ist ein Alias: a += b
ändert a in main.

i	a	b	Ausgabe
0	1	1	1
1	2	2	1 2
2	4	4	1 2 4

Lösung (a): `int foo(int& a, int b)`

```
int foo(int& a, int b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

a in foo ist ein Alias: a += b
ändert a in main.

i	a	b	Ausgabe
0	1	1	1
1	2	2	1 2
2	4	4	1 2 4
3	8	8	1 2 4 8

Lösung (a): `int foo(int& a, int b)`

```
int foo(int& a, int b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

a in foo ist ein Alias: a += b
ändert a in main.

i	a	b	Ausgabe
0	1	1	1
1	2	2	1 2
2	4	4	1 2 4
3	8	8	1 2 4 8
4	16	16	1 2 4 8 16

Lösung (a): `int foo(int& a, int b)`

```
int foo(int& a, int b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

a in foo ist ein Alias: a += b
ändert a in main.

i	a	b	Ausgabe
0	1	1	1
1	2	2	1 2
2	4	4	1 2 4
3	8	8	1 2 4 8
4	16	16	1 2 4 8 16

Ausgabe: 1 2 4 8 16

Lösung (b): `int foo(int a, int b)`

```
int foo(int a, int b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

i	a	b	Ausgabe
---	---	---	---------

foo arbeitet nur mit Kopien. a in main bleibt 0, also gibt foo immer 0 + 1 zurück.

Lösung (b): `int foo(int a, int b)`

```
int foo(int a, int b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

foo arbeitet nur mit Kopien. a in main bleibt 0, also gibt foo immer 0 + 1 zurück.

i	a	b	Ausgabe
0	0	1	1

Lösung (b): `int foo(int a, int b)`

```
int foo(int a, int b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

foo arbeitet nur mit Kopien. a in main bleibt 0, also gibt foo immer 0 + 1 zurück.

i	a	b	Ausgabe
0	0	1	1
1	0	1	1 1
...	...	...	...
4	0	1	1 1 1 1 1

Lösung (b): `int foo(int a, int b)`

```
int foo(int a, int b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

`foo` arbeitet nur mit Kopien. `a` in `main` bleibt 0, also gibt `foo` immer `0 + 1` zurück.

i	a	b	Ausgabe
0	0	1	1
1	0	1	1 1
...	...	...	...
4	0	1	1 1 1 1 1
Ausgabe: 1 1 1 1 1			

Lösung (c): `int foo(int a, int& b)`

```
int foo(int a, int& b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

i	a	b	Ausgabe
---	---	---	---------

b ist zwar ein Alias, aber `foo` schreibt nur in seine Kopie von `a`. Also wie (b).

Lösung (c): `int foo(int a, int& b)`

```
int foo(int a, int& b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

i	a	b	Ausgabe
0	0	1	1

b ist zwar ein Alias, aber `foo` schreibt nur in seine Kopie von `a`. Also wie (b).

Lösung (c): `int foo(int a, int& b)`

```
int foo(int a, int& b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

b ist zwar ein Alias, aber `foo` schreibt nur in seine Kopie von `a`. Also wie (b).

i	a	b	Ausgabe
0	0	1	1
1	0	1	1 1
...	...	...	...
4	0	1	1 1 1 1 1

Lösung (c): `int foo(int a, int& b)`

```
int foo(int a, int& b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

b ist zwar ein Alias, aber `foo` schreibt nur in seine Kopie von `a`. Also wie (b).

i	a	b	Ausgabe
0	0	1	1
1	0	1	1 1
...	...	...	...
4	0	1	1 1 1 1 1
Ausgabe: 1 1 1 1 1			

Lösung (c): `int foo(int a, int& b)`

```
int foo(int a, int& b) {  
    a += b;  
    return a;  
}  
...  
for (int i = 0; i < 5; ++i) {  
    b = foo(a, b);  
    std::cout << b << " ";  
}
```

i	a	b	Ausgabe				
0	0	1	1				
1	0	1	1	1			
...	...	...	...				
4	0	1	1	1	1	1	1
			Ausgabe: 1 1 1 1 1				

`b` ist zwar ein Alias, aber `foo` schreibt nur in seine Kopie von `a`. Also wie (`b`).

💡 Ein `&` nützt nur bei Parametern, in die die Funktion auch schreibt.

Aufgabe “Swap” (falls Zeit bleibt)

Aufgabe

Schreibt eine Funktion `swap`, die die Werte von zwei `int`-Variablen vertauscht.

```
int a = 5;
int b = 6;
// here comes your function call
std::cout << a << "\n"; // outputs 6
std::cout << b << "\n"; // outputs 5
```

Skript, Aufgabe 95

Lösung "Swap"

```
// POST: the values of i and j
//       are swapped
void swap(int& i, int& j) {
    const int h = i;
    i = j;
    j = h;
}

int main() {
    int a = 5;
    int b = 6;
    swap(a, b);
    std::cout << a << "\n"; // 6
    std::cout << b << "\n"; // 5
}
```

Lösung "Swap"

```
// POST: the values of i and j
//       are swapped
void swap(int& i, int& j) {
    const int h = i;
    i = j;
    j = h;
}

int main() {
    int a = 5;
    int b = 6;
    swap(a, b);
    std::cout << a << "\n"; // 6
    std::cout << b << "\n"; // 5
}
```

main

a	6
b	5

swap (fertig)

i	Alias für a
j	Alias für b
h	5

Lösung "Swap"

```
// POST: the values of i and j
//       are swapped
void swap(int& i, int& j) {
    const int h = i;
    i = j;
    j = h;
}

int main() {
    int a = 5;
    int b = 6;
    swap(a, b);
    std::cout << a << "\n"; // 6
    std::cout << b << "\n"; // 5
}
```

main

a	6
b	5

swap (fertig)

i	Alias für a
j	Alias für b
h	5

Ohne & würde swap nur seine Kopien vertauschen.

Fragen/Unklarheiten?

7. Vektoren und Referenzen

std::vector: das Wichtigste

```
#include <vector>

std::vector<int> v = std::vector<int>(4, 0);
                                // 0 0 0 0
v[1] = 3;                       // 0 3 0 0
v.push_back(5);                 // 0 3 0 0 5
std::cout << std::ssize(v);    // 5

for (int i = 0; i < std::ssize(v); ++i) {
    std::cout << v[i] << " ";
}
```

std::vector: das Wichtigste

```
#include <vector>

std::vector<int> v = std::vector<int>(4, 0);
                                // 0 0 0 0
v[1] = 3;                       // 0 3 0 0
v.push_back(5);                 // 0 3 0 0 5
std::cout << std::ssize(v);    // 5

for (int i = 0; i < std::ssize(v); ++i) {
    std::cout << v[i] << " ";
}
```

- Indizes gehen von 0 bis $\text{std::ssize}(v) - 1$

std::vector: das Wichtigste

```
#include <vector>

std::vector<int> v = std::vector<int>(4, 0);
                                // 0 0 0 0
v[1] = 3;                       // 0 3 0 0
v.push_back(5);                 // 0 3 0 0 5
std::cout << std::ssize(v);    // 5

for (int i = 0; i < std::ssize(v); ++i) {
    std::cout << v[i] << " ";
}
```

- Indizes gehen von 0 bis $\text{std::ssize}(v) - 1$
- Ein Vektor kennt seine Länge: $\text{std::ssize}(v)$

std::vector: das Wichtigste

```
#include <vector>

std::vector<int> v = std::vector<int>(4, 0);
                                // 0 0 0 0
v[1] = 3;                       // 0 3 0 0
v.push_back(5);                 // 0 3 0 0 5
std::cout << std::ssize(v);    // 5

for (int i = 0; i < std::ssize(v); ++i) {
    std::cout << v[i] << " ";
}
```

- Indizes gehen von 0 bis $\text{std::ssize}(v) - 1$
- Ein Vektor kennt seine Länge: $\text{std::ssize}(v)$
- $v.at(7)$: prüft den Index, Fehler std::out_of_range

std::vector: das Wichtigste

```
#include <vector>

std::vector<int> v = std::vector<int>(4, 0);
                                // 0 0 0 0
v[1] = 3;                       // 0 3 0 0
v.push_back(5);                 // 0 3 0 0 5
std::cout << std::ssize(v);    // 5

for (int i = 0; i < std::ssize(v); ++i) {
    std::cout << v[i] << " ";
}
```

- Indizes gehen von 0 bis $\text{std::ssize}(v) - 1$
- Ein Vektor kennt seine Länge: $\text{std::ssize}(v)$
- $v.at(7)$: prüft den Index, Fehler std::out_of_range
- $v[7]$: prüft nichts, undefiniertes Verhalten

Live-Demo: “Reversing Vectors”

Code Example auf **code expert**. Der Code enthält *absichtlich* Fehler. Schaut vorerst nur, was PRE und POST versprechen:

Live-Demo: “Reversing Vectors”

Code Example auf **code expert**. Der Code enthält *absichtlich* Fehler. Schaut vorerst nur, was PRE und POST versprechen:

- `fill_increasing_sequence`: füllt einen Vektor mit 0, 1, 2, ...
- `swap`: vertauscht zwei `ints`, wie eben
- `reverse`: dreht die Folge um und gibt sie auf Wunsch aus
- `efficient_reverse_print`: noch leer, nicht einmal Parameter hat sie. Die schreiben wir zusammen
- `user_input_sequence`: liest eine Folge von der Eingabe
- `print_sequence`: gibt jedes Element auf einer eigenen Zeile aus

Teil 1: Folge einlesen

```
int user_input_sequence(std::vector<int> sequence){  
    // reads the length, sequence = std::vector<int>(length);, the numbers  
    ...  
}  
  
int main() {  
    int length;  
    std::vector<int> sequence;  
    length = user_input_sequence(sequence);  
    std::cout << "The size of sequence is: " << std::ssize(sequence) << std::endl;
```

Eingabe 3, dann 7 8 9. Ausgabe:

Teil 1: Folge einlesen

```
int user_input_sequence(std::vector<int> sequence){  
    // reads the length, sequence = std::vector<int>(length);, the numbers  
    ...  
}  
  
int main() {  
    int length;  
    std::vector<int> sequence;  
    length = user_input_sequence(sequence);  
    std::cout << "The size of sequence is: " << std::ssize(sequence) << std::endl;
```

Eingabe 3, dann 7 8 9. Ausgabe: The size of sequence is: 0. Warum stimmt die Länge nicht?

Teil 1: Folge einlesen

```
int user_input_sequence(std::vector<int> sequence){  
    // reads the length, sequence = std::vector<int>(length);, the numbers  
    ...  
}  
  
int main() {  
    int length;  
    std::vector<int> sequence;  
    length = user_input_sequence(sequence);  
    std::cout << "The size of sequence is: " << std::ssize(sequence) << std::endl;
```

Eingabe 3, dann 7 8 9. Ausgabe: The size of sequence is: 0. Warum stimmt die Länge nicht?

- `sequence` wird *by value* übergeben: die Funktion füllt eine Kopie, und die verschwindet am Ende der Funktion

Teil 1: Folge einlesen

```
int user_input_sequence(std::vector<int> sequence){  
    // reads the length, sequence = std::vector<int>(length);, the numbers  
    ...  
}  
  
int main() {  
    int length;  
    std::vector<int> sequence;  
    length = user_input_sequence(sequence);  
    std::cout << "The size of sequence is: " << std::ssize(sequence) << std::endl;
```

Eingabe 3, dann 7 8 9. Ausgabe: The size of sequence is: 0. Warum stimmt die Länge nicht?

- `sequence` wird *by value* übergeben: die Funktion füllt eine Kopie, und die verschwindet am Ende der Funktion
- Lösung: `int user_input_sequence(std::vector<int>& sequence)`

Teil 2: ausserhalb der Grenzen

```
// Prints each element in the sequence by using the random access operator '[]'.  
// Prints each element on a new line.  
void print_sequence(std::vector<int>& sequence){  
    for(int i = 1; i <= std::ssize(sequence); i++) {  
        std::cout << sequence[i] << std::endl;  
    }  
}  
  
int main() {  
    ...  
    length = fill_increasing_sequence(sequence, 4); // 0 1 2 3  
    print_sequence(sequence);  
}
```

Das Programm bricht mit einem Fehler ab. Warum? Und wie behebt man es?

Teil 2: Was ist schiefgelaufen?

Ausgabe: 1, 2, 3, dann der Fehler bei Index 4

Teil 2: Was ist schiefgelaufen?

Ausgabe: 1, 2, 3, dann der Fehler bei Index 4

- Gültig sind die Indizes 0 bis 3. Die Schleife liegt gleich zweimal daneben:
 - Sie beginnt bei 1: `sequence[0]` wird nie ausgegeben
 - `<=` geht bis 4, einen Schritt zu weit

Teil 2: Was ist schiefgelaufen?

Ausgabe: 1, 2, 3, dann der Fehler bei Index 4

- Gültig sind die Indizes 0 bis 3. Die Schleife liegt gleich zweimal daneben:
 - Sie beginnt bei 1: `sequence[0]` wird nie ausgegeben
 - `<=` geht bis 4, einen Schritt zu weit
- Lösung: `for (int i = 0; i < std::ssize(sequence); i++)`

Teil 2: Was ist schiefgelaufen?

Ausgabe: 1, 2, 3, dann der Fehler bei Index 4

- Gültig sind die Indizes 0 bis 3. Die Schleife liegt gleich zweimal daneben:
 - Sie beginnt bei 1: `sequence[0]` wird nie ausgegeben
 - `<=` geht bis 4, einen Schritt zu weit
- Lösung: `for (int i = 0; i < std::ssize(sequence); i++)`
- Dass [] hier abbricht, liegt an zwei Compiler-Flags dieser Aufgabe. Sonst ist das undefiniertes Verhalten: mein Mac gibt still 1 2 3 0 aus

Teil 3: rückwärts ausgeben

Aufgabe

Schreibt `efficient_reverse_print`: die Funktion gibt die Folge rückwärts aus, ohne sie zu verändern. Macht sie so schnell wie möglich!

2–3 Minuten alleine, auf Papier

Teil 3: rückwärts ausgeben

Aufgabe

Schreibt `efficient_reverse_print`: die Funktion gibt die Folge rückwärts aus, ohne sie zu verändern. Macht sie so schnell wie möglich!

2–3 Minuten alleine, auf Papier

Danach vergleichen wir die Zeit mit `reverse(sequence, length, true)`.

Die `chrono`-Zeilen messen nur die Zeit. Die müsst ihr für die Prüfung nicht kennen.

Mögliche Lösung

```
// POST: prints sequence in reverse without side effects
void efficient_reverse_print(const std::vector<int>& sequence) {
    for (int i = std::ssize(sequence) - 1; i >= 0; i--) {
        std::cout << sequence[i] << " ";
    }
    std::cout << std::endl;
}
```

Mögliche Lösung

```
// POST: prints sequence in reverse without side effects
void efficient_reverse_print(const std::vector<int>& sequence) {
    for (int i = std::ssize(sequence) - 1; i >= 0; i--) {
        std::cout << sequence[i] << " ";
    }
    std::cout << std::endl;
}
```

- &: der Vektor wird nicht kopiert. **const**: wir lesen nur

Mögliche Lösung

```
// POST: prints sequence in reverse without side effects
void efficient_reverse_print(const std::vector<int>& sequence) {
    for (int i = std::ssize(sequence) - 1; i >= 0; i--) {
        std::cout << sequence[i] << " ";
    }
    std::cout << std::endl;
}
```

- &: der Vektor wird nicht kopiert. **const**: wir lesen nur
- Keine Länge als Parameter: der Vektor kennt seine Länge

Mögliche Lösung

```
// POST: prints sequence in reverse without side effects
void efficient_reverse_print(const std::vector<int>& sequence) {
    for (int i = std::ssize(sequence) - 1; i >= 0; i--) {
        std::cout << sequence[i] << " ";
    }
    std::cout << std::endl;
}
```

- &: der Vektor wird nicht kopiert. **const**: wir lesen nur
- Keine Länge als Parameter: der Vektor kennt seine Länge
- Nichts umdrehen: es reicht, rückwärts durchzugehen

Mögliche Lösung

```
// POST: prints sequence in reverse without side effects
void efficient_reverse_print(const std::vector<int>& sequence) {
    for (int i = std::ssize(sequence) - 1; i >= 0; i--) {
        std::cout << sequence[i] << " ";
    }
    std::cout << std::endl;
}
```

- &: der Vektor wird nicht kopiert. **const**: wir lesen nur
- Keine Länge als Parameter: der Vektor kennt seine Länge
- Nichts umdrehen: es reicht, rückwärts durchzugehen

Auf meinem Mac: etwa 15 000 ns mit `reverse`, 10 000 ns so. Nur etwa 1.5-mal schneller, denn das meiste kostet die Ausgabe selbst.

Genauer hingeschaut

Warum `int` `i` und `std::ssize` und nicht das hier?

```
for (unsigned int i = sequence.size() - 1; i >= 0; i--) {  
    std::cout << sequence[i] << " ";  
}
```

Genauer hingeschaut

Warum `int` `i` und `std::ssize` und nicht das hier?

```
for (unsigned int i = sequence.size() - 1; i >= 0; i--) {  
    std::cout << sequence[i] << " ";  
}
```

- Ein `unsigned int` ist immer `>= 0`: die Bedingung ist nie `false`

Genauer hingeschaut

Warum `int` `i` und `std::ssize` und nicht das hier?

```
for (unsigned int i = sequence.size() - 1; i >= 0; i--) {  
    std::cout << sequence[i] << " ";  
}
```

- Ein `unsigned int` ist immer `>= 0`: die Bedingung ist nie `false`
- Nach `0` kommt durch `i--` der Überlauf auf $2^{32} - 1$: weit ausserhalb des Vektors

Genauer hingeschaut

Warum `int` `i` und `std::ssize` und nicht das hier?

```
for (unsigned int i = sequence.size() - 1; i >= 0; i--) {  
    std::cout << sequence[i] << " ";  
}
```

- Ein `unsigned int` ist immer `>= 0`: die Bedingung ist nie `false`
- Nach `0` kommt durch `i--` der Überlauf auf $2^{32} - 1$: weit ausserhalb des Vektors
- Auf meinem Mac: 4 3 2 1 0, dann Segmentation fault

Genauer hingeschaut

Warum `int` `i` und `std::ssize` und nicht das hier?

```
for (unsigned int i = sequence.size() - 1; i >= 0; i--) {  
    std::cout << sequence[i] << " ";  
}
```

- Ein `unsigned int` ist immer `>= 0`: die Bedingung ist nie `false`
- Nach `0` kommt durch `i--` der Überlauf auf $2^{32} - 1$: weit ausserhalb des Vektors
- Auf meinem Mac: 4 3 2 1 0, dann `Segmentation fault`

💡 `std::ssize` gibt die Länge als vorzeichenbehaftete Zahl zurück. Damit funktioniert `i >= 0`.

Teil 4: grosse Vektoren (falls Zeit bleibt)

In der Vorlage nimmt `reverse` den Vektor *by value*. Wir drehen einen Vektor mit 500 Millionen Elementen um, einmal so und einmal mit `&`:

```
void reverse(std::vector<int> sequence, int length, bool print) // template
void reverse(std::vector<int>& sequence, int length, bool print) // with &
```

Teil 4: grosse Vektoren (falls Zeit bleibt)

In der Vorlage nimmt `reverse` den Vektor *by value*. Wir drehen einen Vektor mit 500 Millionen Elementen um, einmal so und einmal mit `&`:

```
void reverse(std::vector<int> sequence, int length, bool print) // template
void reverse(std::vector<int>& sequence, int length, bool print) // with &
```

Übergabe	Zeit (mein Mac)
by value	≈ 2.9 s
by reference	≈ 1.1 s

Teil 4: grosse Vektoren (falls Zeit bleibt)

In der Vorlage nimmt `reverse` den Vektor *by value*. Wir drehen einen Vektor mit 500 Millionen Elementen um, einmal so und einmal mit `&`:

```
void reverse(std::vector<int> sequence, int length, bool print) // template
void reverse(std::vector<int>& sequence, int length, bool print) // with &
```

Übergabe	Zeit (mein Mac)
by value	≈ 2.9 s
by reference	≈ 1.1 s

Der Unterschied ist nur das Kopieren: 2 GB, bevor die Funktion überhaupt anfängt.

Nur lokal, nicht auf **code expert**!

Fragen/Unklarheiten?

8. Mehr zu Referenzen

Wozu Referenzen?

Was können wir mit Referenzen, was ohne nicht geht?

Wozu Referenzen?

Was können wir mit Referenzen, was ohne nicht geht?

- Mehr als ein Resultat aus einer Funktion herausgeben (**return** gibt nur eines)

Wozu Referenzen?

Was können wir mit Referenzen, was ohne nicht geht?

- Mehr als ein Resultat aus einer Funktion herausgeben (**return** gibt nur eines)
- Teures Kopieren sparen (Teil 4 eben)

Wozu Referenzen?

Was können wir mit Referenzen, was ohne nicht geht?

- Mehr als ein Resultat aus einer Funktion herausgeben (**return** gibt nur eines)
- Teures Kopieren sparen (Teil 4 eben)
- Mit Dingen arbeiten, die man gar nicht kopieren kann, z.B. `std::cout`

Beispiele

Code Example “Quadratic Equation”: zwei Lösungen

```
// POST: return value is the number of distinct real solutions of
//        $ax^2 + bx + c = 0$  (-1 if there are infinitely many). The
//       solutions are written to x1, x2 (if existent)
int solve_quadratic_equation(const double a, const double b, const double c,
                             double& x1, double& x2)
```

Beispiele

Code Example “Quadratic Equation”: zwei Lösungen

```
// POST: return value is the number of distinct real solutions of
//       ax^2 + bx + c = 0 (-1 if there are infinitely many). The
//       solutions are written to x1, x2 (if existent)
int solve_quadratic_equation(const double a, const double b, const double c,
                             double& x1, double& x2)
```

Es gibt nur einen Ausgabestrom, den kann man nicht kopieren:

```
int a = 5;
int b = a;           // making a copy of an int
std::ostream o = std::cout; // error: copying std::cout is impossible
std::ostream& o = std::cout; // this works
```

Streams müsst ihr nicht im Detail verstehen. Es reicht zu wissen, dass es so etwas gibt.

const-Referenzen

```
double a = 3.0;
double& b = a;
const double& c = a;

c = 4.0;           // error: read-only
a = 5.0;           // fine
b = 6.0;           // fine
std::cout << c;    // 6, reading is fine

double& d = c;      // error: drops const
const double& e = 5.0; // fine for const refs
```

const-Referenzen

```
double a = 3.0;
double& b = a;
const double& c = a;

c = 4.0;           // error: read-only
a = 5.0;           // fine
b = 6.0;           // fine
std::cout << c;    // 6, reading is fine

double& d = c;      // error: drops const
const double& e = 5.0; // fine for const refs
```

- Über eine **const**-Referenz darf man nur lesen, nicht schreiben

const-Referenzen

```
double a = 3.0;
double& b = a;
const double& c = a;

c = 4.0;           // error: read-only
a = 5.0;           // fine
b = 6.0;           // fine
std::cout << c;    // 6, reading is fine

double& d = c;      // error: drops const
const double& e = 5.0; // fine for const refs
```

- Über eine **const**-Referenz darf man nur lesen, nicht schreiben
- Typisch für Parameter: nicht kopieren, aber auch nicht verändern

const-Referenzen

```
double a = 3.0;
double& b = a;
const double& c = a;

c = 4.0;           // error: read-only
a = 5.0;           // fine
b = 6.0;           // fine
std::cout << c;    // 6, reading is fine

double& d = c;      // error: drops const
const double& e = 5.0; // fine for const refs
```

- Über eine **const**-Referenz darf man nur lesen, nicht schreiben
- Typisch für Parameter: nicht kopieren, aber auch nicht verändern
- Kommt gleich bei der Matrix wieder:
const imatrix&

Referenzen als Rückgabetyt

Ausgabe:

```
int& increment(int& m) {  
    return ++m;  
}  
  
int main() {  
    int n = 3;  
    increment(increment(n));  
    std::cout << n << "\n";  
    return 0;  
}
```

Referenzen als Rückgabetyt

Ausgabe: 5

```
int& increment(int& m) {  
    return ++m;  
}  
  
int main() {  
    int n = 3;  
    increment(increment(n));  
    std::cout << n << "\n";  
    return 0;  
}
```

Referenzen als Rückgabotyp

```
int& increment(int& m) {  
    return ++m;  
}  
  
int main() {  
    int n = 3;  
    increment(increment(n));  
    std::cout << n << "\n";  
    return 0;  
}
```

Ausgabe: 5

Das innere `increment(n)` gibt `n` selbst zurück (einen L-Wert), nicht eine Kopie. Das äussere erhöht also nochmals `n`.

Referenzen als Rückgabotyp

```
int& increment(int& m) {  
    return ++m;  
}  
  
int main() {  
    int n = 3;  
    increment(increment(n));  
    std::cout << n << "\n";  
    return 0;  
}
```

Ausgabe: 5

Das innere `increment(n)` gibt `n` selbst zurück (einen L-Wert), nicht eine Kopie. Das äussere erhöht also nochmals `n`.

Genau so ist der Prä-Inkrement-Operator `++i` gebaut. Dazu in ein paar Wochen mehr.

Referenzen als Rückgabotyp

```
int& increment(int& m) {  
    return ++m;  
}  
  
int main() {  
    int n = 3;  
    increment(increment(n));  
    std::cout << n << "\n";  
    return 0;  
}
```

Ausgabe: 5

Das innere `increment(n)` gibt `n` selbst zurück (einen L-Wert), nicht eine Kopie. Das äussere erhöht also nochmals `n`.

Genau so ist der Prä-Inkrement-Operator `++i` gebaut. Dazu in ein paar Wochen mehr.

💡 Gebt nie eine Referenz auf eine lokale Variable zurück: die ist nach dem `return` weg.

Fragen/Unklarheiten?

9. Multidimensionale Vektoren

Vektoren von Vektoren

```
using irow = std::vector<int>;    // one row
using imatrix = std::vector<irow>; // rows

imatrix m(2, irow(3, 0)); // 2 x 3, all 0
m[1][2] = 7;

irow second = m[1];           // 0 0 7
std::ssize(m)                 // 2 rows
std::ssize(m[0])              // 3 columns
```

Vektoren von Vektoren

```
using irow = std::vector<int>;    // one row
using imatrix = std::vector<irow>; // rows

imatrix m(2, irow(3, 0)); // 2 x 3, all 0
m[1][2] = 7;

irow second = m[1];    // 0 0 7
std::ssize(m)          // 2 rows
std::ssize(m[0])       // 3 columns
```

	[0]	[1]	[2]
m[0]	0	0	0
m[1]	0	0	7

Vektoren von Vektoren

```
using irow = std::vector<int>;    // one row
using imatrix = std::vector<irow>; // rows

imatrix m(2, irow(3, 0)); // 2 x 3, all 0
m[1][2] = 7;

irow second = m[1];           // 0 0 7
std::ssize(m)                 // 2 rows
std::ssize(m[0])              // 3 columns
```

	[0]	[1]	[2]
m[0]	0	0	0
m[1]	0	0	7

- irow: ein Vektor von Zahlen, eine Zeile
- imatrix: ein Vektor von Zeilen
- m[r][c]: Zeile r, Spalte c

Aufgabe “Matrix Transpose”

Aufgabe

Schreibt eine **Funktion**, die eine Matrix von `ints` nimmt und die transponierte Matrix zurückgibt. Das Programm liest eine Matrix ein, transponiert sie und gibt sie aus. Auf der ersten Zeile stehen die Anzahl Zeilen r und Spalten c , danach r Zeilen mit je c Zahlen.

Eingabe

```
2 3
1 2 3
4 5 6
```

Ausgabe

```
3 2
1 4
2 5
3 6
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Die Vorlage I: Matrix Zeile für Zeile einlesen

```
// POST: returns a matrix read from standard input
imatrix read_matrix() {
    int r, c;
    std::cin >> r >> c;
    imatrix matrix = imatrix(0); // zero rows
    for (int row_index = 0; row_index < r; row_index++) {
        irow row = irow(0); // zero entries
        for (int col_index = 0; col_index < c; col_index++) {
            int element;
            std::cin >> element;
            row.push_back(element);
        }
        matrix.push_back(row);
    }
    return matrix;
}
```

Die Vorlage I: Matrix Zeile für Zeile einlesen

```
// POST: returns a matrix read from standard input
imatrix read_matrix() {
    int r, c;
    std::cin >> r >> c;
    imatrix matrix = imatrix(0); // zero rows
    for (int row_index = 0; row_index < r; row_index++) {
        irow row = irow(0); // zero entries
        for (int col_index = 0; col_index < c; col_index++) {
            int element;
            std::cin >> element;
            row.push_back(element);
        }
        matrix.push_back(row);
    }
    return matrix;
}
```

- Die Matrix startet ohne Zeilen

Die Vorlage I: Matrix Zeile für Zeile einlesen

```
// POST: returns a matrix read from standard input
imatrix read_matrix() {
    int r, c;
    std::cin >> r >> c;
    imatrix matrix = imatrix(0); // zero rows
    for (int row_index = 0; row_index < r; row_index++) {
        irow row = irow(0); // zero entries
        for (int col_index = 0; col_index < c; col_index++) {
            int element;
            std::cin >> element;
            row.push_back(element);
        }
        matrix.push_back(row);
    }
    return matrix;
}
```

- Die Matrix startet ohne Zeilen
- Jede Zeile startet leer und wächst mit `push_back`

Die Vorlage I: Matrix Zeile für Zeile einlesen

```
// POST: returns a matrix read from standard input
imatrix read_matrix() {
    int r, c;
    std::cin >> r >> c;
    imatrix matrix = imatrix(0); // zero rows
    for (int row_index = 0; row_index < r; row_index++) {
        irow row = irow(0); // zero entries
        for (int col_index = 0; col_index < c; col_index++) {
            int element;
            std::cin >> element;
            row.push_back(element);
        }
        matrix.push_back(row);
    }
    return matrix;
}
```

- Die Matrix startet ohne Zeilen
- Jede Zeile startet leer und wächst mit `push_back`
- Die fertige Zeile wird hinten an die Matrix angehängt

Die Vorlage II: Matrix auf einmal anlegen

```
// POST: returns a matrix read from standard input
imatrix read_matrix() {
    int r, c;
    std::cin >> r >> c;
    imatrix matrix(r, irow(c, 0)); // r x c zeros
    for (int row_index = 0; row_index < r; row_index++) {
        for (int col_index = 0; col_index < c; col_index++) {
            int element;
            std::cin >> element;
            matrix[row_index][col_index] = element;
        }
    }
    return matrix;
}
```

Die Vorlage II: Matrix auf einmal anlegen

```
// POST: returns a matrix read from standard input
imatrix read_matrix() {
    int r, c;
    std::cin >> r >> c;
    imatrix matrix(r, irow(c, 0)); // r x c zeros
    for (int row_index = 0; row_index < r; row_index++) {
        for (int col_index = 0; col_index < c; col_index++) {
            int element;
            std::cin >> element;
            matrix[row_index][col_index] = element;
        }
    }
    return matrix;
}
```

- r Zeilen, jede eine Kopie von `irow(c, 0)`

Die Vorlage II: Matrix auf einmal anlegen

```
// POST: returns a matrix read from standard input
imatrix read_matrix() {
    int r, c;
    std::cin >> r >> c;
    imatrix matrix(r, irow(c, 0)); // r x c zeros
    for (int row_index = 0; row_index < r; row_index++) {
        for (int col_index = 0; col_index < c; col_index++) {
            int element;
            std::cin >> element;
            matrix[row_index][col_index] = element;
        }
    }
    return matrix;
}
```

- r Zeilen, jede eine Kopie von `irow(c, 0)`
- Die Matrix hat von Anfang an die richtige Grösse, danach weisen wir nur noch zu

Die Vorlage II: Matrix auf einmal anlegen

```
// POST: returns a matrix read from standard input
imatrix read_matrix() {
    int r, c;
    std::cin >> r >> c;
    imatrix matrix(r, irow(c, 0)); // r x c zeros
    for (int row_index = 0; row_index < r; row_index++) {
        for (int col_index = 0; col_index < c; col_index++) {
            int element;
            std::cin >> element;
            matrix[row_index][col_index] = element;
        }
    }
    return matrix;
}
```

- r Zeilen, jede eine Kopie von `irow(c, 0)`
- Die Matrix hat von Anfang an die richtige Grösse, danach weisen wir nur noch zu
- Ausserdem in der Vorlage: `get_rows`, `get_cols`, `write_matrix`

Mal schauen, was ihr gelernt habt

- Öffnet “Matrix Transpose” auf **code expert**

Mal schauen, was ihr gelernt habt

- Öffnet “Matrix Transpose” auf **code expert**
- Überlegt euch alleine, wie ihr das mit Stift und Papier lösen würdet (5 Minuten)

Mal schauen, was ihr gelernt habt

- Öffnet “Matrix Transpose” auf **code expert**
- Überlegt euch alleine, wie ihr das mit Stift und Papier lösen würdet (5 Minuten)
- Programmiert die Lösung zu zweit (10 Minuten)

Mal schauen, was ihr gelernt habt

- Öffnet “Matrix Transpose” auf **code expert**
- Überlegt euch alleine, wie ihr das mit Stift und Papier lösen würdet (5 Minuten)
- Programmiert die Lösung zu zweit (10 Minuten)
- Danach diktiert ihr mir eure Lösung (5 Minuten)

Mal schauen, was ihr gelernt habt

- Öffnet “Matrix Transpose” auf **code expert**
 - Überlegt euch alleine, wie ihr das mit Stift und Papier lösen würdet (5 Minuten)
 - Programmiert die Lösung zu zweit (10 Minuten)
 - Danach diktiert ihr mir eure Lösung (5 Minuten)
- 💡 Das Element in Zeile i , Spalte j landet in Zeile j , Spalte i .

Mögliche Lösung I: Zeile für Zeile

```
// POST: returns the transpose of matrix
imatrix transpose_matrix(const imatrix& matrix) {
    int r = get_rows(matrix);
    int c = get_cols(matrix);
    imatrix transposed = imatrix(0);
    for (int col_index = 0; col_index < c; col_index++) {
        irow row = irow(0);
        for (int row_index = 0; row_index < r; row_index++) {
            row.push_back(matrix[row_index][col_index]);
        }
        transposed.push_back(row);
    }
    return transposed;
}
```

Mögliche Lösung I: Zeile für Zeile

```
// POST: returns the transpose of matrix
imatrix transpose_matrix(const imatrix& matrix) {
    int r = get_rows(matrix);
    int c = get_cols(matrix);
    imatrix transposed = imatrix(0);
    for (int col_index = 0; col_index < c; col_index++) {
        irow row = irow(0);
        for (int row_index = 0; row_index < r; row_index++) {
            row.push_back(matrix[row_index][col_index]);
        }
        transposed.push_back(row);
    }
    return transposed;
}
```

- Spalte j der Matrix wird Zeile j des Resultats

Mögliche Lösung I: Zeile für Zeile

```
// POST: returns the transpose of matrix
imatrix transpose_matrix(const imatrix& matrix) {
    int r = get_rows(matrix);
    int c = get_cols(matrix);
    imatrix transposed = imatrix(0);
    for (int col_index = 0; col_index < c; col_index++) {
        irow row = irow(0);
        for (int row_index = 0; row_index < r; row_index++) {
            row.push_back(matrix[row_index][col_index]);
        }
        transposed.push_back(row);
    }
    return transposed;
}
```

- Spalte j der Matrix wird Zeile j des Resultats
- Darum läuft die äussere Schleife über die Spalten

Mögliche Lösung I: Zeile für Zeile

```
// POST: returns the transpose of matrix
imatrix transpose_matrix(const imatrix& matrix) {
    int r = get_rows(matrix);
    int c = get_cols(matrix);
    imatrix transposed = imatrix(0);
    for (int col_index = 0; col_index < c; col_index++) {
        irow row = irow(0);
        for (int row_index = 0; row_index < r; row_index++) {
            row.push_back(matrix[row_index][col_index]);
        }
        transposed.push_back(row);
    }
    return transposed;
}
```

- Spalte j der Matrix wird Zeile j des Resultats
- Darum läuft die äussere Schleife über die Spalten
- **const** imatrix&: nicht kopieren, nicht verändern

Mögliche Lösung II: auf einmal

```
// POST: returns the transpose of matrix
imatrix transpose_matrix(const imatrix& matrix) {
    int r = get_rows(matrix);
    int c = get_cols(matrix);
    imatrix transposed(c, irow(r, 0)); // c x r zeros
    for (int row_index = 0; row_index < r; row_index++) {
        for (int col_index = 0; col_index < c; col_index++) {
            transposed[col_index][row_index] =
                matrix[row_index][col_index];
        }
    }
    return transposed;
}
```

Mögliche Lösung II: auf einmal

```
// POST: returns the transpose of matrix
imatrix transpose_matrix(const imatrix& matrix) {
    int r = get_rows(matrix);
    int c = get_cols(matrix);
    imatrix transposed(c, irow(r, 0)); // c x r zeros
    for (int row_index = 0; row_index < r; row_index++) {
        for (int col_index = 0; col_index < c; col_index++) {
            transposed[col_index][row_index] =
                matrix[row_index][col_index];
        }
    }
    return transposed;
}
```

- Das Resultat ist $c \times r$ statt $r \times c$

Mögliche Lösung II: auf einmal

```
// POST: returns the transpose of matrix
imatrix transpose_matrix(const imatrix& matrix) {
    int r = get_rows(matrix);
    int c = get_cols(matrix);
    imatrix transposed(c, irow(r, 0)); // c x r zeros
    for (int row_index = 0; row_index < r; row_index++) {
        for (int col_index = 0; col_index < c; col_index++) {
            transposed[col_index][row_index] =
                matrix[row_index][col_index];
        }
    }
    return transposed;
}
```

- Das Resultat ist $c \times r$ statt $r \times c$
- Hier sieht man das Vertauschen der Indizes direkt

Mögliche Lösung II: auf einmal

```
// POST: returns the transpose of matrix
imatrix transpose_matrix(const imatrix& matrix) {
    int r = get_rows(matrix);
    int c = get_cols(matrix);
    imatrix transposed(c, irow(r, 0)); // c x r zeros
    for (int row_index = 0; row_index < r; row_index++) {
        for (int col_index = 0; col_index < c; col_index++) {
            transposed[col_index][row_index] =
                matrix[row_index][col_index];
        }
    }
    return transposed;
}
```

- Das Resultat ist $c \times r$ statt $r \times c$
- Hier sieht man das Vertauschen der Indizes direkt

get_cols der Vorlage gibt für eine leere Matrix 0 zurück: die hat keine Zeile matrix[0].

Fragen/Unklarheiten?

10. Outro

Vorlage für Cheat Sheets

- Meine $\text{\LaTeX}$ -Vorlage für dichte Cheat Sheets:
A4 quer, fünf Spalten
- Läuft direkt in Overleaf, nichts zu installieren
- Frei verwendbar (MIT-0)

 Cheat Sheet

`adelgavranovic.ch/cheatsheet`



Allgemeine Fragen?

Gibt es noch offene Fragen zu dem, was wir bisher in den Übungsstunden gemacht haben?

Bis nächste Woche!