Programmierkurs C/C++

"Dienstag"

Sandro Andreotti
andreott@inf.fu-berlin.de
(Slides: Andreas Döring)

WS 2009/10

Ein Quiz zum Warmwerden

Was gibt folgendes Programm aus?

```
#include <cstdio>
int main()
{
    int i = 1;
    int j = 0;
    switch (i)
    {
          case '1': ++j;
          case '2': j += 2;
          default: j += 3;
    }
    printf("%i", j);
    return 0;
}
```

Vorsicht Falle!

Antwort: 3.

Grund: Die Fälle ‘1’ und ‘2’ werden nicht genommen, weil wegen der Anführungszeichen ja ‘1’ == 49 und ‘2’ == 50 ist, somit beides ungleich i ist.

Mehrere Dateien

Ein C/C++-Projekt kann aus mehreren Dateien bestehen:

- .cpp-Datei: Quelltextdatei.
  (bei C: .c-Dateien)
- .h-Dateien: Headerdateien.
  (bei C++: Auch .hpp)

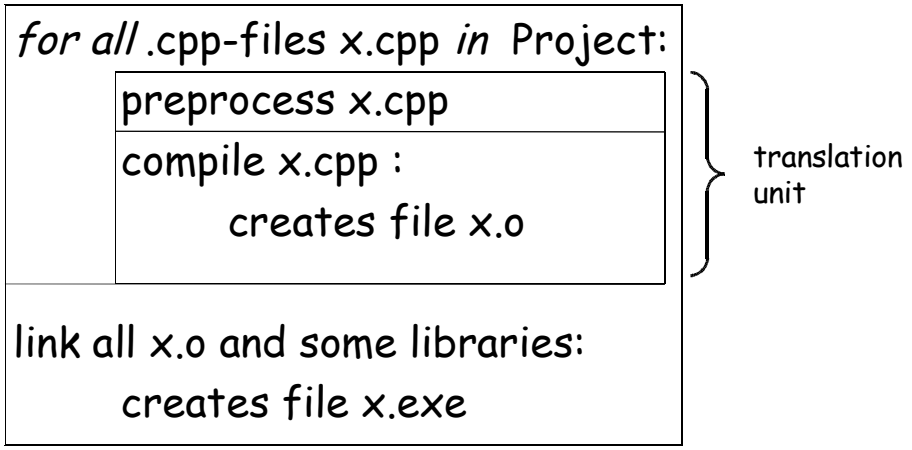
Header-Dateien enthalten im Prinzip ebenfalls Quelltext; sie werden jedoch anders behandelt als .c-Dateien. (Siehe unten).

Manche Leute nennen die Headerdateien unter C++ auch .hpp.

Unter Linux/Unix wird für C++-Dateien auch .C (großes „C“) als Endung verwendet.

Compilieren: So geht das

Ein Projekt wird nach folgendem Schema compiliert:



Man beachte, dass in der „for all“-Schleife nur über die .cpp-Dateien, nicht aber über die .h-Dateien iteriert wird. Der Präprozessor (wir verwenden im Folgenden die anglisierte Form „Preprocessor“) ist u.A. für die Einbindung der Header-Dateien zuständig (siehe unten). Header-Dateien dienen somit dazu, um von .cpp-Dateien (oder von anderen Header-Dateien) eingebunden zu werden.

Die .o-Dateien („Objektdateien“) enthalten fertig compilierte Programmstücke. Zusammen mit Programmstücken, die sich in Libraries befinden, wird durch den Linker das fertige ausführbare Programm zusammen „gelinkt“. Der Gesamte Vorgang (Preprocessing, Compilieren und Linken) wird kann gewöhnlich durch einen einzigen Aufruf des Compilers durchgeführt werden. Eine Alternative wäre es jedoch, Compilieren und Linken getrennt voneinander auszuführen, und die Organisation darüber z.B. einem Makefile anzuvertrauen (Tool: „make“).

Der Preprocessor

- Befehle für den Preprocessor beginnen mit #.
- #include fügt eine Header-Datei ein.

```
#include <stdio.h>
```

- #define definiert ein Macro: Das Schlüsselwort wird im ganzen Programm durch den angegebenen Text ersetzt.

```
#define PI 3.141596
```

Preprocessor-Befehle enden durch den nächsten Zeilenumbruch.

Man beachte: Der Preprocessor manipuliert den Quelltext *als Text*, ohne dabei die C++-Syntax zu berücksichtigen. Mit **#include** wird der Inhalt einer Datei textuell eingefügt, mit **#define** wird im ganzen folgenden Text ein String durch einen anderen String ersetzt.

Einem **#define** kann man auch Argumente übergeben. Bsp:

#define AUSGABE(a,b) printf("%s ist %i\n“, a, b)

Nun wird

AUSGABE(“meine Lieblingszahl”, 42);

in

printf(“%s ist %i\n”, “meine Lieblingszahl”, 42);

übersetzt.

Anmerkung: Es handelt sich um eine einfache Textersetzung. Macros können Texte erzeugen, die andere Macros enthalten, rekursive Macros sind jedoch nicht möglich.

Anmerkung: Es gibt in C++ inzwischen deutlich bessere Konstrukte, die **#define** in den meisten Fällen ersetzen können.

Verteilen von Quelltext

Jede translation unit wird getrennt übersetzt.

Aufgabe: Wir haben ein Programm auf mehrere Dateien verteilt: In `teil1.cpp` und `teil2.cpp` wird jeweils die Funktion `max(x, y)` verwendet.

Problem: Wo wird `max(x, y)` definiert?

Ansatz: In einer Header-Datei, die jeweils von `teil1.cpp` und von `teil2.cpp` eingebunden wird.

One Definition Rule

Regel: » Jede Entität (Variable, Typ, Funktion, Klasse) darf nur einmal definiert werden. «

Problem: Bei unserem Ansatz würde `max(x,y)` mehrfach definiert werden.

Dies gilt für beide Ebenen:

1. Auf der Ebene der Translation Unit: nur eine Definition pro Translation Unit!
2. Auf der Ebene des Linkers: nur eine Definition pro Projekt!

Der Grund für die ODR ist, dass der Compiler (in Fall 2) bzw. der Linker (in Fall 1) bei mehrfacher Definition nicht entscheiden könnte, welche der Definitionen er wählen sollte. Durch die ODR spart sich der Compiler/Linker viele Probleme, die ohne die ODR zu lösen wären.

Wenn man die selbe Entität in verschiedenen Translation Units definiert, dort jeweils nur einmal, und würde dann gegen 2. verstoßen. Andererseits ist es bei Klassen (die nur in C++ vorkommen) durchaus erlaubt, diese in jeder Translation Unit erneut zu definieren, da bei der bloßen Definition einer Klasse keine Entität erzeugt wird, mit der der Linker noch etwas zu tun hätte.

Man beachte: Es ist durchaus möglich, verschiedene Entitäten mit gleichem Namen zu erzeugen (z.B. verschiedene lokale Variablen mit gleichem Namen). Hieraus ergeben sich aber keine Probleme bezüglich der ODR.

Deklaration & Definition

Deklaration von X =

Beschreibung, was X ist

Definition von X =

Deklaration von X

\+ Erzeugung von X

ODR: » X darf beliebig oft deklariert, aber nur einmal definiert werden. «

Mit Headerdateien können Symbole, die in einer .c-Datei definiert worden sind, auch anderen .cpp-Dateien mitgeteilt werden. Die anderen .cpp-Dateien benötigen dazu lediglich die Deklarationen für diese Symbole. Allerdings sollten Headerdateien nur Deklarationen und keine Definitionen enthalten, da sonst jede .cpp Datei ein neues X definieren würde.

Man beachte jedoch, dass durchaus der gleiche Bezeichner jeweils in verschiedenen { }-Blöcken als lokales Symbol definiert werden darf. Auch können die gleichen Bezeichner in unterschiedlichen namespaces jeweils für etwas anderes stehen.

Wie Definitionen und Deklarationen im Einzelnen aussehen, wird auf den folgenden Folien beschrieben.

Deklaration & Definition (2)

Deklaration **Definition**

- Variablen:

```
extern int x;
```

```
int x;
```

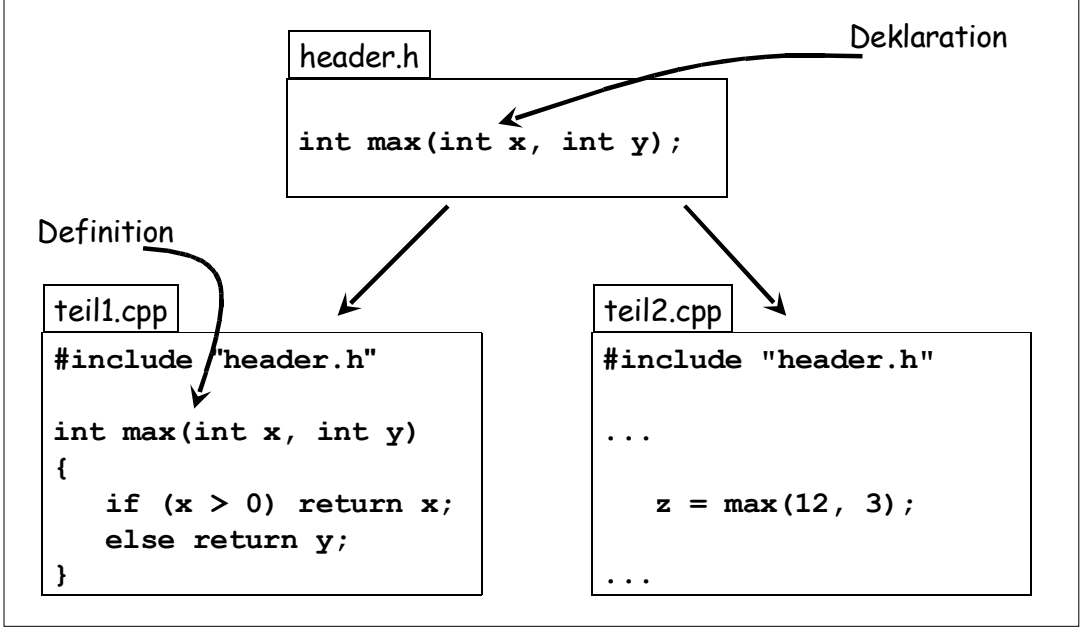
- Funktionen:

```
int add(int, int);
```

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

Defaultwerte für Variablen dürfen immer nur bei der Definition angegeben werden.

Verteilen von Quelltext (2)



Für die Benutzung der Funktion **max(x,y)** in **teil2.cpp** wird vom Compiler zunächst lediglich die Deklaration von **max(x,y)** benötigt. Diese kann man der Bequemlichkeit halber in einen Header schreiben (**header.h**), den man immer dann einbindet, wenn die Funktion verwendet werden soll. In einer der *translation units* (oder in einer Library) muss **max(x,y)** dann aber auch definiert sein; hier wird das durch **teil1.cpp** erledigt.

Nochmals zur Verdeutlichung: In der *translation unit* von **teil2.cpp** wird **max(x,y)** also verwendet, ohne dass **max(x,y)** dort definiert wird! Der Compiler muss nämlich gar nicht wissen, was **max(x,y)** tut, sondern nur, wie man **max(x,y)** aufruft, und das erfährt er über die Deklaration von **max(x,y)**. Der Linker sorgt am Ende dafür, dass tatsächlich die richtige Funktion aufgerufen wird.

Gültigkeit von Deklarationen

Regel 1:

»Ein Symbol ist im Programmtext nur unterhalb seiner Deklaration bekannt.«

Regel 2:

»Eine Deklaration, die innerhalb eines { } -Blocks steht, gilt auch nur lokal innerhalb dieses Blocks.«

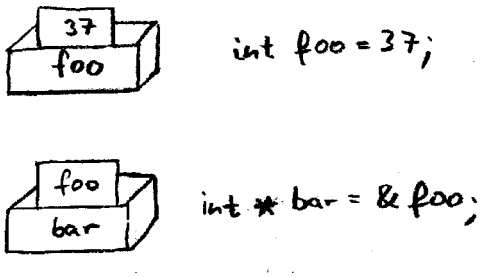
Zur Erinnerung: Pointer

Zeiger sind Variablen, deren Inhalt auf andere Variablen verweist.

```
int * X;
float * Y;
```

⇦ Zeiger auf eine int Variable

⇦ Zeiger auf eine float Variable



Mit Pointern werden wir uns im nächsten Kapitel noch eingehender beschäftigen.

Der *address-of*-Operator &

Pointer speichern Adressen von Variablen ab.

```
int foo;
int * bar = & foo;
```

⇐ Variable foo

⇐ & gibt Adresse von foo, die dann in bar gespeichert wird.

Mit dem address-of-Operator & kann man die Adresse einer Variablen bestimmen.

Der address-of-Operator ist ein unärer Operator. & als binärer Operator steht für bitweises UND.

Man verwechsele den address-of-Operator auch nicht mit dem &, das bei der Definition von Referenzen (siehe unten) verwendet wird!

Wenn eine Variable **X** vom Typ **t** ist, dann ist **& X** vom Typ **t ***.

Der *indirection*-Operator *

Das Gegenteil zum address-of-Operator & ist der indirection-Operator *:

```
int foo = 37;
int * bar = & foo;
int foo2 = * bar;
```

⇐ foo enthält 37.
⇐ bar zeigt auf foo.
⇐ foo2 enthält auch 37.

Mit dem indirection-Operator kann man den Inhalt einer Variablen bestimmen, wenn man deren Adresse hat.

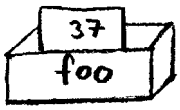
Der indirection-Operator * wird auch „Dereferenzierungs-Operator“ genannt. Er ist ein unärer Operator, als binärer Operator wird * für die Multiplikation verwendet.

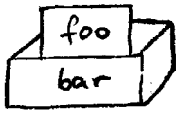
Man verwechsele den indirection-Operator nicht mit dem *, der bei der Deklaration/Definition von Pointervariablen verwendet wird!

Wenn man sich Pointer als Pfeile vorstellt, dann bedeutet der *-Operator, dass man dem Pfeil folgt und sich das Ziel anschaut, auf das der Pfeil zeigt.

* bar ≡ foo

```
int foo = 37;
int * bar = & foo;
```





X1 = foo; ⇦ **X1** = „Inhalt" von foo
= 37.

X2 = * bar; ⇦ **X2** = „Inhalt" von * bar
= „Inhalt" von foo
= 37.

Anwendung: Rückgabe in Argumenten

Beispiel für eine Funktion mit mehr als einem Rückgabewert:

```
void nimmZwei(int * X, int * Y)
{
    *X = 1;
    *Y = 2;
}

int A, B;
nimmZwei(&A, &B);
```

Exkurs: Referenzen

Alternative: Mit Referenzen

C++

```
void nimmZwei(int & X, int & Y)
{
    X = 1;
    Y = 2;
}

int A, B;
nimmZwei(A, B);
```

Referenzen sind technisch gesehen Pointer, die einmal bei der Initialisierung auf eine feste Zielvariable eingestellt und danach nicht mehr umgebogen werden können, und die sich anschließend so verhalten, als wären sie die Variable, auf die sie zeigen.

Exkurs: Referenzen (2)

„Referenzen sind Pointer, die so tun, als wären sie keine."

C++

Referenzen

```
int i = 10;
int & r = i;
r = 20;
```

==

Pointer

```
int i = 10;
int * p = & i;
* p = 20;
```

(Ergebnis in beiden Fällen: i == 20)

Da bei Referenzen sichergestellt ist, dass sie immer auf eine existierende Variable zeigen, sind Referenzen sicherer als Pointer. Außerdem kann man sich bei Referenzen die Benutzung des Operators * sparen.

Nullpointer

Ein Pointer kann auch eine 0 enthalten:

```
int * bar = 0;
int foo = *bar; //boeser Fehler!
```

In C ist es auch üblich, statt **NULL** statt 0 zu schreiben.

Der indirection-Operator geht davon aus, dass in der Pointer-Variablen die Adresse einer Variablen steht. Wird er auf einen Nullpointer angewendet, so gibt der indirection-Operator also eine Variable wieder, die angeblich an der Adresse 0 liegt. Der Speicher dort ist aber niemals zum Lesen freigegeben, versucht man das dennoch, so wird die Programmausführung abgeprochen.

Ähnliches gilt natürlich auch für alle anderen Speicheradressen, hinter denen keine lesbaren Speicherbereiche liegen.

Der Compiler hilft einem nicht dabei, diesen Fehler zu vermeiden. Man muss eben darauf aufpassen, dass auch etwas Sinnvolles in einem Pointer steht, bevor man auf ihn den indirection-Operator anwendet.

Anmerkung: Trotzdem kann es sinnvoll sein, einen Nullpointer zu verwenden – soll ein Zeiger auf „nichts“ zeigen, kennzeichnet man dies, indem man eine 0 hineinschreibt.

Übungsaufgabe: Schaffen Sie es, eine Referenzvariable auf die Adresse 0 zeigen zu lassen?

void-Pointer

Ein void-Pointer zeigt auf etwas von unbestimmtem Typ:

```
int x;
void * p = & x;

float y;
p = & y;
```

Das Symbol **NULL** ist in C definiert als ein Nullpointer vom Typ **void ***.

`void *` weiß zwar wo, aber nicht was

Ein void * speichert eine Adresse, weiß aber nicht, was sich an dieser Adresse befindet.

```
int i = 0;
void * p = & i;
int j = * p;
```

⇦ Fehler!
p weiß nicht, worauf er zeigt.

Casts

Mit einem Cast lässt sich ein Typ in einen anderen umwandeln (uminterpretieren).

```
int i = 0;
void * p = & i;
int j = * (int *) p;
```

`p`	hat Typ `void *`
`(int *) p`	hat Typ `int *`
`* (int *) p`	hat Typ `int`

Quiz: Richtiges Teilen

Frage: Welchen Wert erhält die Variable x?

```
int i = 10;
float x = 1 / i;
```

Zusatzfrage: Wie kann man es erreichen, dass x den korrekten Wert von 0.1 erhält?

Antwort: x = 0.0

Grund: 1/i wird als ganzzahlige Division ausgeführt (Ergebnis 0) und das Ergebnis danach in ein **float** konvertiert.

Abhilfe könnte man durch einen Cast schaffen:

```
int i = 10;
float x = 1 / (float) i;
```

Pointer auf Pointer

Pointer können auch auf Pointer zeigen:

```
int foo = 5;
int * bar = & foo;
int * * bar2 = & bar;
printf("%i", * * bar2);
```

⇐ Ausgabe: 5

`bar2` ist ein Pointer auf einen Pointer auf ein int.

`*bar2` ist ein Pointer auf ein int.

`**bar2` ist ein int.

Das Spiel kann man beliebig weiter treiben. Wer Lust hat, kann sich durchaus einen int ******* bar8 definieren – ob das Sinn macht, muss man selbst entscheiden.

Pointer und Arrays

Pointer zeigt auf Elemente eines Arrays:

```
int arr [3] = {2,3,5};
int * bar = arr;

int foo = * bar;
```

⇐ bar zeigt auf das das erste Element von arr

⇐ setzt foo auf 2

Ein Array verhält sich so, als wäre es ein Pointer auf sein erstes Element.

Abgesehen von der Definition/Deklaration unterscheiden sich Array-Variablen in ihrer Verwendung kaum von Pointer-Variablen.

Aber man beachte: **Arrays SIND keine Pointer**.

Die Definition von Arrays und Pointern unterscheidet sich beträchtlich voneinander: Bei Arrays der Länge n wird ein Speicherbereich für das Feld von n Variablen reserviert, Pointer brauchen lediglich Platz für die Speicherung einer Speicheradresse (bei PCs gewöhnlich 4 bytes).

Der *subscript*-Operator [ ]

Mit [ ] kann man auf die einzelnen Elemente eines Arrays zugreifen:

```
int arr [3] = {2,3,5};
int foo = arr [2];
```

⇐ setzt `foo` auf 5

Der Index des ersten Elements eines Arrays ist 0.

Der subscript-Operator [ ] ist ein suffix-Operator. Man verwechsele ihn nicht mit den [ ], die bei der Definition/Deklaration von Arrays verwendet werden.

Man beachte, dass C/C++ von sich aus keinerlei Checks übernimmt, ob sich der Index auch innerhalb des „erlaubten“ Bereiches befindet.

Bsp:

```
int arr[3] = {2, 3, 5};
int foo = arr [34];          //boeser Fehler
```

Der subscript-Operator kommt sogar klaglos mit negativen Indices klar:

```
int foo = arr [-3];          //nochmal boeser Fehler
```

Durch zu große oder kleine Indices wird unter Umständen auf einen nicht freigegebener Speicherbereich zugegriffen, was zum Abbruch des Programmes führt. Oder es wird eigener Speicher unkontrolliert überschrieben, was zum Absturz des Programmes führen kann.

0 + 2 = 8

Indirection-Operator * und Subcript-Operator [ ] sind miteinander verwandt:

```
foo = arr[2];
```

⇔

```
foo = *(arr + 2);
```

Beides funktioniert auch mit Pointern.
Aber Achtung:

```
int * bar = 0;
```

⇐ bar enthält 0

```
bar = bar + 2;
```

⇐ bar wird um 2 int weiter gesetzt, und enthält nun also eine 8!

Die Änderung der Adresse in einem Pointer nach Addieren oder Subtrahieren einer Zahl hängt davon ab, wie viel Speicher der Typ benötigt, auf den der Pointer zeigt. Ints brauchen 4 bytes, also wird bei jeder Erhöhung des Pointers um 1 dessen Inhalt um 4 vergrößert. Bei einem Pointer auf short int wären es 2, bei einem Pointer auf doube wären es 8, usw.

C-Strings

In C sind Strings Arrays von chars:

```
char s1 [] = "hallo";
s1[1] = 'e';
```

⇐ **s1 == "hello"**

In der C++-Standard-Library gibt es übrigens auch eine String-Klasse, aber es ist an dieser Stelle wichtig, dass Sie auch das Konzept der „C-style Strings“ verstehen.

Mehrdimensionale Arrays

Mehrdimensionale Arrays werden einfach als „Arrays von Arrays" realisiert.

Bsp.:

```
int arr2 [3][6];
```

ist ein Array von 3 Arrays von 6 ints.

Der Zugriff erfolgt z.B. mit mehreren [ ]:

```
int arr2 [3][6];
arr2[1][4] = 17;
```

Auf diese Weise können Arrays fast beliebiger Dimension erzeugt werden. Die Felder liegen alle hintereinander in einem zusammenhängenden Speicherblock.

Auch eine Initialisierung ist möglich. Beispiel für 2 Dimensionen:

```
int arr2 [2][3] = { {1, 2, 3}, {4, 5, 6} };
```

Die Variablen liegen nun tatsächlich in der gleichen Reihenfolge wie hier aufgeschrieben im Speicher.

Quiz: ein kurzes Array

Frage: Was ist der Unterschied zwischen folgenden Variablen?

```
int Var1 = 5;
int Var2 [] = {5};
```

In beiden Fällen wird genau eine Variable vom Typ **int** erzeugt. Wenn ich aber auf den Wert von Var1 zugreifen will kann ich

Var1 = 12;

schreiben, bei Var2 brauche ich einen indirection-Operator (***Var2 = 12;**) oder einen subscript-Operator (**Var2[0] = 12;**).

Quiz: Lauter Zeichen!

Was bedeuten die einzelnen markierten Zeichen in folgendem Code?

```
int X [] = { 1, 2, 3 };
int Y = X[1] & X[2];
int & Z = X[0];
int * A = & Z;
Z = Z * * X;
```

1. [ ] in der Variablendefinition: Definition eines Arrays.
2. [ ] in einem Ausdruck: subscript-Operator.
3. & als binärer Operator: bitweises UND.
4. & in einer Variablendefinition: Definition einer Referenz.
5. * in einer Variablendefinition: Definition eines Pointers.
6. & als unärer Operator: address-of-Operator-
7. * als binärer Operator: Multiplikation
8. * als unärer Operator: indirection-Operator

Frage zum Schluss

Warum sind Referenzen
oft besser als Pointer?

Mögliche Antworten z.B.:

-Weil man sie nicht versehentlich uninitialisiert lassen kann.

-Weil man Argumente, die per Referenz übergeben werden, ohne den lästigen * benutzen kann, und das macht den Code ein ganzes Stück besser lesbar.

Aber es gibt natürlich auch Fälle, in denen man auf Pointer angewiesen ist, z.B. wenn man es mit Arrays zu tun hat.

Regel: **Wenn sowohl Referenzen als auch Pointer möglich sind, bevorzuge Refenzen!**

Aufgaben zum Dienstag:

4. Aufgabe:

Erzeugen Sie ein Projekt mit mindestens zwei .cpp- und einer .h-Datei. Definieren Sie eine Funktion in einer der .cpp Dateien und rufen Sie diese von der anderen Datei aus auf. Machen Sie das gleiche mit einer Variablen. Was würde passiert, wenn Sie in beiden .cpp-Dateien eine eigene main-Funktion definieren würden?

5. Aufgabe:

Schreiben Sie ein Programm, das mit **scanf** eine Reihe von 20 Strings einliest und diese anschließend lexikalisch sortiert wieder ausgibt.

6. Aufgabe:

Eine int-Variable benötigt 4 Bytes Speicher. Finden Sie durch ein geeignetes Programm heraus, ob Sie gerade auf einer *big-endian*-Maschine (das erste Byte ist das höchstwertige) oder auf einer *little-endian*-Maschine (das erste Byte ist das niederwertigste) arbeiten.

PRÜFUNGSAUFGABE:

Schreiben Sie eine Funktion, die einen String (d.h. ein 0-terminiertes char-Array) umdreht, so dass z.B. "Hamster" zu "retsmaH" wird. Dabei dürfen Sie natürlich keine Funktion aus den Standardbibliotheken verwenden, die genau das bereits tut. Sie können die Funktion entweder so schreiben, dass sie den String „in place“ umdreht, oder aber so, dass der umgedrehte String in einen anderen String geschrieben wird.