C-Kurs
Guter Code

Daniel Sturm

24. September 2009

C-Kurs

	Mo	Konzepte, Syntax, ... printf, scanf
	Die	Pointers, Arrays, ... Structs, malloc, ...
	Mi	Compiler, Headers, ... Debugging I
Next →	Do	Guter Code Debugging II
	Fr	stdlib, Bücher, ...

Übersicht

- Wozu guten Code?
- Was ist guter Code?
- Coding Standards
- Refaktorisieren und Testen
- Tipps

Wozu guten Code?

Was ist überhaupt gut?

Wann ist etwas gut?

- Die Frage, ob etwas gut ist, richtet sich nach dem Sinn und Zweck, nach dem Ziel.

Wann ist etwas gut?

- Die Frage, ob etwas gut ist, richtet sich nach dem Sinn und Zweck, nach dem Ziel.
- Was für ein Art von Auto ist gut?

Wann ist etwas gut?

- Die Frage, ob etwas gut ist, richtet sich nach dem Sinn und Zweck, nach dem Ziel.
- Was für ein Art von Auto ist gut?
- Ein Bus, ein Lastwagen, ein Rennwagen?

Wozu wollen wir das Auto benutzen?

Wozu wollen wir den Code benutzen?

Wozu schreiben wir Code?

- Um Probleme zu lösen?

Wozu schreiben wir Code?

- Um Probleme zu lösen?
- Um kreativ zu sein?

Wozu schreiben wir Code?

- Um Probleme zu lösen?
- Um kreativ zu sein?
- Um zu kommunizieren?

Wozu schreiben wir Code?

- Um Probleme zu lösen?
- Um kreativ zu sein?
- Um zu kommunizieren!

Kommunikation

- Code dient der Kommunikation

Kommunikation

- Code dient der Kommunikation
  - mit dem Computer

Kommunikation

- Code dient der Kommunikation
  - mit dem Computer
  - mit Anderen

Kommunikation

- Code dient der Kommunikation
  - mit dem Computer
  - mit Anderen
  - mit einem selbst

Gute Kommunikation

- Gute Kommunikation...

Gute Kommunikation

- Gute Kommunikation...
  - hilft beim Vermeiden von Missverständnissen

Gute Kommunikation

- Guter Code...
  - hilft beim Vermeiden von Fehlern
  - lässt sich leicht anpassen und ändern

Gute Kommunikation

- Guter Code...
  - hilft beim Vermeiden von Fehlern
  - lässt sich leicht anpassen und ändern
  - macht Vieles leichter

</Wozu guten Code?>

Was ist guter Code?

Gute Kommunikation

- Was macht gute Kommunikation aus?

Gute Kommunikation

- Was macht gute Kommunikation aus?
  - Korrekte Syntax

Gute Kommunikation

- Was macht gute Kommunikation aus?
  - Korrekte Syntax
  - Richtige Semantik

Gute Kommunikation

- Was macht gute Kommunikation aus?
  - Korrekte Syntax
  - Richtige Semantik
  - Ansprechend

Gute Kommunikation

- Was macht gute Kommunikation aus?
  - Korrekte Syntax
  - Richtige Semantik
  - Ansprechend
    - Leicht zu verstehen

Gute Kommunikation

- Was macht gute Kommunikation aus?
  - Korrekte Syntax
  - Richtige Semantik
  - Ansprechend
    - Leicht zu verstehen
    - Kurz und einfach

Gute Kommunikation

- Was macht gute Kommunikation aus?
  - Korrekte Syntax
  - Richtige Semantik
  - Ansprechend
    - Leicht zu verstehen
    - Kurz und einfach
    - In sich geschlossen

Gute Kommunikation

- Was macht gute Kommunikation aus?
  - Korrekte Syntax
  - Richtige Semantik
  - Ansprechend
    - Leicht zu verstehen
    - Kurz und einfach
    - In sich geschlossen
    - Keine unnötigen Wiederholungen

Guter Code

- Was macht guten Code aus?
  - Korrekte Syntax
  - Richtige Semantik
  - Ansprechend
    - Leicht zu verstehen
    - Kurz und einfach
    - In sich geschlossen
    - Keine unnötigen Wiederholungen

Guter Code

- Was macht guten Code aus?
  - Korrekte Syntax (Compiler)
  - Richtige Semantik
  - Ansprechend
    - Leicht zu verstehen
    - Kurz und einfach
    - In sich geschlossen
    - Keine unnötigen Wiederholungen

Guter Code

- Was macht guten Code aus?
  - Korrekte Syntax (Compiler)
  - Richtige Semantik (Beweise/Testen)
  - Ansprechend
    - Leicht zu verstehen
    - Kurz und einfach
    - In sich geschlossen
    - Keine unnötigen Wiederholungen

Guter Code

- Was macht guten Code aus?
  - Korrekte Syntax (Compiler)
  - Richtige Semantik (Beweise/Testen)
  - Ansprechend (Intuition/Geschmack)
    - Leicht zu verstehen
    - Kurz und einfach
    - In sich geschlossen
    - Keine unnötigen Wiederholungen

</Was ist guter Code?>

Coding Standards

Code Hygiene

- Guter Code

Code Hygiene

- Guter Code
  - ist nicht messbar oder entscheidbar

Code Hygiene

- Guter Code
  - ist nicht messbar oder entscheidbar
  - braucht ein gutes Design

Code Hygiene

- Guter Code
  - ist nicht messbar oder entscheidbar
  - braucht ein gutes Design
  - muss gepflegt werden

Konventionen

- Konventionen legen fest wie Code geschrieben wird

Konventionen

- Konventionen legen fest wie Code geschrieben wird
- Es gibt viele...
  - ANSI, GNU, Linux, ...
  - Jede größere Organisation hat einen.

Anordnung

Einrückung

```
if  (    hours<
24  && minutes<
60  && seconds<
60  )
{return     1
;}         else
{return   0
;}
```

Einrückung

```
if (hours < 24 && minutes < 60 && seconds < 60)
{
  return  1;
} else
{
  return 0;
}
```

Vertikale Ausrichtung

```
char search[] = {'g', 'n', 'u'};
char* replacement[] = {"Gnu", "is not", "Unix"};

int value = 4711;
int anothervalue = 42;
float yetanothervalue = 3.16;
```

Vertikale Ausrichtung

```
char  search[]      = {'g',   'n',      'u'};
char* replacement[] = {"Gnu", "is not", "Unix"};

int             value = 4711;
int      anothervalue =   42;
float yetanothervalue =    3.16;
```

Leerräume

Leerzeichen

```
int i;
for(i=0;i<10;++i){
  printf("%d",i*i+i);
}
```

Leerzeichen

```
int i;
for(i=0; i<10; ++i){
  printf("%d", i*i+i);
}
```

Leerzeichen

```
int i;
for (i = 0; i < 10; ++i){
  printf("%d", i * i + i);
}
```

Tabulatoren

```
int     ix;     /* Index to scan array */
long    sum;    /* Accumulator for sum */
```

- Sollte man nicht verwenden

Tabulatoren

```
int             ix;             /* Index to scan array
*/
long    sum;    /* Accumulator for sum */
```

- Sollte man nicht verwenden
- Denn bei Jemand anderem sieht das dann vielleicht so aus

Tabulatoren

```
int     ix;     /* Index to scan array */
long    sum;    /* Accumulator for sum */
```

- Sollte man nicht verwenden
- Denn bei Jemand anderem sieht das dann vielleicht so aus
- Besser Leerzeichen verwenden

Klammern

Klammern

```
int result;
if (hours < 24 && minutes < 60 && seconds < 60)
  result = 1;
else
result 0;
return result;
```

- Klammern könnte man weglassen

Klammern

```
int result;
if (hours < 24 && minutes < 60 && seconds < 60)
  doSomething();
  result = 1;
else
result 0;
return result;
```

- Klammern könnte man weglassen
- Sollte man aber nicht

Namen und Bezeichner

Namen und Bezeichner

```
int checkValid(int a, int b, int c){
  if (a < 24 && b < 60 && c < 60){
    return  1;
  } else {
    return 0;
  }
}
```

Namen und Bezeichner

```
int checkValid(int hours, int minutes, int seconds){
  if (hours < 24 && minutes < 60 && seconds < 60){
    return  1;
  } else {
    return 0;
  }
}
```

- Sollen aussagekräftig sein

Namen und Bezeichner

```
int isValid(int hours, int minutes, int seconds){
  if (hours < 24 && minutes < 60 && seconds < 60){
    return  1;
  } else {
    return 0;
  }
}
```

- Sollen aussagekräftig sein

Namen und Bezeichner

```
typedef int bool;


bool isValid(int hours, int minutes, int seconds){
  if (hours < 24 && minutes < 60 && seconds < 60){
    return  1;
  } else {
    return 0;
  }
}
```

- Sollen aussagekräftig sein
- Ein typ bool macht es eindeutiger

Namen und Bezeichner

```
typedef int bool;
#define FALSE 0
#define TRUE 1
bool isValid(int hours, int minutes, int seconds){
  if (hours < 24 && minutes < 60 && seconds < 60){
    return TRUE;
  } else {
    return FALSE;
  }
}
```

- Sollen aussagekräftig sein
- Ein typ bool macht es eindeutiger
- TRUE und FALSE sind noch eindeutiger

Namen und Bezeichner

```
typedef enum {FALSE=0, TRUE } bool;


bool isValid(int hours, int minutes, int seconds){
  if (hours < 24 && minutes < 60 && seconds < 60){
    return TRUE;
  } else {
    return FALSE;
  }
}
```

- Sollen aussagekräftig sein
- Ein typ bool macht es eindeutiger
- TRUE und FALSE sind noch eindeutiger
- Ein Enum ist auch möglich (Vorsicht: 0 == FALSE?)

Namenskonventionen

- Komplette Wörter nutzen, keine Abkürzungen
  - nicht: `AKueFi` lieber: `KompletteWoerter`

Namenskonventionen

- Komplette Wörter nutzen, keine Abkürzungen
  - nicht: `AKueFi` lieber: `KompletteWoerter`
- Unnötig lange Namen vermeiden
  - `dieser_name_hier_ist_unnoetigerweise_viel_zu_lang_gerate`

Namenskonventionen

- Komplette Wörter nutzen, keine Abkürzungen
  - nicht: `AKueFi` lieber: `KompletteWoerter`
- Unnötig lange Namen vermeiden
  - `dieser_name_hier_ist_unnoetigerweise_viel_zu_lang_gerate`
- Ähnliche Namen vermeiden
  - nicht: `simile` , `smiles` und `similar`

Namenskonventionen

- Konstanten werden groß geschrieben
  - `KONSTANTEN_NAME`

Namenskonventionen

- Konstanten werden groß geschrieben
  - `KONSTANTEN_NAME`
- Funktions- und Variablennamen werden klein geschrieben
  - `funktionsname()` und `variablenname`

Namenskonventionen

- Konstanten werden groß geschrieben
  - `KONSTANTEN_NAME`
- Funktions- und Variablennamen werden klein geschrieben
  - `funktionsname()` und `variablenname`
- Structnamen beginnen groß
  - `Student`

Namenskonventionen

- Konstanten werden groß geschrieben
  - `KONSTANTEN_NAME`
- Funktions- und Variablennamen werden klein geschrieben
  - `funktionsname()` und `variablenname`
- Structnamen beginnen groß
  - `Student`
- Definierte Typen und Enums gelegentlich auch
  - `Typ` und `Enum`

Namenskonventionen

- Unterstrich-Trennung
  - delimiter_seperated_words

Namenskonventionen

- Unterstrich-Trennung
  - delimiter_seperated_words
- CamelCase
  - letterCaseSeperatedWords

Namenskonventionen

- Unterstrich-Trennung
  - delimiter_seperated_words
- CamelCase
  - letterCaseSeperatedWords
- Ungarische Notation

Namenskonventionen

- Unterstrich-Trennung
  - delimiter_seperated_words
- CamelCase
  - letterCaseSeperatedWords
- Ungarische Notation
  - `pNext` $\widehat{=}$ Pointer

Namenskonventionen

- Unterstrich-Trennung
  - delimiter_seperated_words
- CamelCase
  - letterCaseSeperatedWords
- Ungarische Notation
  - `pNext` $\widehat{=}$ Pointer
  - `iHash` $\widehat{=}$ Integer

Namenskonventionen

- Unterstrich-Trennung
  - delimiter_seperated_words
- CamelCase
  - letterCaseSeperatedWords
- Ungarische Notation
  - `pNext` $\widehat{=}$ Pointer
  - `iHash` $\widehat{=}$ Integer
  - `pszData` $\widehat{=}$ Pointer auf einen Null-terminierten String

Namenskonventionen

- Unterstrich-Trennung
  - delimiter_seperated_words
- CamelCase
  - letterCaseSeperatedWords
- Ungarische Notation
  - `pNext` $\widehat{=}$ Pointer
  - `iHash` $\widehat{=}$ Integer
  - `pszData` $\widehat{=}$ Pointer auf einen Null-terminierten String
  - `g_aHashTable` $\widehat{=}$ globales Array

Kommentare

Kommentare

Jemand hat mal gesagt (sinngemäß):

Kommentare sind wie Deo.
Wenn der Code stinkt, nutzt man
Kommentare um ihn verständlich zu machen.

Kommentare

Jemand hat mal gesagt (sinngemäß):

Kommentare sind wie Deo.
Wenn der Code stinkt, nutzt man
Kommentare um ihn verständlich zu machen.

- Es ist besser, den Code zu reinigen.

Kommentare

Jemand hat mal gesagt (sinngemäß):

Kommentare sind wie Deo.
Wenn der Code stinkt, nutzt man
Kommentare um ihn verständlich zu machen.

- Es ist besser, den Code zu reinigen.
- Wenn der Code gut geschrieben ist, braucht man keinen Kommentar um ihn zu verstehen.

Kein Kommentar

Kommentare sollten **nicht** wiederholen was im Code steht

Kein Kommentar

Kommentare sollten **nicht** wiederholen was im Code steht

```
int i;
/* counting from 0 to 9 */
for(i=0; i<10; ++i){
  /* printing a decimal number */
  printf("%d", i*i+i);
}
```

Kein Kommentar

Kommentare sollten **nicht** wiederholen was im Code steht

```
int i;
/* counting from 0 to 9 */
for(i=0; i<10; ++i){
  /* printing a decimal number */
  printf("%d", i*i+i);
}
```

1. überflüssig

Kein Kommentar

Kommentare sollten **nicht** wiederholen was im Code steht

```
int i;
/* counting from 0 to 9 */
for(i=0; i<20; ++i){
  /* printing a decimal number */
  printf("%x", i*i+i);
}
```

1. überflüssig
2. potentiell veraltet

Dokumentation

Kommentare sollen dokumentieren wie Funktionen benutzt werden bzw. was sie machen.

Dokumentation

Kommentare sollen dokumentieren wie Funktionen benutzt werden bzw. was sie machen.

```
/**
 * Diese Funktion versucht den Quelltext zu verstehen.
 * Wenn comment ungleich NULL ist wird der Kommentar
 * zu Hilfe genommen.  Wenn code NULL ist dann wird
 * ein Kommentar erwartet, sonst versteht sie nichts.
 * @code Der Quellcode als NULL terminierter String
 * @comment Der Kommentar als NULL terminierter String
 * @return 1 wenn verstanden, sonst 0;
 */
int understand( char *code, char *comment);
```

</Coding Standards>

Refaktorisieren und Testen

Stinkender Code

- Man sagt 'Code Smells' wenn...

Stinkender Code

- Man sagt 'Code Smells' wenn...
  - ähnlicher Code an mehreren Stellen steht.

Stinkender Code

- Man sagt 'Code Smells' wenn...
  - ähnlicher Code an mehreren Stellen steht.
  - eine Funktion zu groß ist.

Stinkender Code

- Man sagt 'Code Smells' wenn...
  - ähnlicher Code an mehreren Stellen steht.
  - eine Funktion zu groß ist.
  - eine Funktion andere Funktionen übermäßig nutzt.

Stinkender Code

- Man sagt 'Code Smells' wenn...
  - ähnlicher Code an mehreren Stellen steht.
  - eine Funktion zu groß ist.
  - eine Funktion andere Funktionen übermäßig nutzt.
  - eine Funktion von internen Details anderer abhängt.

Stinkender Code

- Man sagt 'Code Smells' wenn...
  - ähnlicher Code an mehreren Stellen steht.
  - eine Funktion zu groß ist.
  - eine Funktion andere Funktionen übermäßig nutzt.
  - eine Funktion von internen Details anderer abhängt.
  - eine Funktion zu wenig macht.

Stinkender Code

- Man sagt 'Code Smells' wenn...
  - ähnlicher Code an mehreren Stellen steht.
  - eine Funktion zu groß ist.
  - eine Funktion andere Funktionen übermäßig nutzt.
  - eine Funktion von internen Details anderer abhängt.
  - eine Funktion zu wenig macht.
  - zwei Funktionen etwas sehr ähnliches tun.

Refaktorisieren

"If it smells, change it"

Testen

- Testfälle
  - Werden immer wieder durchlaufen
  - Stellen sicher, dass der Code das macht, was er soll
  - Besonders sinnvoll beim Refaktorisieren

</Refaktorisieren und Testen>

Tipps

Vergleiche

Bei Vergleichen immer die Konstante auf die linke Seite

```
if (7 == number){
  /* ... */
}
```

Vergleiche

Bei Vergleichen immer die Konstante auf die linke Seite

```
if (7 = number){
  /* ... */
}
```

- Falls ein Gleichheitszeichen vergessen ⇒ Compilerfehler

Vergleiche

Bei Vergleichen immer die Konstante auf die linke Seite

```
if (number = 7){
  /* ... */
}
```

- Falls ein Gleichheitszeichen vergessen ⇒ Compilerfehler
- Anders herum merkt man den Fehler nicht sofort.

Magic Numbers

In der Regel keine Zahlen einfach so in den Quelltext schreiben.

```
screen = SDL_SetVideoMode(640, 480, 16);
```

Magic Numbers

In der Regel keine Zahlen einfach so in den Quelltext schreiben.

```
screen = SDL_SetVideoMode(640, 480, 16);
```

- Überlegen, was diese Zahl bedeutet und...

Magic Numbers

In der Regel keine Zahlen einfach so in den Quelltext schreiben.

```
#define WIDTH 640
#define HEIGHT 480

screen = SDL_SetVideoMode(WIDTH, HEIGHT, 16);
```

- Überlegen, was diese Zahl bedeutet und...
- entweder mit einem Makro definieren

Magic Numbers

In der Regel keine Zahlen einfach so in den Quelltext schreiben.

```
#define WIDTH 640
#define HEIGHT 480
const int BPP 16;
screen = SDL_SetVideoMode(WIDTH, HEIGHT, BPP);
```

- Überlegen, was diese Zahl bedeutet und...
- entweder mit einem Makro definieren
- oder als Konstante definieren

12. Schritte zu besserem Code - Der Joel Test

12. Schritte zu besserem Code - Der Joel Test

1. Habt ihr eine Quellenverwaltung?

12. Schritte zu besserem Code - Der Joel Test

1. Habt ihr eine Quellenverwaltung?
2. Baut ihr das Softwarepaket in einem Schritt?

12. Schritte zu besserem Code - Der Joel Test

1. Habt ihr eine Quellenverwaltung?
2. Baut ihr das Softwarepaket in einem Schritt?
3. Baut ihr das Paket täglich?

12. Schritte zu besserem Code - Der Joel Test

1. Habt ihr eine Quellenverwaltung?
2. Baut ihr das Softwarepaket in einem Schritt?
3. Baut ihr das Paket täglich?
4. Habt ihr eine Fehlerdatenbank?

12. Schritte zu besserem Code - Der Joel Test

1. Habt ihr eine Quellenverwaltung?
2. Baut ihr das Softwarepaket in einem Schritt?
3. Baut ihr das Paket täglich?
4. Habt ihr eine Fehlerdatenbank?
5. Beseitigt ihr Fehler, bevor ihr weiterkodieren?

12. Schritte zu besserem Code - Der Joel Test

1. Habt ihr eine Quellenverwaltung?
2. Baut ihr das Softwarepaket in einem Schritt?
3. Baut ihr das Paket täglich?
4. Habt ihr eine Fehlerdatenbank?
5. Beseitigt ihr Fehler, bevor ihr weiterkodieren?
6. Habt ihr immer einen aktuellen Zeitplan?

12. Schritte zu besserem Code - Der Joel Test

1. Habt ihr eine Quellenverwaltung?
2. Baut ihr das Softwarepaket in einem Schritt?
3. Baut ihr das Paket täglich?
4. Habt ihr eine Fehlerdatenbank?
5. Beseitigt ihr Fehler, bevor ihr weiterkodieren?
6. Habt ihr immer einen aktuellen Zeitplan?
7. Habt ihr ein schriftliches Konzept?

12. Schritte zu besserem Code - Der Joel Test

1. Habt ihr eine Quellenverwaltung?
2. Baut ihr das Softwarepaket in einem Schritt?
3. Baut ihr das Paket täglich?
4. Habt ihr eine Fehlerdatenbank?
5. Beseitigt ihr Fehler, bevor ihr weiterkodieren?
6. Habt ihr immer einen aktuellen Zeitplan?
7. Habt ihr ein schriftliches Konzept?
8. Haben die Programmierer Ruhe zum Arbeiten?

12. Schritte zu besserem Code - Der Joel Test

1. Habt ihr eine Quellenverwaltung?
2. Baut ihr das Softwarepaket in einem Schritt?
3. Baut ihr das Paket täglich?
4. Habt ihr eine Fehlerdatenbank?
5. Beseitigt ihr Fehler, bevor ihr weiterkodieren?
6. Habt ihr immer einen aktuellen Zeitplan?
7. Habt ihr ein schriftliches Konzept?
8. Haben die Programmierer Ruhe zum Arbeiten?
9. Habt ihr die besten Werkzeuge gekauft?

12. Schritte zu besserem Code - Der Joel Test

1. Habt ihr eine Quellenverwaltung?
2. Baut ihr das Softwarepaket in einem Schritt?
3. Baut ihr das Paket täglich?
4. Habt ihr eine Fehlerdatenbank?
5. Beseitigt ihr Fehler, bevor ihr weiterkodieren?
6. Habt ihr immer einen aktuellen Zeitplan?
7. Habt ihr ein schriftliches Konzept?
8. Haben die Programmierer Ruhe zum Arbeiten?
9. Habt ihr die besten Werkzeuge gekauft?
10. Habt ihr Tester?

12. Schritte zu besserem Code - Der Joel Test

1. Habt ihr eine Quellenverwaltung?
2. Baut ihr das Softwarepaket in einem Schritt?
3. Baut ihr das Paket täglich?
4. Habt ihr eine Fehlerdatenbank?
5. Beseitigt ihr Fehler, bevor ihr weiterkodieren?
6. Habt ihr immer einen aktuellen Zeitplan?
7. Habt ihr ein schriftliches Konzept?
8. Haben die Programmierer Ruhe zum Arbeiten?
9. Habt ihr die besten Werkzeuge gekauft?
10. Habt ihr Tester?
11. Müssen Bewerber ein Stück Code schreiben?

12. Schritte zu besserem Code - Der Joel Test

1. Habt ihr eine Quellenverwaltung?
2. Baut ihr das Softwarepaket in einem Schritt?
3. Baut ihr das Paket täglich?
4. Habt ihr eine Fehlerdatenbank?
5. Beseitigt ihr Fehler, bevor ihr weiterkodieren?
6. Habt ihr immer einen aktuellen Zeitplan?
7. Habt ihr ein schriftliches Konzept?
8. Haben die Programmierer Ruhe zum Arbeiten?
9. Habt ihr die besten Werkzeuge gekauft?
10. Habt ihr Tester?
11. Müssen Bewerber ein Stück Code schreiben?
12. Macht ihr "Flurtests"?

</Tipps>

Danke!

Danke!

Fragen?