

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

8. Oktober 2018

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

rerum naturalium = der natürlichen Dinge (lat.)

8. Oktober 2018

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

rerum naturalium = der natürlichen Dinge (lat.)

8. Oktober 2018

Zu dieser Lehrveranstaltung



- **Lehrmaterialien:**
<https://gitlab.cvh-server.de/pgerwinski/cpp.git>
- **Statt Klausur: Projektaufgabe**
 - Hausarbeit und Kolloquium
 - ein neues, nichttriviales Programm in einer C++-ähnlichen Sprache selbst entwickeln
 - ein vorhandenes, nichttriviales Programm in einer C++-ähnlichen Sprache mit- und/oder weiterentwickeln
 - Sonstiges, was zum Thema der Lehrveranstaltung paßt und sich auf Master-Niveau bewegt
- **Plan:**
die Theorie möglichst zügig abarbeiten,
möglichst frühzeitig mit dem praktischen Arbeiten beginnen
- **Wir sind flexibel. ;-)**



Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

1.1 Was ist C?

1.2 Was ist C++?

2 Wiederholung: Programmieren in C

3 Einführung in C++

4 Standard-Bibliotheken (STL)

5 C++11

6 Plug-In-Architekturen

7 Die Boost-Bibliothek



Änderungen
vorbehalten

1 Einführung

1.1 Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen



Hardware und/oder Betriebssystem

- Hardware direkt ansprechen und effizient einsetzen
- ... bis hin zu komplexen Software-Projekten

→ Man kann Computer vollständig beherrschen.

1 Einführung

1.1 Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen
- Hardware direkt ansprechen und effizient einsetzen
- ... bis hin zu komplexen Software-Projekten
- leistungsfähig, aber gefährlich

„High-Level-Assembler“

- kein „Fallschirm“
- kompakte Schreibweise

C makes it easy to shoot yourself in the foot.

Bjarne Stroustrup, ca. 1986

http://www.stroustrup.com/bs_faq.html#really-say-that

Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln
- kein „Fallschirm“

1 Einführung

1.2 Was ist C++?

Etabliertes Profi-Werkzeug

- kompatibel zu C

C++ unterstützt

- *objektorientierte Programmierung*
- *Datenabstraktion*
- *generische Programmierung*

C++ is a better C.

Bjarne Stroustrup, Autor von C++
<http://www.stroustrup.com/C++.html>

*C makes it easy to shoot yourself in the foot;
C++ makes it harder, but when you do
it blows your whole leg off.*

Bjarne Stroustrup, Autor von C++, ca. 1986
http://www.stroustrup.com/bs_faq.html#really-say-that

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

1.1 Was ist C?

1.2 Was ist C++?

2 Wiederholung: Programmieren in C

2.1 Hello, world!

2.2 Programme compilieren und ausführen

2.3 Elementare Aus- und Eingabe

2.4 Elementares Rechnen

2.5 Verzweigungen

2.6 Schleifen

2.7 Seiteneffekte

2.8 Strukturierte Programmierung

...

3 Einführung in C++

...



Änderungen
vorbehalten

2 Wiederholung: Programmieren in C

2.1 Hello, world!

Text ausgeben

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

```
int main (void)
```

```
{  
    printf ("Hello, _world!\n");  
    return 0;  
}
```

2.2 Programme compilieren und ausführen

```
$ gcc hello-1.c -o hello-1
```

```
$ ./hello-1
```

```
Hello, world!
```


```
$
```

2.2 Programme compilieren und ausführen

```
$ gcc -Wall -O hello-1.c -o hello-1  
$ ./hello-1  
Hello, world!  
$
```

2.2 Programme compilieren und ausführen

```
$ gcc -Wall -O hello-1.c -o hello-1
$ ./hello-1
Hello, world!
$
```



-Wall	alle Warnungen einschalten
-O	optimieren
-O3	maximal optimieren
-Os	Codegröße optimieren
...	gcc hat <i>sehr viele</i> Optionen.

2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

```
    printf ("Die_Antwort_lautet:_");
```

```
    printf (42);
```

```
    printf ("\n");
```

```
    return 0;
```

```
}
```

2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

```
    printf ("Die_Antwort_lautet:_");
```

```
    printf (42);
```

```
    printf ("\n");
```

```
    return 0;
```

```
}
```

→ Absturz

2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

```
    printf ("Die_Antwort_lautet:_%d\n", 42);
```

```
    return 0;
```

```
}
```

Formatspezifikation „d“: „dezimal“



2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

```
    printf ("Die_Antwort_lautet:_%d\n", 42);
```

```
    return 0;
```

```
}
```



Formatspezifikation „d“: „dezimal“

Weitere Formatspezifikationen:
siehe Online-Dokumentation
(z. B. man 3 printf),
Internet-Recherche oder Literatur

2.3 Elementare Aus- und Eingabe

Wert einlesen

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

```
int main (void)
```

```
{
```

```
    double a;
```

```
    printf ("Bitte_eine_Zahl_eingeben:_");
```

```
    scanf ("%lf", &a);
```

```
    printf ("Ihre_Antwort_war:_%lf\n", a);
```

```
    return 0;
```

```
}
```

Formatspezifikation „lf“:
„long floating-point“

Das „&“ nicht vergessen!

2.4 Elementares Rechnen

Wert an Variable zuweisen

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

```
int main (void)
```

```
{
```

```
    int a;
```

```
    printf ("Bitte_eine_Zahl_eingeben:_");
```

```
    scanf ("%d", &a);
```

```
    a = 2 * a;
```

```
    printf ("Das_Doppelte_ist:_%d\n", a);
```

```
    return 0;
```

```
}
```

2.5 Verzweigungen

if-Verzweigung

```
if (b != 0)  
    printf ("%d\n", a / b);
```

2.5 Verzweigungen

if-Verzweigung

```
if (b != 0)
    printf ("%d\n", a / b);
```

Wahrheitswerte in C: numerisch

0 steht für *falsch* (*false*),
≠ 0 steht für *wahr* (*true*).

```
if (b)
    printf ("%d\n", a / b);
```

2.6 Schleifen

while-Schleife

```
a = 1;  
while (a <= 10)  
{  
    printf ("%d\n", a);  
    a = a + 1;  
}
```

2.6 Schleifen

while-Schleife

```
a = 1;
while (a <= 10)
{
    printf ("%d\n", a);
    a = a + 1;
}
```

for-Schleife

```
for (a = 1; a <= 10; a = a + 1)
    printf ("%d\n", a);
```

2.6 Schleifen

while-Schleife

```
a = 1;
while (a <= 10)
{
    printf ("%d\n", a);
    a = a + 1;
}
```

for-Schleife

```
for (a = 1; a <= 10; a = a + 1)
    printf ("%d\n", a);
```

do-while-Schleife

```
a = 1;
do
{
    printf ("%d\n", a);
    a = a + 1;
}
while (a <= 10);
```


2.7 Seiteneffekte

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

```
int main (void)
{
    printf ("%d\n", 42);
    "\n";
    return 0;
}
```

2.7 Seiteneffekte

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

```
int main (void)
```

```
{
```

```
    printf ("%d\n", 42);
```

```
    "\n";
```

← Ausdruck als Anweisung: Wert wird ignoriert

```
    return 0;
```

```
}
```

2.7 Seiteneffekte

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

```
int main (void)
```

```
{
```

```
    printf ("%d\n", 42);
```

```
    "\n";
```

```
    return 0;
```

```
}
```

← Ausdruck als Anweisung: Wert wird ignoriert

2.7 Seiteneffekte

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

```
int main (void)
```

```
{  
    int a = printf ("%d\n", 42);  
    printf ("%d\n", a);  
    return 0;  
}
```

2.7 Seiteneffekte

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

```
int main (void)
```

```
{  
    int a = printf ("%d\n", 42);  
    printf ("%d\n", a);  
    return 0;  
}
```

```
$ gcc -Wall -O side-effects-1.c -o side-effects-1
```

```
$ ./side-effects-1
```

```
42
```

```
3
```

```
$
```

2.7 Seiteneffekte

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

```
int main (void)
```

```
{
```

```
    int a = printf ("%d\n", 42);
```

```
    printf ("%d\n", a);
```

```
    return 0;
```

```
}
```

- `printf()` ist eine Funktion.

2.7 Seiteneffekte

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

```
int main (void)
```

```
{  
    int a = printf ("%d\n", 42);  
    printf ("%d\n", a);  
    return 0;  
}
```

- `printf()` ist eine Funktion.
- „Haupteffekt“: Wert zurückliefern
(hier: Anzahl der ausgegebenen Zeichen)

2.7 Seiteneffekte

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

```
int main (void)
```

```
{
```

```
    int a = printf ("%d\n", 42);
```

```
    printf ("%d\n", a);
```

```
    return 0;
```

```
}
```

- `printf()` ist eine Funktion.
- „Haupteffekt“: Wert zurückliefern
(hier: Anzahl der ausgegebenen Zeichen)
- *Seiteneffekt*: Ausgabe

2.7 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `¬foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo--`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `--foo`

Binäre Operatoren:

- Rechnen: `+` `-` `*` `/` `%`
- Vergleich: `==` `!=` `<` `>` `<=` `>=`
- Zuweisung: `=` `+=` `-=` `*=` `/=` `%=`
- Ignorieren: `,`

2.7 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `—foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo—`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `—foo`

Binäre Operatoren:

- Rechnen: `+ — * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `,`

rot = mit Seiteneffekt

2.7 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `—foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo—`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `—foo`

Binäre Operatoren:

- Rechnen: `+ — * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `,`

rot = mit Seiteneffekt

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

```
{
```

```
    printf ("%d\n", i);
```

```
    i++;
```

```
}
```

2.7 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `—foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo—`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `—foo`

Binäre Operatoren:

- Rechnen: `+ — * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `,`

rot = mit Seiteneffekt

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

```
{  
    printf ("%d\n", i);  
    i++;  
}
```

```
for (i = 0; i < 10; i++)
```

```
    printf ("%d\n", i);
```

2.7 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `—foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo—`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `—foo`

Binäre Operatoren:

- Rechnen: `+ — * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `,`

rot = mit Seiteneffekt

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

```
{  
    printf ("%d\n", i);  
    i++;  
}
```

```
for (i = 0; i < 10; i++)
```

```
    printf ("%d\n", i);
```

```
i = 0;
```

```
while (i < 10)
```

```
    printf ("%d\n", i++);
```

2.7 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `—foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo—`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `—foo`

Binäre Operatoren:

- Rechnen: `+ — * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `,`

rot = mit Seiteneffekt

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

```
{  
    printf ("%d\n", i);  
    i++;  
}
```

```
for (i = 0; i < 10; i++)
```

```
    printf ("%d\n", i);
```

```
i = 0;
```

```
while (i < 10)
```

```
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

```
int i;
```

```
i = 0;
```

```
while (i < 10)
```

```
{
```

```
    printf ("%d\n", i);
```

```
    i++;
```

```
}
```

```
for (i = 0; i < 10; i++)
```

```
    printf ("%d\n", i);
```

```
i = 0;
```

```
while (i < 10)
```

```
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

```
i = 0;  
while (1)  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
int i;
```

```
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```



```
i = 0;  
while (1)  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
i = 0;  
loop:  
if (i >= 10)  
    goto endloop;  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

```
int i;
```

```
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

2.8 Strukturierte Programmierung

```
i = 0;  
while (1)  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
i = 0;  
loop:  
if (i >= 10)  
    goto endloop;  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

```
int i;  
  
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

2.8 Strukturierte Programmierung

```
i = 0;  
while (1)      fragwürdig  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
i = 0;  
loop:  
if (i >= 10)   sehr fragwürdig  
    goto endloop;  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

(siehe z. B.:
<http://xkcd.com/292/>)

```
int i;  
  
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

gut

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

nur, wenn
Sie wissen,
was Sie tun

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

1.1 Was ist C?

1.2 Was ist C++?

2 Wiederholung: Programmieren in C

2.1 Hello, world!

2.2 Programme compilieren und ausführen

2.3 Elementare Aus- und Eingabe

2.4 Elementares Rechnen

2.5 Verzweigungen

2.6 Schleifen

2.7 Seiteneffekte

2.8 Strukturierte Programmierung

...

3 Einführung in C++

...



Änderungen
vorbehalten

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

15. Oktober 2018

Zu dieser Lehrveranstaltung



- **Lehrmaterialien:**
<https://gitlab.cvh-server.de/pgerwinski/cpp.git>
- **Statt Klausur: Projektaufgabe**
 - Hausarbeit und Kolloquium
 - ein neues, nichttriviales Programm in einer C++-ähnlichen Sprache selbst entwickeln
 - ein vorhandenes, nichttriviales Programm in einer C++-ähnlichen Sprache mit- und/oder weiterentwickeln
 - Sonstiges, was zum Thema der Lehrveranstaltung paßt und sich auf Master-Niveau bewegt
- **Plan:**
die Theorie möglichst zügig abarbeiten,
möglichst frühzeitig mit dem praktischen Arbeiten beginnen
- **Wir sind flexibel. ;-)**



Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

1.1 Was ist C?

1.2 Was ist C++?

2 Wiederholung: Programmieren in C

3 Einführung in C++

4 Standard-Bibliotheken (STL)

5 C++11

6 Plug-In-Architekturen

7 Die Boost-Bibliothek



Änderungen
vorbehalten

1 Einführung

1.1 Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen



Hardware und/oder Betriebssystem

- Hardware direkt ansprechen und effizient einsetzen
- ... bis hin zu komplexen Software-Projekten

→ Man kann Computer vollständig beherrschen.

1 Einführung

1.1 Was ist C?

Etabliertes Profi-Werkzeug

- kleinster gemeinsamer Nenner für viele Plattformen
- Hardware direkt ansprechen und effizient einsetzen
- ... bis hin zu komplexen Software-Projekten
- leistungsfähig, aber gefährlich

„High-Level-Assembler“

- kein „Fallschirm“
- kompakte Schreibweise

C makes it easy to shoot yourself in the foot.

Bjarne Stroustrup, ca. 1986

http://www.stroustrup.com/bs_faq.html#really-say-that

Unix-Hintergrund

- Baukastenprinzip
- konsequente Regeln
- kein „Fallschirm“

1 Einführung

1.2 Was ist C++?

Etabliertes Profi-Werkzeug

- kompatibel zu C

C++ unterstützt

- *objektorientierte Programmierung*
- *Datenabstraktion*
- *generische Programmierung*

C++ is a better C.

Bjarne Stroustrup, Autor von C++
<http://www.stroustrup.com/C++.html>

*C makes it easy to shoot yourself in the foot;
C++ makes it harder, but when you do
it blows your whole leg off.*

Bjarne Stroustrup, Autor von C++, ca. 1986
http://www.stroustrup.com/bs_faq.html#really-say-that

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

1.1 Was ist C?

1.2 Was ist C++?

2 Wiederholung: Programmieren in C

2.1 Hello, world!

2.2 Programme compilieren und ausführen

2.3 Elementare Aus- und Eingabe

2.4 Elementares Rechnen

2.5 Verzweigungen

2.6 Schleifen

2.7 Seiteneffekte

2.8 Strukturierte Programmierung

...

3 Einführung in C++

...



Änderungen
vorbehalten

2 Wiederholung: Programmieren in C

2.1 Hello, world!

Text ausgeben

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

```
int main (void)
```

```
{  
    printf ("Hello, _world!\n");  
    return 0;  
}
```

2.2 Programme compilieren und ausführen

```
$ gcc hello-1.c -o hello-1
```


```
$ ./hello-1
```

```
Hello, world!
```

```
$
```

2.2 Programme compilieren und ausführen

```
$ gcc -Wall -O hello-1.c -o hello-1
$ ./hello-1
Hello, world!
$
```



-Wall	alle Warnungen einschalten
-O	optimieren
-O3	maximal optimieren
-Os	Codegröße optimieren
...	gcc hat <i>sehr viele</i> Optionen.

2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

```
    printf ("Die_Antwort_lautet:_");
```

```
    printf (42);
```

```
    printf ("\n");
```

```
    return 0;
```

```
}
```

→ Absturz

2.3 Elementare Aus- und Eingabe

Wert ausgeben

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

```
int main (void)
```

```
{
```

```
    printf ("Die_Antwort_lautet:_%d\n", 42);
```

```
    return 0;
```

```
}
```



Formatspezifikation „d“: „dezimal“

Weitere Formatspezifikationen:
siehe Online-Dokumentation
(z. B. man 3 printf),
Internet-Recherche oder Literatur

2.3 Elementare Aus- und Eingabe

Wert einlesen

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

```
int main (void)
```

```
{
```

```
    double a;
```

```
    printf ("Bitte_eine_Zahl_eingeben:_");
```

```
    scanf ("%lf", &a);
```

```
    printf ("Ihre_Antwort_war:_%lf\n", a);
```

```
    return 0;
```

```
}
```

Formatspezifikation „lf“:
„long floating-point“

Das „&“ nicht vergessen!

2.4 Elementares Rechnen

Wert an Variable zuweisen

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

```
int main (void)
```

```
{
```

```
    int a;
```

```
    printf ("Bitte_eine_Zahl_eingeben:_");
```

```
    scanf ("%d", &a);
```

```
    a = 2 * a;
```

```
    printf ("Das_Doppelte_ist:_%d\n", a);
```

```
    return 0;
```

```
}
```

2.5 Verzweigungen

if-Verzweigung

```
if (b != 0)
    printf ("%d\n", a / b);
```

Wahrheitswerte in C: numerisch

0 steht für *falsch* (*false*),
≠ 0 steht für *wahr* (*true*).

```
if (b)
    printf ("%d\n", a / b);
```

2.6 Schleifen

while-Schleife

```
a = 1;
while (a <= 10)
{
    printf ("%d\n", a);
    a = a + 1;
}
```

for-Schleife

```
for (a = 1; a <= 10; a = a + 1)
    printf ("%d\n", a);
```

do-while-Schleife

```
a = 1;
do
{
    printf ("%d\n", a);
    a = a + 1;
}
while (a <= 10);
```

2.7 Seiteneffekte

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

```
int main (void)
```

```
{
```

```
    printf ("%d\n", 42);
```

```
    "\n";
```

← Ausdruck als Anweisung: Wert wird ignoriert

```
    return 0;
```

```
}
```

2.7 Seiteneffekte

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

```
int main (void)
```

```
{
```

```
    printf ("%d\n", 42);
```

```
    "\n";
```

```
    return 0;
```

```
}
```

← Ausdruck als Anweisung: Wert wird ignoriert

2.7 Seiteneffekte

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

```
int main (void)
```

```
{  
    int a = printf ("%d\n", 42);  
    printf ("%d\n", a);  
    return 0;  
}
```

```
$ gcc -Wall -O side-effects-1.c -o side-effects-1
```

```
$ ./side-effects-1
```

```
42
```

```
3
```

```
$
```

2.7 Seiteneffekte

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

```
int main (void)
```

```
{  
    int a = printf ("%d\n", 42);  
    printf ("%d\n", a);  
    return 0;  
}
```

- `printf()` ist eine Funktion.
- „Haupteffekt“: Wert zurückliefern
(hier: Anzahl der ausgegebenen Zeichen)
- *Seiteneffekt*: Ausgabe

2.7 Seiteneffekte bei Operatoren

Unäre Operatoren:

- Negation: `—foo`
- Funktionsaufruf: `foo ()`
- Post-Inkrement: `foo++`
- Post-Dekrement: `foo—`
- Prä-Inkrement: `++foo`
- Prä-Dekrement: `—foo`

Binäre Operatoren:

- Rechnen: `+ — * / %`
- Vergleich: `== != < > <= >=`
- Zuweisung: `= += -= *= /= %=`
- Ignorieren: `,`

rot = mit Seiteneffekt

```
int i;
```

```
i = 0;
```

```
while (i < 10)
{
    printf ("%d\n", i);
    i++;
}
```

```
for (i = 0; i < 10; i++)
    printf ("%d\n", i);
```

```
i = 0;
```

```
while (i < 10)
    printf ("%d\n", i++);
```

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

2.8 Strukturierte Programmierung

```
i = 0;  
while (1)      fragwürdig  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
i = 0;  
loop:  
if (i >= 10)   sehr fragwürdig  
    goto endloop;  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

(siehe z. B.:
<http://xkcd.com/292/>)

```
int i;  
  
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

gut

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

nur, wenn
Sie wissen,
was Sie tun

```
for (i = 0; i < 10; printf ("%d\n", i++));
```

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

1.1 Was ist C?

1.2 Was ist C++?

2 Wiederholung: Programmieren in C

2.1 Hello, world!

2.2 Programme compilieren und ausführen

2.3 Elementare Aus- und Eingabe

2.4 Elementares Rechnen

2.5 Verzweigungen

2.6 Schleifen

2.7 Seiteneffekte

2.8 Strukturierte Programmierung

...

3 Einführung in C++

...



Änderungen
vorbehalten

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

1.1 Was ist C?

1.2 Was ist C++?

2 Wiederholung: Programmieren in C

...

2.7 Seiteneffekte

2.8 Strukturierte Programmierung

2.9 Funktionen

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

2.13 Dateien und Fehlerbehandlung

2.14 Parameter des Hauptprogramms

2.15 String-Operationen

...

3 Einführung in C++

...



Änderungen
vorbehalten

2.9 Funktionen

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

```
int answer (void)
```

```
{  
    return 42;  
}
```

```
void foo (void)
```

```
{  
    printf ("%d\n", answer ());  
}
```

```
int main (void)
```

```
{  
    foo ();  
    return 0;  
}
```

2.9 Funktionen

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

```
int answer (void)
```

```
{  
    return 42;  
}
```

```
void foo (void)
```

```
{  
    printf ("%d\n", answer ());  
}
```

```
int main (void)
```

```
{  
    foo ();  
    return 0;  
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}

2.9 Funktionen

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

```
void add_verbose (int a, int b)
{
    printf ("%d_+_d=_d\n", a, b, a + b);
}
```

```
int main (void)
{
    add_verbose (3, 7);
    return 0;
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}

2.9 Funktionen

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

```
void add_verbose (int a, int b)
{
    printf ("%d_+_d=_d\n", a, b, a + b);
}
```

```
int main (void)
{
    add_verbose (3, 7);
    return 0;
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}
- Der Datentyp **void**
steht für „nichts“
und kann ignoriert werden.

2.9 Funktionen

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

```
void add_verbose (int a, int b)
{
    printf ("%d_+_d=_d\n", a, b, a + b);
}
```

```
int main (void)
{
    add_verbose (3, 7);
    return 0;
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}
- Der Datentyp **void**
steht für „nichts“
und muß ignoriert werden.

2.9 Funktionen

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

```
int a, b = 3;
```

```
void foo (void)
```

```
{  
    b++;  
    static int a = 5;  
    int b = 7;  
    printf ("foo():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    a++;  
}
```

```
int main (void)
```

```
{  
    printf ("main():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    foo ();  
    printf ("main():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    a = b = 12;  
    printf ("main():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    foo ();  
    printf ("main():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    return 0;  
}
```

2.10 Zeiger

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

```
void calc_answer (int *a)
```

```
{
```

```
    *a = 42;
```

```
}
```

```
int main (void)
```

```
{
```

```
    int answer;
```

```
    calc_answer (&answer);
```

```
    printf ("The_answer_is_%d.\n", answer);
```

```
    return 0;
```

```
}
```

2.10 Zeiger

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

```
void calc_answer (int *a)
```

```
{
```

```
    *a = 42;
```

```
}
```

- `*a` ist eine `int`.

```
int main (void)
```

```
{
```

```
    int answer;
```

```
    calc_answer (&answer);
```

```
    printf ("The_answer_is_%d.\n", answer);
```

```
    return 0;
```

```
}
```

2.10 Zeiger

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

```
void calc_answer (int *a)
{
    *a = 42;
}
```

- `*a` ist eine **int**.
- unärer Operator `*`:
Pointer-Derferenzierung

```
int main (void)
{
    int answer;
    calc_answer (&answer);
    printf ("The_answer_is_%d.\n", answer);
    return 0;
}
```

2.10 Zeiger

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

```
void calc_answer (int *a)
{
    *a = 42;
}
```

- `*a` ist eine **int**.
- unärer Operator `*`:
Pointer-Derferenzierung

→ `a` ist ein Zeiger (Pointer) auf eine **int**.

```
int main (void)
{
    int answer;
    calc_answer (&answer);
    printf ("The_answer_is_%d.\n", answer);
    return 0;
}
```

2.10 Zeiger

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

```
void calc_answer (int *a)
{
    *a = 42;
}
```

- `*a` ist eine **int**.
- unärer Operator `*`:
Pointer-Derferenzierung

→ `a` ist ein Zeiger (Pointer) auf eine **int**.

```
int main (void)
{
    int answer;
    calc_answer (&answer);
    printf ("The_answer_is_%d.\n", answer);
    return 0;
}
```

- unärer Operator `&`: Adresse

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

- `prime` ist eine Ansammlung von fünf ganzen Zahlen.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.



- `prime` ist ein Zeiger auf eine `int`.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.


2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`. 
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.


2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`. 
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise: `p[i]` statt `*(p + i)`

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", prime[i]);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello_world[] = "Hello,_world!\n";
```

```
    int i = 0;
```

```
    while (hello_world[i] != 0)
```

```
        printf ("%d", hello_world[i++]);
```

```
    return 0;
```

```
}
```

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello_world[] = "Hello,_world!\n";
```

```
    int i = 0;
```

```
    while (hello_world[i])
```

```
        printf ("%d", hello_world[i++]);
```

```
    return 0;
```

```
}
```

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello_world[] = "Hello,_world!\n";
```

```
    char *p = hello_world;
```

```
    while (*p)
```

```
        printf ("%d", *p++);
```

```
    return 0;
```

```
}
```

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello_world[] = "Hello,_world!\n";
```

```
    char *p = hello_world;
```

```
    while (*p)
```

```
        printf ("%c", *p++);
```

```
    return 0;
```

```
}
```

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello_world[] = "Hello,_world!\n";
```

```
    char *p = hello_world;
```

```
    while (*p)
```

```
        printf ("%c", *p++);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello_world[] = "Hello,_world!\n";
```

```
    char *p = hello_world;
```

```
    while (*p)
```

```
        printf ("%c", *p++);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**.

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello_world[] = "Hello,_world!\n";
```

```
    char *p = hello_world;
```

```
    while (*p)
```

```
        printf ("%c", *p++);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars**.

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello_world[] = "Hello,_world!\n";
```

```
    char *p = hello_world;
```

```
    while (*p)
```

```
        printf ("%c", *p++);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello_world[] = "Hello, _world!\n";
```

```
    char *p = hello_world;
```

```
    while (*p)
```

```
        printf ("%c", *p++);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello_world[] = "Hello,_world!\n";
```

```
    char *p = hello_world;
```

```
    while (*p)
```

```
        printf ("%c", *p++);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.
- Die Formatspezifikation entscheidet über die Ausgabe:
 %d dezimal **%c** Zeichen
 %x hexadezimal

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello_world[] = "Hello,_world!\n";
```

```
    char *p = hello_world;
```

```
    while (*p)
```

```
        printf ("%c", *p++);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.
- Die Formatspezifikation entscheidet über die Ausgabe:

%d	dezimal	%c	Zeichen
%x	hexadezimal	%s	String

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

1.1 Was ist C?

1.2 Was ist C++?

2 Wiederholung: Programmieren in C

...

2.7 Seiteneffekte

2.8 Strukturierte Programmierung

2.9 Funktionen

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

2.13 Dateien und Fehlerbehandlung

2.14 Parameter des Hauptprogramms

2.15 String-Operationen

...

3 Einführung in C++

...



Änderungen
vorbehalten

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

29. Oktober 2018

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

2 Wiederholung: Programmieren in C

...

2.7 Seiteneffekte

2.8 Strukturierte Programmierung

2.9 Funktionen

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

2.13 Dateien und Fehlerbehandlung

2.14 Parameter des Hauptprogramms

2.15 String-Operationen

...

3 Einführung in C++

...



Änderungen
vorbehalten

2.8 Strukturierte Programmierung

```
i = 0;  
while (1)      fragwürdig  
{  
    if (i >= 10)  
        break;  
    printf ("%d\n", i++);  
}
```

```
i = 0;  
loop:  
if (i >= 10)   sehr fragwürdig  
    goto endloop;  
printf ("%d\n", i++);  
goto loop;  
endloop:
```

(siehe z. B.:
<http://xkcd.com/292/>)

```
int i;  
  
i = 0;  
while (i < 10)  
{  
    printf ("%d\n", i);  
    i++;  
}
```

gut

```
for (i = 0; i < 10; i++)  
    printf ("%d\n", i);
```

```
i = 0;  
while (i < 10)  
    printf ("%d\n", i++);
```

nur, wenn
Sie wissen,
was Sie tun

```
for (i = 0; i < 10; printf ("%d\n", i++));
```


2.8 Strukturierte Programmierung

Beispiel:

Gesucht ist eine Zahl a mit der Eigenschaft $a \cdot 1117 \bmod 65535 = 137$ sowie ihr kleinster Primfaktor t .

(Aufgaben dieses Typs treten im Kontext von Verschlüsselung auf.)

```
for (int a = 0; a < 1000000; a++)
```

```
{
    if ((a * 1117) % 65535 == 137)
    {
        for (int t = 2; t < 1000000; t++)
        {
            if (a % t == 0)
            {
                printf("a=_%d\nt=_%d\n", a, t);
                a = 2000000;
                break;
            }
            else
                continue;
        }
    }
}
```

(schlecht)

```
int N = 65535;
```

```
int p = 1117;
```

```
int q = 137;
```

```
int a = 0;
```

```
while ((a * p) % N != q)
```

```
    a++;
```

```
int t = 2;
```

```
while (a % t != 0)
```

```
    t++;
```

```
printf("a=_%d\nt=_%d\n", a, t);
```

(deutlich besser)

2.9 Funktionen

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

```
int answer (void)
```

```
{  
    return 42;  
}
```

```
void foo (void)
```

```
{  
    printf ("%d\n", answer ());  
}
```

```
int main (void)
```

```
{  
    foo ();  
    return 0;  
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}
- Das Hauptprogramm ist eine gewöhnliche Funktion mit dem Namen `main()` und dem Rückgabewert `int`:
0: Erfolg
≠ 0: Fehler

2.9 Funktionen

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

```
void add_verbose (int a, int b)
{
    printf ("%d_+_%d=_%d\n", a, b, a + b);
}
```

```
int main (void)
{
    add_verbose (3, 7);
    return 0;
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}
- Der Datentyp **void**
steht für „nichts“
und kann ignoriert werden.

2.9 Funktionen

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

```
void add_verbose (int a, int b)
{
    printf ("%d_+_%d=_%d\n", a, b, a + b);
}
```

```
int main (void)
{
    add_verbose (3, 7);
    return 0;
}
```

- Funktionsdeklaration:
Typ Name (Parameterliste)
{
 Anweisungen
}
- Der Datentyp **void**
steht für „nichts“
und muß ignoriert werden.

2.9 Funktionen

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

```
int a, b = 3;
```

```
void foo (void)
```

```
{  
    b++;  
    static int a = 5;  
    int b = 7;  
    printf ("foo():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    a++;  
}
```

```
int main (void)
```

```
{  
    printf ("main():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    foo ();  
    printf ("main():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    a = b = 12;  
    printf ("main():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    foo ();  
    printf ("main():_"  
           "a=_%d,_b=_%d\n",  
           a, b);  
    return 0;  
}
```

2.10 Zeiger

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

```
void calc_answer (int *a)
```

```
{
```

```
    *a = 42;
```

```
}
```

```
int main (void)
```

```
{
```

```
    int answer;
```

```
    calc_answer (&answer);
```

```
    printf ("The_answer_is_%d.\n", answer);
```

```
    return 0;
```

```
}
```

2.10 Zeiger

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

```
void calc_answer (int *a)
```

```
{
```

```
    *a = 42;
```

```
}
```

- *a ist eine **int**.

```
int main (void)
```

```
{
```

```
    int answer;
```

```
    calc_answer (&answer);
```

```
    printf ("The_answer_is_%d.\n", answer);
```

```
    return 0;
```

```
}
```

2.10 Zeiger

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

```
void calc_answer (int *a)
{
    *a = 42;
}
```

- `*a` ist eine **int**.
- unärer Operator `*`:
Pointer-Derferenzierung

```
int main (void)
{
    int answer;
    calc_answer (&answer);
    printf ("The_answer_is_%d.\n", answer);
    return 0;
}
```


2.10 Zeiger

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

```
void calc_answer (int *a)
{
    *a = 42;
}
```

- `*a` ist eine **int**.
- unärer Operator `*`:
Pointer-Derferenzierung

→ `a` ist ein Zeiger (Pointer) auf eine **int**.

```
int main (void)
{
    int answer;
    calc_answer (&answer);
    printf ("The_answer_is_%d.\n", answer);
    return 0;
}
```

2.10 Zeiger

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

```
void calc_answer (int *a)
{
    *a = 42;
}
```

- `*a` ist eine **int**.
- unärer Operator `*`:
Pointer-Derferenzierung

→ `a` ist ein Zeiger (Pointer) auf eine **int**.

```
int main (void)
{
    int answer;
    calc_answer (&answer);
    printf ("The_answer_is_%d.\n", answer);
    return 0;
}
```

- unärer Operator `&`: Adresse

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

- `prime` ist eine Ansammlung von fünf ganzen Zahlen.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{
```

```
    int prime[5] = { 2, 3, 5, 7, 11 };
```

```
    int *p = prime;
```

```
    for (int i = 0; i < 5; i++)
```

```
        printf ("%d\n", *(p + i));
```

```
    return 0;
```

```
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.



- `prime` ist ein Zeiger auf eine `int`.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.


2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", *(p + i));  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`. 
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.


2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    int *p = prime;  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", p[i]);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`. 
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    for (int i = 0; i < 5; i++)  
        printf ("%d\n", prime[i]);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{  
    int prime[5] = { 2, 3, 5, 7, 11 };  
    for (int *p = prime;  
         p < prime + 5; p++)  
        printf ("%d\n", *p);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`
- Zeiger-Arithmetik:
`p++` rückt den Zeiger `p` um eine `int` weiter.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{  
    int prime[6] = { 2, 3, 5, 7, 11, 0 };  
    for (int *p = prime; *p; p++)  
        printf ("%d\n", *p);  
    return 0;  
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`
- Zeiger-Arithmetik:
`p++` rückt den Zeiger `p` um eine `int` weiter.

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{
```

```
    int prime[] = { 2, 3, 5, 7, 11, 0 };
```

```
    for (int *p = prime; *p; p++)
```

```
        printf ("%d\n", *p);
```

```
    return 0;
```

```
}
```

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`
- Zeiger-Arithmetik:
`p++` rückt den Zeiger `p` um eine `int` weiter.
- Array ohne Längenangabe:
Compiler zählt selbst

2.11 Arrays und Strings

Ein Zeiger zeigt auf eine Variable und deren Nachbarn.

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

```
int main (void)
```

```
{  
    int prime[] = { 2, 3, 5, 7, 11, 0 };  
    for (int *p = prime; *p; p++)  
        printf ("%d\n", *p);  
    return 0;  
}
```

**Die Länge des Arrays
ist *nicht* veränderlich!**

- `prime` ist ein Array von fünf ganzen Zahlen.
- `prime` ist ein Zeiger auf eine `int`.
- `p + i` ist ein Zeiger auf den `i`-ten Nachbarn von `*p`.
- `*(p + i)` ist der `i`-te Nachbar von `*p`.
- Andere Schreibweise:
`p[i]` statt `*(p + i)`
- Zeiger-Arithmetik:
`p++` rückt den Zeiger `p` um eine `int` weiter.
- Array ohne explizite Längenangabe:
Compiler zählt selbst

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, _world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, _world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**.

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars**.

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.

2.11 Arrays und Strings

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

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello, world!\n";
```

```
    for (char *p = hello; *p; p++)
```

```
        printf ("%d", *p);
```

```
    return 0;
```

```
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.

2.11 Arrays und Strings

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

```
int main (void)
{
    char hello[] = "Hello, world!\n";
    for (char *p = hello; *p; p++)
        printf ("%c", *p);
    return 0;
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.
- Die Formatspezifikation entscheidet über die Ausgabe:
 - %d** dezimal
 - %c** Zeichen
 - %x** hexadezimal

2.11 Arrays und Strings

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

```
int main (void)
{
    char hello[] = "Hello, _world!\n";
    printf ("%s", hello);
    return 0;
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.
- Die Formatspezifikation entscheidet über die Ausgabe:

%d	dezimal	%c	Zeichen
%x	hexadezimal	%s	String

2.11 Arrays und Strings

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

```
int main (void)
{
    char *hello = "Hello, _world!\n";
    printf ("%s", hello);
    return 0;
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.
- Die Formatspezifikation entscheidet über die Ausgabe:

%d	dezimal	%c	Zeichen
%x	hexadezimal	%s	String

2.11 Arrays und Strings

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

```
int main (void)
{
    char *hello = "Hello, _world!\n";
    while (*hello)
        printf ("%c", *hello++);
    return 0;
}
```

- Ein **char** ist eine kleinere **int**.
- Ein „String“ in C ist ein Array von **chars**, also ein Zeiger auf **chars** also ein Zeiger auf (kleinere) Integer.
- Der letzte **char** muß 0 sein. Er kennzeichnet das Ende des Strings.
- Die Formatspezifikation entscheidet über die Ausgabe:

%d	dezimal	%c	Zeichen
%x	hexadezimal	%s	String

2.11 Arrays und Zeiger und Strings

h =

H	e	l	l	o	!	\n	Null
---	---	---	---	---	---	----	------

H	e	l	l	o	!	\n	Null
---	---	---	---	---	---	----	------

↑
h =

--

```
char h[] = "Hello!\n";
```

```
printf ("%s", h);
```

```
h[1] = 'a';
```

```
int i = 0;
```

```
while (h[i])  
    printf ("%c", h[i++]);
```

```
while (*h)  
    printf ("%c", *h++);
```

```
char *h = "Hello!\n";
```

```
printf ("%s", h);
```

```
h[1] = 'a';
```

```
int i = 0;
```

```
while (h[i])  
    printf ("%c", h[i++]);
```

```
while (*h)  
    printf ("%c", *h++);
```

2.11 Arrays und Zeiger und Strings

h =

H	e	l	l	o	!	\n	Null
---	---	---	---	---	---	----	------

h =

H	e	l	l	o	!	\n	Null
---	---	---	---	---	---	----	------

↑

--

 ohne Schreibzugriff

```
char h[] = "Hello!\n";
```

```
printf ("%s", h);
```

```
h[1] = 'a';
```

```
int i = 0;
```

```
while (h[i])  
    printf ("%c", h[i++]);
```

```
while (*h)  
    printf ("%c", *h++);
```

```
char *h = "Hello!\n";
```

```
printf ("%s", h);
```

```
h[1] = 'a';
```

```
int i = 0;
```

```
while (h[i])  
    printf ("%c", h[i++]);
```

```
while (*h)  
    printf ("%c", *h++);
```

2.11 Arrays und Zeiger und Strings

`h =`

H	e	l	l	o	!	\n	Null
---	---	---	---	---	---	----	------

Adresse des Arrays

```
char h[] = "Hello!\n";
```

```
printf ("%s", h);
```

```
h[1] = 'a';
```

```
int i = 0;
```

```
while (h[i])  
    printf ("%c", h[i++]);
```

```
while (*h)  
    printf ("%c", *h++);
```

H	e	l	l	o	!	\n	Null
---	---	---	---	---	---	----	------

`h =`

--

ohne Schreibzugriff

```
char *h = "Hello!\n";
```

```
printf ("%s", h);
```

~~`h[1] = 'a';`~~

```
int i = 0;
```

```
while (h[i])  
    printf ("%c", h[i++]);
```

```
while (*h)  
    printf ("%c", *h++);
```

2.11 Arrays und Strings und Zeichen

„Alles ist Zahl.“ – Schule der Pythagoreer, 6. Jh. v. Chr.

"Hello"		{ 72, 101, 108, 108, 111, 0 }
'H'	ist nur eine andere	72
	Schreibweise für	
'a' + 4		'e'

- Welchen Zahlenwert hat '*' im Zeichensatz?

```
printf ("%d\n", '*');
```

(normalerweise: ASCII)

- Ist **char** **ch** ein Großbuchstabe?

```
if (ch >= 'A' && ch <= 'Z')
```

...

- Groß- in Kleinbuchstaben umwandeln

```
ch += 'a' - 'A';
```

2.12 Strukturen

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

```
typedef struct
```

```
{  
    char day, month;  
    int year;  
}  
date;
```

- Eigener Datentyp: `date`
- zusammengesetzt aus elementareren Datentypen

```
int main (void)
```

```
{  
    date today = { 30, 10, 2014 };  
    printf ("%d.%d.%d\n", today.day, today.month, today.year);  
    return 0;  
}
```

2.12 Strukturen

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

```
typedef struct
```

```
{
```

```
    char day, month;
```

```
    int year;
```

```
}
```

```
date;
```

```
void set_date (date *d)
```

```
{
```

```
    (*d).day = 30;
```

```
    (*d).month = 10;
```

```
    (*d).year = 2014;
```

```
}
```

- Eigener Datentyp: `date`
- zusammengesetzt aus elementarerer Datentypen
- `set_date()`: „Methode“ von `date`
- `d->day` ist Abkürzung für `(*d).day`

```
int main (void)
```

```
{
```

```
    date today;
```

```
    set_date (&today);
```

```
    printf ("%d.%d.%d\n", today.day,  
            today.month, today.year);
```

```
    return 0;
```

```
}
```


2.12 Strukturen

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

```
typedef struct
```

```
{
```

```
    char day, month;
```

```
    int year;
```

```
}
```

```
date;
```

```
void set_date (date *d)
```

```
{
```

```
    d->day = 30;
```

```
    d->month = 10;
```

```
    d->year = 2014;
```

```
}
```

- Eigener Datentyp: `date`
- zusammengesetzt aus elementarerer Datentypen
- `set_date()`: „Methode“ von `date`
- `d->day` ist Abkürzung für `(*d).day`

```
int main (void)
```

```
{
```

```
    date today;
```

```
    set_date (&today);
```

```
    printf ("%d.%d.%d\n", today.day,  
            today.month, today.year);
```

```
    return 0;
```

```
}
```

2.13 Dateien und Fehlerbehandlung

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

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    fprintf (f, "Hello, _world!\n");
```

```
    fclose (f);
```

```
    return 0;
```

```
}
```

2.13 Dateien und Fehlerbehandlung

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

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    if (f)
```

```
    {
```

```
        fprintf (f, "Hello,_world!\n");
```

```
        fclose (f);
```

```
    }
```

```
    return 0;
```

```
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.

2.13 Dateien und Fehlerbehandlung

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

```
#include <errno.h>
```

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    if (f)
```

```
    {
```

```
        fprintf (f, "Hello,_world!\n");
```

```
        fclose (f);
```

```
    }
```

```
    else
```

```
        fprintf (stderr, "error_#%d\n", errno);
```

```
    return 0;
```

```
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.
- Die globale Variable `int errno` enthält dann die Nummer des Fehlers. Benötigt: `#include <errno.h>`

2.13 Dateien und Fehlerbehandlung

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

```
int main (void)
{
    FILE *f = fopen ("fhello.txt", "w");
    if (f)
    {
        fprintf (f, "Hello, _world!\n");
        fclose (f);
    }
    else
    {
        char *msg = strerror (errno);
        fprintf (stderr, "%s\n", msg);
    }
    return 0;
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.
- Die globale Variable `int errno` enthält dann die Nummer des Fehlers. Benötigt: `#include <errno.h>`
- Die Funktion `strerror()` wandelt `errno` in einen Fehlermeldungstext um. Benötigt: `#include <string.h>`

2.13 Dateien und Fehlerbehandlung

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

```
#include <errno.h>
```

```
#include <error.h>
```

```
int main (void)
```

```
{
```

```
    FILE *f = fopen ("fhello.txt", "w");
```

```
    if (!f)
```

```
        error (-1, errno, "cannot_open_file");
```

```
    fprintf (f, "Hello,_world!\n");
```

```
    fclose (f);
```

```
    return 0;
```

```
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.
- Die globale Variable `int errno` enthält dann die Nummer des Fehlers. Benötigt: `#include <errno.h>`
- Die Funktion `strerror()` wandelt `errno` in einen Fehlermeldungstext um. Benötigt: `#include <string.h>`
- Die Funktion `error()` gibt eine Fehlermeldung aus und beendet das Programm. Benötigt: `#include <error.h>`

2.13 Dateien und Fehlerbehandlung

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

```
int main (void)
```

```
{
    FILE *f = fopen ("fhello.txt", "w");
    if (!f)
        error (-1, errno, "cannot_open_file");
    fprintf (f, "Hello,_world!\n");
    fclose (f);
    return 0;
}
```

- Wenn die Datei nicht geöffnet werden kann, gibt `fopen()` den Wert `NULL` zurück.
- Die globale Variable `int errno` enthält dann die Nummer des Fehlers. Benötigt: `#include <errno.h>`
- Die Funktion `strerror()` wandelt `errno` in einen Fehlermeldungstext um. Benötigt: `#include <string.h>`
- Die Funktion `error()` gibt eine Fehlermeldung aus und beendet das Programm. Benötigt: `#include <error.h>`
- **Niemals Fehler einfach ignorieren!**

2.14 Parameter des Hauptprogramms

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

```
int main (int argc, char **argv)
{
    printf ("argc=%d\n", argc);
    for (int i = 0; i < argc; i++)
        printf ("argv[%d]=\n%s\n", i, argv[i]);
    return 0;
}
```


2.14 Parameter des Hauptprogramms

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

```
int main (int argc, char **argv)
{
    printf ("argc=%d\n", argc);
    for (int i = 0; *argv; i++, argv++)
        printf ("argv[%d]=\n%s\n", i, *argv);
    return 0;
}
```

2.15 String-Operationen

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

```
int main (void)
```

```
{
```

```
    char name[100];
```

```
    printf ("Ihr_Name:_");
```

```
    fgets (name, 100, stdin);
```

```
    printf ("Hallo,_%s!\n", name);
```

```
    return 0;
```

```
}
```

Probleme mit `scanf ("%s", name)`:

- Leerzeichen beendet Eingabe
- keine Prüfung der Puffergröße

2.15 String-Operationen

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

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char hello[] = "Hello,_world!\n";
```

```
    printf ("%s\n", hello);
```

```
    printf ("%zd\n", strlen (hello));
```

```
    printf ("%s\n", hello + 7);
```

```
    printf ("%zd\n", strlen (hello + 7));
```

```
    hello[5] = 0;
```

```
    printf ("%s\n", hello);
```

```
    printf ("%zd\n", strlen (hello));
```

```
    return 0;
```

```
}
```

2.15 String-Operationen

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

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char *anton = "Anton";
```

```
    char *zacharias = "Zacharias";
```

```
    printf ("%d\n", strcmp (anton, zacharias));
```

```
    printf ("%d\n", strcmp (zacharias, anton));
```

```
    printf ("%d\n", strcmp (anton, anton));
```

```
    char buffer[100] = "Huber_";
```

```
    strcat (buffer, anton);
```

```
    printf ("%s\n", buffer);
```

```
    return 0;
```

```
}
```

2.15 String-Operationen

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

```
#include <string.h>
```

```
int main (void)
```

```
{
```

```
    char buffer[100] = "";
```

```
    sprintf (buffer, "Die_Antwort_lautet:%d", 42);
```

```
    printf ("%s\n", buffer);
```

```
    char *answer = strstr (buffer, "Antwort");
```

```
    printf ("%s\n", answer);
```

```
    printf ("found_at:%zd\n", answer - buffer);
```

```
    return 0;
```

```
}
```

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

2 Wiederholung: Programmieren in C

...

2.7 Seiteneffekte

2.8 Strukturierte Programmierung

2.9 Funktionen

2.10 Zeiger

2.11 Arrays und Strings

2.12 Strukturen

2.13 Dateien und Fehlerbehandlung

2.14 Parameter des Hauptprogramms

2.15 String-Operationen

2.16 Bibliotheken

...

3 Einführung in C++

...



Änderungen
vorbehalten

2.16 Bibliotheken

2.16.1 Der Präprozessor

#include: Text einbinden

- **#include** <stdio.h>: Standard-Verzeichnisse – Standard-Header
- **#include** "answer.h": auch aktuelles Verzeichnis – eigene Header

#define VIER 4: Text ersetzen lassen – Konstante definieren

- Kein Semikolon!
- Berechnungen in Klammern setzen:
#define VIER (2 + 2)
- Konvention: Großbuchstaben

2.16 Bibliotheken

2.16.2 Bibliotheken einbinden

Inhalt der Header-Datei: externe Deklarationen

```
extern int answer (void);
```

```
extern int printf (__const char *__restrict __format, ...);
```

Funktion wird „anderswo“ definiert

- separater C-Quelltext: mit an `gcc` übergeben
- vorcompilierte Bibliothek: `-lfoo`
= Datei `libfoo.a` in Standard-Verzeichnis

2.16 Bibliotheken

2.16.3 Bibliothek verwenden (Beispiel: OpenGL)

- Include-Dateien:

```
#include <GL/gl.h>
```

```
#include <GL/glu.h>
```

```
#include <GL/glut.h>
```

- Compiler-Aufruf:

```
gcc -Wall -O cube.c -lGL -lGLU -lglut -o cube
```

2.16 Bibliotheken

2.16.3 Bibliothek verwenden (Beispiel: OpenGL)

Selbst geschriebene Funktion übergeben: *Callback*

```
void draw (void)
```

```
{
```

```
    ...
```

```
}
```

```
...
```

```
glutDisplayFunc (draw);
```

→ OpenGL ruft immer dann, wenn es etwas zu zeichnen gibt, die Funktion **draw** auf.

2.16 Bibliotheken

2.16.3 Bibliothek verwenden (Beispiel: OpenGL)

Selbst geschriebene Funktion übergeben: *Callback*

```
void key_handler (unsigned char key, int x, int y)
```

```
{
```

```
    ...
```

```
}
```

```
...
```

gedrückte Taste

Mausposition

```
glutKeyboardFunc (key_handler);
```


→ OpenGL ruft immer dann, wenn eine Taste gedrückt wurde, die Funktion `key_handler` auf.

2.16 Bibliotheken

2.16.3 Bibliothek verwenden (Beispiel: OpenGL)

Selbst geschriebene Funktion übergeben: *Callback*

```
void timer_handler (int value)
{
    t += 0.05;
    glutPostRedisplay ();
    glutTimerFunc (50, timer_handler, 0);
}
```

 benutzerdefinierte Daten

...

```
glutTimerFunc (50, timer_handler, 0);
```

→ OpenGL ruft nach 50 Millisekunden die Funktion `timer_handler` auf und übergibt ihr für den Parameter `value` den Wert `0`.

2.16 Bibliotheken

2.16.4 Projekt organisieren: make

- Regeln
- Makros

2.16 Bibliotheken

2.16.4 Projekt organisieren: make

- Regeln

```
philosophy: philosophy.o answer.o  
           gcc philosophy.o answer.o -o philosophy
```

```
answer.o: answer.c  
          gcc -c answer.c -o answer.o
```

```
philosophy.o: philosophy.c answer.h  
             gcc -c philosophy.c -o philosophy.o
```

- Makros

2.16 Bibliotheken

2.16.4 Projekt organisieren: make

- Regeln
- Makros

PHILOSOPHY_SOURCES = philosophy.c answer.h

PHILOSOPHY_OBJECTS = philosophy.o answer.o

ANSWER_SOURCES = answer.c

philosophy: \$(PHILOSOPHY_OBJECTS)
gcc \$(PHILOSOPHY_OBJECTS) -o philosophy

answer.o: \$(ANSWER_SOURCES)
gcc -c answer.c -o answer.o

philosophy.o: \$(PHILOSOPHY_SOURCES)
gcc -c philosophy.c -o philosophy.o

clean:
rm -f \$(PHILOSOPHY_OBJECTS) philosophy

2.16 Bibliotheken

2.16.4 Projekt organisieren: make

- Regeln
- Makros

→ 3 Sprachen: C, Präprozessor, make

2.17 Algorithmen

2.17.1 Differentialgleichungen

$$\varphi'(t) = \omega(t)$$

$$\omega'(t) = -\frac{g}{l} \cdot \sin \varphi(t)$$

- Von Hand (analytisch): Lösung raten (Ansatz), Parameter berechnen
- Mit Computer (numerisch): Eulersches Polygonzugverfahren

```
phi += dt * omega;  
omega += - dt * g / l * sin (phi);
```

Praktikumsaufgabe im 5. Semester Bachelor: Basketball

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

5. November 2018

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

2 Wiederholung: Programmieren in C

...

2.11 Arrays und Strings

2.12 Strukturen

2.13 Dateien und Fehlerbehandlung

2.14 Parameter des Hauptprogramms

2.15 String-Operationen

2.16 Bibliotheken

2.17 Algorithmen

2.18 Bit-Operationen

2.19 Dynamische Speicherverwaltung

...

3 Einführung in C++

...



Änderungen
vorbehalten

2.17 Algorithmen

2.17.1 Differentialgleichungen

$$\varphi'(t) = \omega(t)$$

$$\omega'(t) = -\frac{g}{l} \cdot \sin \varphi(t)$$

- Von Hand (analytisch): Lösung raten (Ansatz), Parameter berechnen
- Mit Computer (numerisch): Eulersches Polygonzugverfahren

```
phi += dt * omega;  
omega += - dt * g / l * sin (phi);
```

Praktikumsaufgabe im 5. Semester Bachelor: Basketball

2.17 Algorithmen

2.17.2 Rekursion

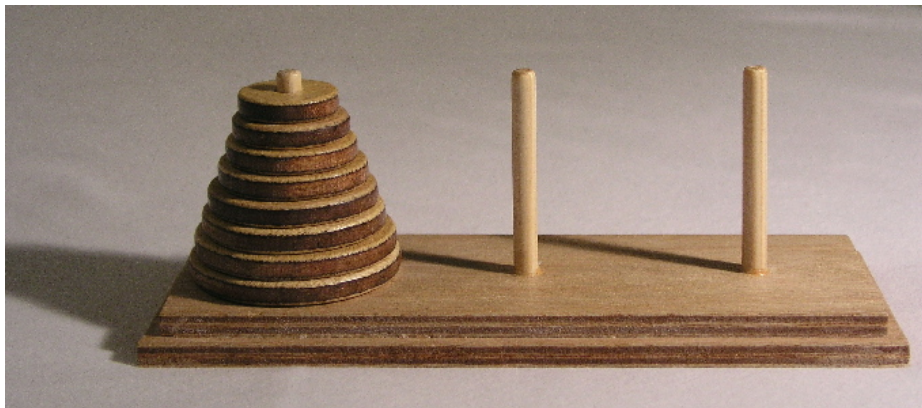
Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

2.17 Algorithmen

2.17.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi



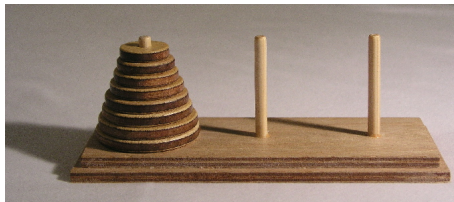
2.17 Algorithmen

2.17.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.



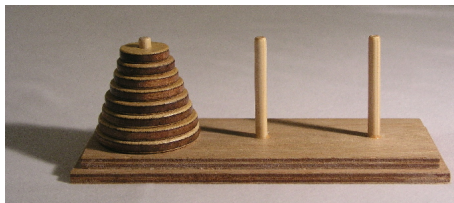
2.17 Algorithmen

2.17.2 Rekursion

Vollständige Induktion:
$$\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$$

Türme von Hanoi

- 64 Scheiben, 3 Plätze, immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar: schiebe $n - 1$ Scheiben auf Hilfsplatz, verschiebe die darunterliegende, hole $n - 1$ Scheiben von Hilfsplatz



2.17 Algorithmen

2.17.2 Rekursion

Vollständige Induktion:
$$\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$$

Türme von Hanoi

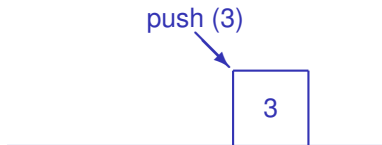
- 64 Scheiben, 3 Plätze,
immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben
auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar:
schiebe $n - 1$ Scheiben auf Hilfsplatz,
verschiebe die darunterliegende,
hole $n - 1$ Scheiben von Hilfsplatz

```
void verschiebe (int n, int start, int ziel)
{
    if (n == 1)
        verschiebe_1_scheibe (start, ziel);
    else
    {
        verschiebe (1, start, hilfsplatz);
        verschiebe (n - 1, start, ziel);
        verschiebe (1, hilfsplatz, ziel);
    }
}
```

2.17 Algorithmen

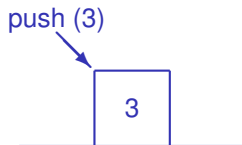
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

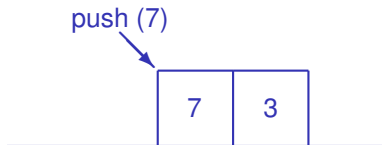


LIFO = Stack = Stapel

2.17 Algorithmen

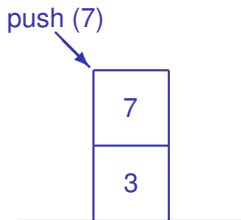
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

2.17 Algorithmen

2.17.3 Stack und FIFO

„First In – First Out“

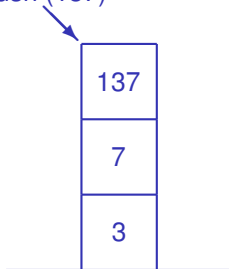
push (137)



FIFO = Queue = Reihe

„Last In – First Out“

push (137)



LIFO = Stack = Stapel

2.17 Algorithmen

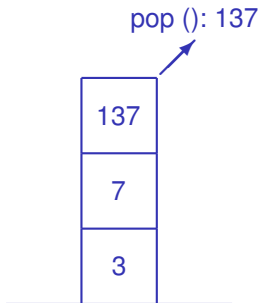
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

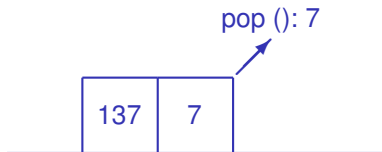


LIFO = Stack = Stapel

2.17 Algorithmen

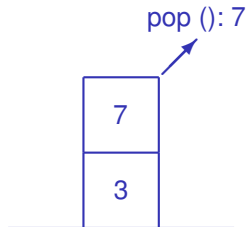
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

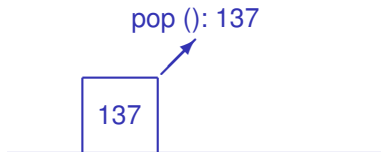


LIFO = Stack = Stapel

2.17 Algorithmen

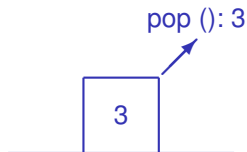
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



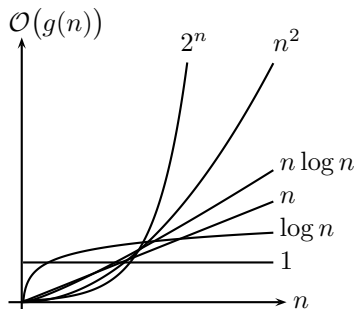
LIFO = Stack = Stapel

2.17 Algorithmen

2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen



n : Eingabedaten

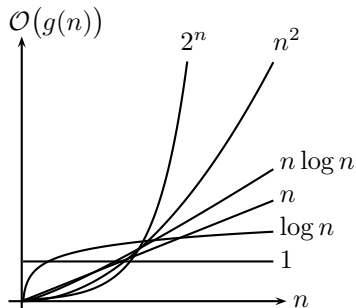
$g(n)$: Rechenzeit

2.17 Algorithmen

2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$



n : Eingabedaten

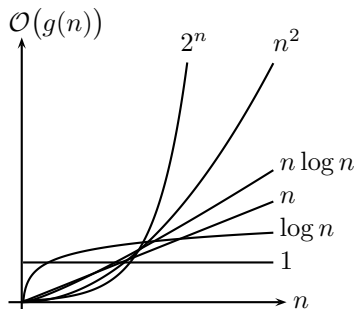
$g(n)$: Rechenzeit

2.17 Algorithmen

2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort



n : Eingabedaten

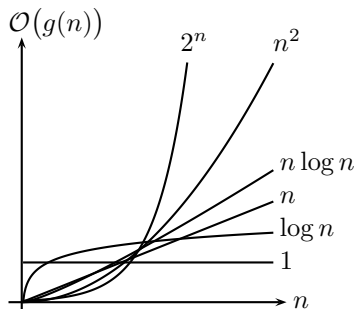
$g(n)$: Rechenzeit

2.17 Algorithmen

2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$



n : Eingabedaten

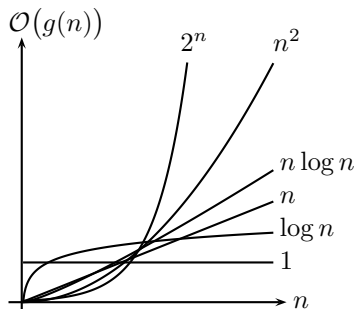
$g(n)$: Rechenzeit

2.17 Algorithmen

2.17.4 Aufwandsabschätzungen

Beispiel: Sortialgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen,
abbrechen, falls schon sortiert
→ Bubblesort



n : Eingabedaten

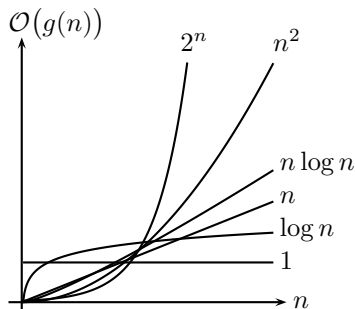
$g(n)$: Rechenzeit

2.17 Algorithmen

2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen,
abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$



n : Eingabedaten

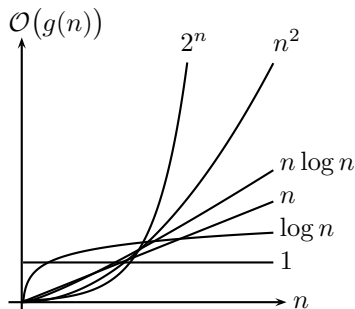
$g(n)$: Rechenzeit

2.17 Algorithmen

2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen,
abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Rekursiv sortieren
→ Quicksort



n : Eingabedaten

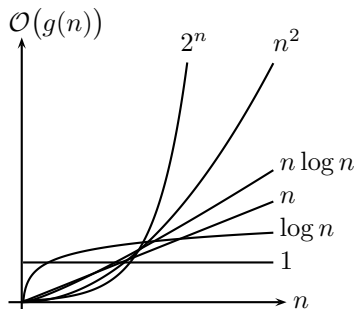
$g(n)$: Rechenzeit

2.17 Algorithmen

2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen, abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Rekursiv sortieren
→ Quicksort: $\mathcal{O}(n \log n)$ bis $\mathcal{O}(n^2)$



n : Eingabedaten

$g(n)$: Rechenzeit

2.18 Bit-Operationen

2.18.1 Zahlensysteme

Basis		Beispiel
2	Binärsystem	1 0000 0011
8	Oktalsystem	0403
10	Dezimalsystem	259
16	Hexadezimalsystem	0x103
256	IP-Adressen (IPv4)	0.0.1.3

2.18 Bit-Operationen

2.18.1 Zahlensysteme

Oktal- und Hexadezimal-Zahlen lassen sich ziffernweise in Binär-Zahlen umrechnen:

000	0	0000	0	1000	8
001	1	0001	1	1001	9
010	2	0010	2	1010	A
011	3	0011	3	1011	B
100	4	0100	4	1100	C
101	5	0101	5	1101	D
110	6	0110	6	1110	E
111	7	0111	7	1111	F

2.18 Bit-Operationen

2.18.2 Bit-Operationen in C

C-Operator	Verknüpfung	Anwendung
&	Und	Bits gezielt löschen
	Oder	Bits gezielt setzen
^	Exklusiv-Oder	Bits gezielt invertieren
~	Nicht	Alle Bits invertieren
<<	Verschiebung nach links	Maske generieren
>>	Verschiebung nach rechts	Bits isolieren

2.19 Dynamische Speicherverwaltung

```
char *name[] = { "Anna", "Berthold", "Caesar" };
```

```
...
```

```
name[3] = "Dieter";
```

2.19 Dynamische Speicherverwaltung

```
#include <stdlib.h>
```

```
...
```

```
char **name = malloc (3 * sizeof (char *));  
    /* Speicherplatz für 3 Zeiger anfordern */
```

```
...
```

```
free (name)  
    /* Speicherplatz freigeben */
```

Aufgabe: Schreiben Sie eine C-Bibliothek, die ein dynamisches „Array von Bits“ realisiert, z. B. mittels der folgenden Funktionen:

<code>bit_array *create_bit_array (int size);</code>	Bit-Array dynamisch erzeugen
<code>void free_bit_array (bit_array *b);</code>	Bit-Array wieder freigeben
<code>void set_bit (bit_array *b, int i);</code>	Bei Index i auf 1 setzen
<code>void clear_bit (bit_array *b, int i);</code>	Bei Index i auf 0 setzen
<code>int get_bit (bit_array *b, int i);</code>	Bei Index i lesen

Hinweise:

- Der Datentyp `bit_array` ist – sinnvollerweise in der `.h`-Datei – selbst zu definieren, z. B. als ein `struct`, das einen Zeiger auf die eigentlichen Daten, eine Größenangabe und evtl. noch zusätzliche Daten enthält.
- Die Benutzung des Bit-Arrays soll vollständig durch die o. a. Funktionen („Methoden“) erfolgen. Der Benutzer braucht sich insbesondere nicht damit zu beschäftigen, wie das Bit-Array intern aufgebaut ist.
- Sie benötigen ein Array, z. B. von `char`- oder `int`-Variablen, am besten `uint8_t` oder einen größeren Typ (aus `stdint.h`).
- Sie benötigen eine Division (`/`) sowie den Divisionsrest (Modulo: `%`).
- Die Größe des Bit-„Arrays“ wird über den Aufruf der Funktion `create_bit_array()` dynamisch festgelegt.

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

26. November 2018

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

2 Wiederholung: Programmieren in C

...

2.17 Algorithmen

2.18 Bit-Operationen

2.19 Dynamische Speicherverwaltung

2.20 Objektorientierte Programmierung

3 Einführung in C++

3.1 Motivation

3.2 Elementare Neuerungen gegenüber C

3.3 Referenz-Typen

3.4 Überladbare Operatoren und Funktionen

3.5 Namensräume

3.6 Objekte

...

...



Änderungen
vorbehalten

2.17 Algorithmen

2.17.1 Differentialgleichungen

$$\varphi'(t) = \omega(t)$$

$$\omega'(t) = -\frac{g}{l} \cdot \sin \varphi(t)$$

- Von Hand (analytisch): Lösung raten (Ansatz), Parameter berechnen
- Mit Computer (numerisch): Eulersches Polygonzugverfahren

```
phi += dt * omega;  
omega += - dt * g / l * sin (phi);
```

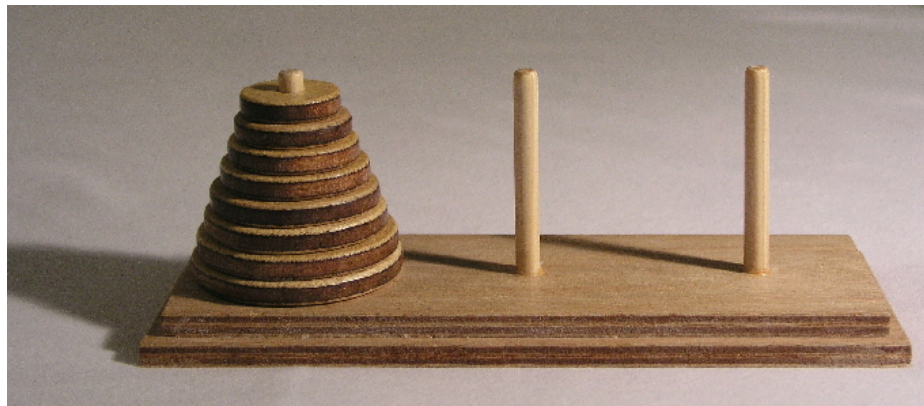
Praktikumsaufgabe im 5. Semester Bachelor: Basketball

2.17 Algorithmen

2.17.2 Rekursion

Vollständige Induktion: $\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$

Türme von Hanoi



2.17 Algorithmen

2.17.2 Rekursion

Vollständige Induktion:
$$\left. \begin{array}{l} \text{Aussage gilt für } n = 1 \\ \text{Schluß von } n - 1 \text{ auf } n \end{array} \right\} \text{ Aussage gilt für alle } n \in \mathbb{N}$$

Türme von Hanoi

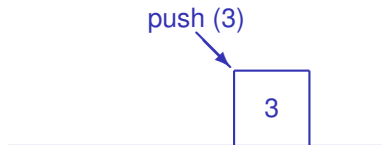
- 64 Scheiben, 3 Plätze,
immer 1 Scheibe verschieben
- Ziel: Turm verschieben
- Es dürfen nur kleinere Scheiben
auf größeren liegen.
- $n = 1$ Scheibe: fertig
- Wenn $n - 1$ Scheiben verschiebbar:
schiebe $n - 1$ Scheiben auf Hilfsplatz,
verschiebe die darunterliegende,
hole $n - 1$ Scheiben von Hilfsplatz

```
void verschiebe (int n, int start, int ziel)
{
    if (n == 1)
        verschiebe_1_scheibe (start, ziel);
    else
    {
        verschiebe (1, start, hilfsplatz);
        verschiebe (n - 1, start, ziel);
        verschiebe (1, hilfsplatz, ziel);
    }
}
```

2.17 Algorithmen

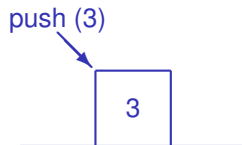
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

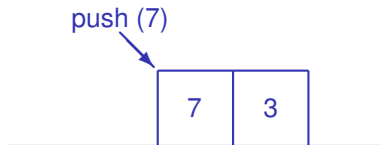


LIFO = Stack = Stapel

2.17 Algorithmen

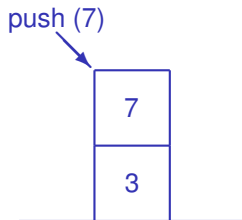
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



LIFO = Stack = Stapel

2.17 Algorithmen

2.17.3 Stack und FIFO

„First In – First Out“

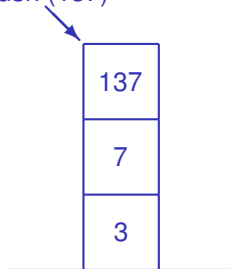
push (137)



FIFO = Queue = Reihe

„Last In – First Out“

push (137)



LIFO = Stack = Stapel

2.17 Algorithmen

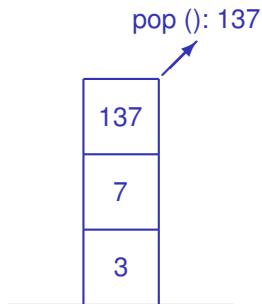
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

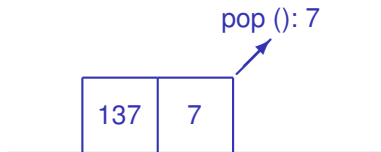


LIFO = Stack = Stapel

2.17 Algorithmen

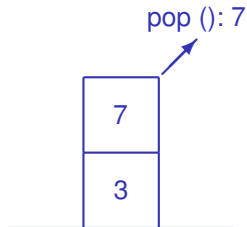
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“

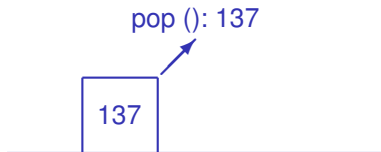


LIFO = Stack = Stapel

2.17 Algorithmen

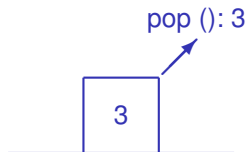
2.17.3 Stack und FIFO

„First In – First Out“



FIFO = Queue = Reihe

„Last In – First Out“



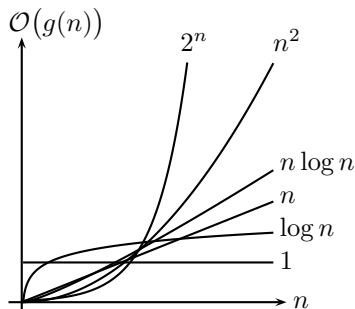
LIFO = Stack = Stapel

2.17 Algorithmen

2.17.4 Aufwandsabschätzungen

Beispiel: Sortieralgorithmen

- Maximum suchen: $\mathcal{O}(n)$
- Maximum ans Ende tauschen
→ Selectionsort: $\mathcal{O}(n^2)$
- Während Maximumsuche prüfen,
abbrechen, falls schon sortiert
→ Bubblesort: $\mathcal{O}(n)$ bis $\mathcal{O}(n^2)$
- Rekursiv sortieren
→ Quicksort: $\mathcal{O}(n \log n)$ bis $\mathcal{O}(n^2)$



n : Eingabedaten

$g(n)$: Rechenzeit

2.18 Bit-Operationen

2.18.1 Zahlensysteme

Basis	Name	Beispiel	Anwendung
2	Binärsystem	1 0000 0011	Bit-Operationen
8	Oktalsystem	0403	Dateizugriffsrechte (Unix)
10	Dezimalsystem	259	Alltag
16	Hexadezimalsystem	0x103	Bit-Operationen
256	(keiner gebräuchlich)	0.0.1.3	IP-Adressen (IPv4)

- Computer rechnen im Binärsystem.
- Für viele Anwendungen (z. B. I/O-Ports, Grafik, ...) ist es notwendig, Bits in Zahlen einzeln ansprechen zu können.

2.18 Bit-Operationen

2.18.1 Zahlensysteme

000	0	0000	0	1000	8
001	1	0001	1	1001	9
010	2	0010	2	1010	A
011	3	0011	3	1011	B
100	4	0100	4	1100	C
101	5	0101	5	1101	D
110	6	0110	6	1110	E
111	7	0111	7	1111	F

- Oktal- und Hexadezimalzahlen lassen sich ziffernweise in Binär-Zahlen umrechnen.
- Hexadezimalzahlen sind eine Kurzschreibweise für Binärzahlen, gruppiert zu jeweils 4 Bits.
- Oktalzahlen sind eine Kurzschreibweise für Binärzahlen, gruppiert zu jeweils 3 Bits.
- Trotz Taschenrechner u. ä. lohnt es sich, die o. a. Umrechnungstabelle **auswendig** zu kennen.

2.18 Bit-Operationen

2.18.2 Bit-Operationen in C

C-Operator	Verknüpfung	Anwendung
&	Und	Bits gezielt löschen
	Oder	Bits gezielt setzen
^	Exklusiv-Oder	Bits gezielt invertieren
~	Nicht	Alle Bits invertieren
<<	Verschiebung nach links	Maske generieren
>>	Verschiebung nach rechts	Bits isolieren

Numerierung der Bits: von rechts ab 0

Bit Nr. 3 auf 1 setzen: `a |= 1 << 3;`

Bit Nr. 4 auf 0 setzen: `a &= ~(1 << 4);`

Bit Nr. 0 invertieren: `a ^= 1 << 0;`

Abfrage, ob Bit Nr. 1 gesetzt ist: `if (a & (1 << 1))`

2.19 Dynamische Speicherverwaltung

```
char *name[] = { "Anna", "Berthold", "Caesar" };
```

```
...
```

```
name[3] = "Dieter";
```

2.19 Dynamische Speicherverwaltung

```
#include <stdlib.h>
```

```
...
```

```
char **name = malloc (3 * sizeof (char *));  
    /* Speicherplatz für 3 Zeiger anfordern */
```

```
...
```

```
free (name)  
    /* Speicherplatz freigeben */
```

Aufgabe: Schreiben Sie eine C-Bibliothek, die ein dynamisches „Array von Bits“ realisiert, z. B. mittels der folgenden Funktionen:

<code>bit_array *create_bit_array (int size);</code>	Bit-Array dynamisch erzeugen
<code>void free_bit_array (bit_array *b);</code>	Bit-Array wieder freigeben
<code>void set_bit (bit_array *b, int i);</code>	Bei Index i auf 1 setzen
<code>void clear_bit (bit_array *b, int i);</code>	Bei Index i auf 0 setzen
<code>int get_bit (bit_array *b, int i);</code>	Bei Index i lesen

Hinweise:

- Der Datentyp `bit_array` ist – sinnvollerweise in der `.h`-Datei – selbst zu definieren, z. B. als ein `struct`, das einen Zeiger auf die eigentlichen Daten, eine Größenangabe und evtl. noch zusätzliche Daten enthält.
- Die Benutzung des Bit-Arrays soll vollständig durch die o. a. Funktionen („Methoden“) erfolgen. Der Benutzer braucht sich insbesondere nicht damit zu beschäftigen, wie das Bit-Array intern aufgebaut ist.
- Sie benötigen ein Array, z. B. von `char`- oder `int`-Variablen, am besten `uint8_t` oder einen größeren Typ (aus `stdint.h`).
- Sie benötigen eine Division (`/`) sowie den Divisionsrest (Modulo: `%`).
- Die Größe des Bit-„Arrays“ wird über den Aufruf der Funktion `create_bit_array()` dynamisch festgelegt.

2.20 Objektorientierte Programmierung

2.20.1 Konzepte und Ziele

- Array: feste Anzahl von Elementen desselben Typs (z. B.: 3 Zeiger)
- Dynamisches Array: variable Anzahl von Elementen desselben Typs
- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern

2.20 Objektorientierte Programmierung

2.20.1 Konzepte und Ziele

- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern
- Zeiger auf verschiedene Strukturen mit einem gemeinsamen Anteil von Datenfeldern
→ „verwandte“ *Objekte*, *Klassen* von Objekten
- Struktur, die *nur* den gemeinsamen Anteil enthält
→ „Vorfahr“, *Basisklasse*, *Vererbung*
- Explizite Typumwandlung eines Zeigers auf die Basisklasse in einen Zeiger auf die *abgeleitete Klasse*
→ Man kann ein Array unterschiedlicher Objekte in einer Schleife abarbeiten.
→ *Polymorphie*

2.20 Objektorientierte Programmierung

2.20.1 Konzepte und Ziele

- *Objekte, Klassen, Basisklassen, abgeleitete Klassen*
- *Vererbung, Polymorphie*
- Funktionen, die mit dem Objekt arbeiten: *Methoden*
- Aufgerufene Funktion hängt vom Typ des Objekts ab: *virtuelle Methode*
- Realisierung über Zeiger, die im Objekt gespeichert sind
(Genaugenommen: Tabelle von Zeigern)

2.20 Objektorientierte Programmierung

2.20.2 Beispiel: Zahlen und Buchstaben

```
typedef struct  
{  
    int type;  
} t_base;
```

```
typedef struct  
{  
    int type;  
    int content;  
} t_integer;
```

```
typedef struct  
{  
    int type;  
    char *content;  
} t_string;
```

```
typedef struct
{
    int type;
} t_base;
```

```
typedef struct
{
    int type;
    int content;
} t_integer;
```

```
typedef struct
{
    int type;
    char *content;
} t_string;
```

```
t_integer i = { 1, 42 };
t_string s = { 2, "Hello,_world!" };
```

```
t_base *object[] = { (t_base *) &i, (t_base *) &s };
```


explizite
Typumwandlung

2.20 Objektorientierte Programmierung

2.20.2 Beispiel: Zahlen und Buchstaben

Weitere Beispiele:

- Editor für graphische Objekte
- Datenbank-Software
- graphische Benutzeroberfläche (GUI)

2.20 Objektorientierte Programmierung

2.20.3 Objektorientierte Programmierung in C

```
typedef struct  
{  
    int type;  
} t_base;
```

```
typedef struct  
{  
    int type;  
    int content;  
} t_integer;
```

```
typedef struct  
{  
    int type;  
    char *content;  
} t_string;
```

2.20 Objektorientierte Programmierung

2.20.3 Objektorientierte Programmierung in C++

```
struct TBase  
{  
};
```

```
struct TInteger: public TBase  
{  
    int content;  
};
```

```
struct TString: public TBase  
{  
    char *content;  
};
```

3 Einführung in C++

3.1 Motivation

- Vermeidung unsicherer Techniken, insbesondere von Präprozessor-Konstruktionen und Zeigern, unter Beibehaltung der Effizienz
- Compiler-Unterstützung für objektorientierte Programmierung

3.2 Elementare Neuerungen gegenüber C

3.2 Elementare Neuerungen gegenüber C

- Kommentare mit //

3.2 Elementare Neuerungen gegenüber C

- Kommentare mit //
- Konstante:
const int answer = 42;

3.2 Elementare Neuerungen gegenüber C

- Kommentare mit //
- Konstante:
const int n = 5;
int prime[n] = { 2, 3, 5, 7, 11 };

3.2 Elementare Neuerungen gegenüber C

- Kommentare mit //
- Konstante:
const int n = 5;
int prime[n] = { 2, 3, 5, 7, 11 };
- Ab C++11: **constexpr**-Funktionen

3.2 Elementare Neuerungen gegenüber C

- Kommentare mit //
- Konstante:
const int n = 5;
int prime[n] = { 2, 3, 5, 7, 11 };
- Ab C++11: **constexpr**-Funktionen
darf nur aus einem einzigen **return**-Statement bestehen
→ keine Schleife, aber Rekursion erlaubt

3.2 Elementare Neuerungen gegenüber C

- Kommentare mit //
- Konstante:
const int n = 5;
int prime[n] = { 2, 3, 5, 7, 11 };
- Ab C++11: **constexpr**-Funktionen
darf nur aus einem einzigen **return**-Statement bestehen
→ keine Schleife, aber Rekursion erlaubt
- Ab C++14: auch „normale“ Funktionen erlaubt

3.2 Elementare Neuerungen gegenüber C

- Kommentare mit //
- Konstante:
`const int n = 5;`
`int prime[n] = { 2, 3, 5, 7, 11 };`
- Ab C++11: `constexpr`-Funktionen
darf nur aus einem einzigen `return`-Statement bestehen
—→ keine Schleife, aber Rekursion erlaubt
- Ab C++14: auch „normale“ Funktionen erlaubt
- leere Parameterliste: `void` optional

3.2 Elementare Neuerungen gegenüber C

- Kommentare mit //
- Konstante:
const int n = 5;
int prime[n] = { 2, 3, 5, 7, 11 };
- Ab C++11: **constexpr**-Funktionen
darf nur aus einem einzigen **return**-Statement bestehen
—> keine Schleife, aber Rekursion erlaubt
- Ab C++14: auch „normale“ Funktionen erlaubt
- leere Parameterliste: **void** optional
in C: ohne **void** = Parameterliste wird nicht geprüft

3.2 Elementare Neuerungen gegenüber C

- Kommentare mit //
- Konstante:
const int n = 5;
int prime[n] = { 2, 3, 5, 7, 11 };
- Ab C++11: **constexpr**-Funktionen
darf nur aus einem einzigen **return**-Statement bestehen
→ keine Schleife, aber Rekursion erlaubt
- Ab C++14: auch „normale“ Funktionen erlaubt
- leere Parameterliste: **void** optional
in C: ohne **void** = Parameterliste wird nicht geprüft
- Operatoren **new** und **delete**
als Alternative zu den Funktionen **malloc()** und **free()**

3.3 Referenz-Typen

```
void calc_answer (int &answer)
{
    answer = 42;
}
```

... als Alternative zu ...

```
void calc_answer (int *answer)
{
    *answer = 42;
}
```

- Zeiger „verborgen“, übersichtlicher und sicherer
- Es gibt keinen **NULL**-Wert.
→ Für verkettete Listen u. ä.: Tricks erforderlich

```
#include <iostream>
```

```
int main ()
```

```
{
```

```
    std::cout << "Hello, world!" << std::endl;
```

```
    return 0;
```

```
}
```

3.4 Überladbare Operatoren und Funktionen

```
#include <iostream>
```

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

Bemerkungen:

- Compilieren mit `g++` statt `gcc`:
C++-Bibliotheken mit einbinden
- Der Operator `<<` hat normalerweise keinen Seiteneffekt, hier schon.

3.4 Überladbare Operatoren und Funktionen

```
void print (const char *s)
```

```
{  
    printf ("%s", s);  
}
```

```
void print (int i)
```

```
{  
    printf ("%d", i);  
}
```

3.4 Überladbare Operatoren und Funktionen

```
void print (const char *s)
```

```
{  
    printf ("%s", s);  
}
```

```
void print (int i)
```

```
{  
    printf ("%d", i);  
}
```

Optionale Parameter:

```
void print (const char *s = "\n")
```

```
{  
    printf ("%s", s);  
}
```

3.4 Überladbare Operatoren und Funktionen

```
void print (const char *s)
```

```
{  
    printf ("%s", s);  
}
```

```
void print (int i)
```

```
{  
    printf ("%d", i);  
}
```

Für den Linker:
veränderte, eindeutige Namen



Optionale Parameter:

```
void print (const char *s = "\n")
```

```
{  
    printf ("%s", s);  
}
```

wird vom Compiler erledigt



3.4 Überladbare Operatoren und Funktionen

```
void print (const char *s)
```

```
{  
    printf ("%s", s);  
}
```

```
void print (int i)
```

```
{  
    printf ("%d", i);  
}
```

Für den Linker:
veränderte, eindeutige Namen

Wenn man das nicht will:
extern "C" { ... }

Optionale Parameter:

```
void print (const char *s = "\n")
```

```
{  
    printf ("%s", s);  
}
```

wird vom Compiler erledigt

3.5 Namensräume

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()  
{  
    cout << "Hello,_world!" << endl;  
    return 0;  
}
```

3.5 Namensräume

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()  
{  
    cout << "Hello,_world!" << endl;  
    return 0;  
}
```

```
namespace my_output  
{  
    ...  
}
```

```
using namespace my_output;
```

3.6 Objekte

```
struct TBase  
{  
};
```

```
struct TInteger: public TBase  
{  
    int content;  
};
```

```
struct TString: public TBase  
{  
    char *content;  
};
```

Aufgabe

Schreiben Sie eine Klasse, die eine (z. B. einfach verkettete) Liste von Objekten (z. B. Strings) verwaltet.

Anregungen:

- Einfügen eines Strings in die Klasse mit einem Operator, z. B. `+=`
- Geht es mit Referenzen anstelle von Zeigern?
- Operator `[]`, um die Liste wie ein Array ansprechen zu können (wenn auch mit $\mathcal{O}(n)$ statt $\mathcal{O}(1)$)
- Methode (z. B. `foreach()`), um eine Callback-Funktion für alle Listenelemente aufzurufen

3.7 Objekte: Zugriffsrechte

- `public`, `private`, `protected`
nicht nur Bürokratie, sondern auch Kapselung
(Maßnahme gegen „Namensraumverschmutzung“)
- **`struct`**: standardmäßig `public`
`class`: standardmäßig `private`
- `friend`-Funktionen und -Klassen
- Klasse als Namensraum:
`static`-„Member“-Variable
`static`-„Methoden“
Deklarationen von z. B. Konstanten und Typen

3.8 Objekte: Konstruktoren und Destruktoren

- leerer Standard-Konstruktor
- *Copy-Konstruktor*
- Konstruktor-Aufruf als „Initialisierung“
- Konstruktor-Aufruf mit `new`
Destruktor-Aufruf mit `delete`
- automatischer Destruktor-Aufruf
beim Verlassen des Gültigkeitsbereichs

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

3. Dezember 2018

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

2 Wiederholung: Programmieren in C

...

2.20 Objektorientierte Programmierung

3 Einführung in C++

3.1 Motivation

3.2 Elementare Neuerungen gegenüber C

3.3 Referenz-Typen

3.4 Überladbare Operatoren und Funktionen

3.5 Namensräume

3.6 Objekte

3.7 Objekte: Zugriffsrechte

3.8 Objekte: Konstruktoren und Destruktoren

3.9 Strings

3.10 Templates

...

...



Änderungen
vorbehalten

2.20 Objektorientierte Programmierung

2.20.1 Konzepte und Ziele

- Array: feste Anzahl von Elementen desselben Typs (z. B.: 3 Zeiger)
- Dynamisches Array: variable Anzahl von Elementen desselben Typs
- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern

2.20 Objektorientierte Programmierung

2.20.1 Konzepte und Ziele

- Problem: Elemente unterschiedlichen Typs
- Lösung: den Typ des Elements zusätzlich speichern
- Zeiger auf verschiedene Strukturen mit einem gemeinsamen Anteil von Datenfeldern
→ „verwandte“ *Objekte*, *Klassen* von Objekten
- Struktur, die *nur* den gemeinsamen Anteil enthält
→ „Vorfahr“, *Basisklasse*, *Vererbung*
- Explizite Typumwandlung eines Zeigers auf die Basisklasse in einen Zeiger auf die *abgeleitete Klasse*
→ Man kann ein Array unterschiedlicher Objekte in einer Schleife abarbeiten.
→ *Polymorphie*

2.20 Objektorientierte Programmierung

2.20.1 Konzepte und Ziele

- *Objekte, Klassen, Basisklassen, abgeleitete Klassen*
- *Vererbung, Polymorphie*
- Funktionen, die mit dem Objekt arbeiten: *Methoden*
- Aufgerufene Funktion hängt vom Typ des Objekts ab: *virtuelle Methode*
- Realisierung über Zeiger, die im Objekt gespeichert sind
(Genaugenommen: Tabelle von Zeigern)

2.20 Objektorientierte Programmierung

2.20.2 Beispiel: Zahlen und Buchstaben

```
typedef struct
{
    int type;
} t_base;
```

```
typedef struct
{
    int type;
    int content;
} t_integer;
```

```
typedef struct
{
    int type;
    char *content;
} t_string;
```

```
typedef struct
{
    int type;
} t_base;
```

```
typedef struct
{
    int type;
    int content;
} t_integer;
```

```
typedef struct
{
    int type;
    char *content;
} t_string;
```

```
t_integer i = { 1, 42 };
t_string s = { 2, "Hello,_world!" };
```

```
t_base *object[] = { (t_base *) &i, (t_base *) &s };
```


explizite
Typumwandlung

2.20 Objektorientierte Programmierung

2.20.2 Beispiel: Zahlen und Buchstaben

Weitere Beispiele:

- Editor für graphische Objekte
- Datenbank-Software
- graphische Benutzeroberfläche (GUI)

2.20 Objektorientierte Programmierung

2.20.3 Objektorientierte Programmierung in C

```
typedef struct  
{  
    int type;  
} t_base;
```

```
typedef struct  
{  
    int type;  
    int content;  
} t_integer;
```

```
typedef struct  
{  
    int type;  
    char *content;  
} t_string;
```


2.20 Objektorientierte Programmierung

2.20.3 Objektorientierte Programmierung in C++

```
struct TBase  
{  
};
```

```
struct TInteger: public TBase  
{  
    int content;  
};
```

```
struct TString: public TBase  
{  
    char *content;  
};
```

3 Einführung in C++

3.1 Motivation

- Vermeidung unsicherer Techniken, insbesondere von Präprozessor-Konstruktionen und Zeigern, unter Beibehaltung der Effizienz
- Compiler-Unterstützung für objektorientierte Programmierung

3.2 Elementare Neuerungen gegenüber C

- Kommentare mit //
- Konstante:
const int answer = 42;
const int n = 5;
int prime[n] = { 2, 3, 5, 7, 11 };
- Ab C++11: **constexpr**-Funktionen
darf nur aus einem einzigen **return**-Statement bestehen
→ keine Schleife, aber Rekursion erlaubt
- Ab C++14: auch „normale“ Funktionen erlaubt
- leere Parameterliste: **void** optional
in C: ohne **void** = Parameterliste wird nicht geprüft
- Operatoren **new** und **delete**
als Alternative zu den Funktionen **malloc()** und **free()**

3.3 Referenz-Typen

```
void calc_answer (int &answer)
{
    answer = 42;
}
```

... als Alternative zu ...

```
void calc_answer (int *answer)
{
    *answer = 42;
}
```

- Zeiger „verborgen“, übersichtlicher und sicherer
- Es gibt keinen **NULL**-Wert.
→ Für verkettete Listen u. ä.: Tricks erforderlich

3.4 Überladbare Operatoren und Funktionen

```
#include <iostream>
```

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

Bemerkungen:

- Compilieren mit `g++` statt `gcc`:
C++-Bibliotheken mit einbinden
- Der Operator `<<` hat normalerweise keinen Seiteneffekt, hier schon.

3.4 Überladbare Operatoren und Funktionen

```
void print (const char *s)
```

```
{  
    printf ("%s", s);  
}
```

```
void print (int i)
```

```
{  
    printf ("%d", i);  
}
```

Für den Linker:
veränderte, eindeutige Namen

Wenn man das nicht will:
extern "C" { ... }

Optionale Parameter:

```
void print (const char *s = "\n")
```

```
{  
    printf ("%s", s);  
}
```

wird vom Compiler erledigt

3.5 Namensräume

```
#include <iostream>
```

```
using namespace std;
```

```
int main ()  
{  
    cout << "Hello,_world!" << endl;  
    return 0;  
}
```

```
namespace my_output  
{  
    ...  
}
```

```
using namespace my_output;
```

3.6 Objekte

```
struct TBase  
{  
};
```

```
struct TInteger: public TBase  
{  
    int content;  
};
```

```
struct TString: public TBase  
{  
    char *content;  
};
```


Aufgabe

Schreiben Sie eine Klasse, die eine (z. B. einfach verkettete) Liste von Objekten (z. B. Strings) verwaltet.

Anregungen:

- Einfügen eines Strings in die Klasse mit einem Operator, z. B. `+=`
- Geht es mit Referenzen anstelle von Zeigern?
- Operator `[]`, um die Liste wie ein Array ansprechen zu können (wenn auch mit $\mathcal{O}(n)$ statt $\mathcal{O}(1)$)
- Methode (z. B. `foreach()`), um eine Callback-Funktion für alle Listenelemente aufzurufen

3.7 Objekte: Zugriffsrechte

- `public`, `private`, `protected`
nicht nur Bürokratie, sondern auch Kapselung
(Maßnahme gegen „Namensraumverschmutzung“)
- **`struct`**: standardmäßig `public`
`class`: standardmäßig `private`
- `friend`-Funktionen und -Klassen
- Klasse als Namensraum:
`static`-„Member“-Variable
`static`-„Methoden“
Deklarationen von z. B. Konstanten und Typen

3.8 Objekte: Konstruktoren und Destruktoren

- leerer Standard-Konstruktor
- *Copy-Konstruktor*
- Konstruktor-Aufruf als „Initialisierung“
- Konstruktor-Aufruf mit `new`
Destruktor-Aufruf mit `delete`
- automatischer Destruktor-Aufruf
beim Verlassen des Gültigkeitsbereichs

3.9 Strings

- **#include** <string>
- String-Klasse
- String-Konstante sind **const char ***
- C-kompatiblen String extrahieren: **c_str ()**
- In String schreiben: **#include** <sstream>, **ostringstream**

3.10 Templates

Anwendung desselben Quelltextes auf verschiedene Datentypen

- `template <typename x> ...`
- `template <class x> ...`

Beide Schreibweisen sind bedeutungsgleich.

3.10 Templates

Anwendung desselben Quelltextes auf verschiedene Datentypen

- `template <typename x> ...`
- `template <class x> ...`

Beide Schreibweisen sind bedeutungsgleich.

Vorsicht: Fehler werden erst bei Instantiierung erkannt!

3.10 Templates

Anwendung desselben Quelltextes auf verschiedene Datentypen

- `template <typename x> ...`
- `template <class x> ...`

Beide Schreibweisen sind bedeutungsgleich.

Vorsicht: Fehler werden erst bei Instantiierung erkannt!

- Template-Spezialisierung:
`template <> foo <int> ...`

3.11 Exceptions

```
try
{
    ...
    throw <value>;
    ...
}
catch (<type> <variable>)
{
    ...
}
catch ...
```

- Nach den `catch()`-Statements wird, soweit nicht anders programmiert, das Programm fortgesetzt.
- `throw`; (ohne Wert):
an übergeordneten Exception-Handler weiterreichen
- C-Äquivalent:
`setjmp()`, `longjmp()`
- speziell für `<type>`:
Nachfahren von `class exception`
- veraltet:
dynamic exception specifications

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

10. Dezember 2018

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

2 Wiederholung: Programmieren in C

3 Einführung in C++

...

3.8 Objekte: Konstruktoren und Destruktoren

3.9 Strings

3.10 Templates

3.11 Exceptions

3.12 Typ-Konversionen

4 Standard-Bibliotheken (STL)

4.1 Container-Templates

4.2 Iteratoren

5 C++11

...

6 Die Boost-Bibliothek

7 Plug-In-Architekturen



Änderungen
vorbehalten

3.9 Strings

- **#include** <string>
- String-Klasse
- String-Konstante sind **const char ***
- C-kompatiblen String extrahieren: **c_str ()**
- In String schreiben: **#include** <sstream>, **ostringstream**

3.10 Templates

Anwendung desselben Quelltextes auf verschiedene Datentypen

- `template <typename x> ...`
- `template <class x> ...`

Beide Schreibweisen sind bedeutungsgleich.

Vorsicht: Fehler werden erst bei Instantiierung erkannt!

- Template-Spezialisierung:
`template <> foo <int> ...`

3.11 Exceptions

```
try
{
    ...
    throw <value>;
    ...
}
catch (<type> <variable>)
{
    ...
}
catch ...
```

- Nach den `catch()`-Statements wird, soweit nicht anders programmiert, das Programm fortgesetzt.
- `throw`; (ohne Wert):
an übergeordneten Exception-Handler weiterreichen
- C-Äquivalent:
`setjmp()`, `longjmp()`
- speziell für `<type>`:
Nachfahren von `class exception`
- veraltet:
dynamic exception specifications

3.12 Typ-Konversionen

- In C:

```
char *hello = "Hello, world!";  
uint64_t address = (uint64_t) hello;  
printf ("%s" PRIu64 "\n", address);
```

- alternative Syntax in C++:

```
char *hello = "Hello, world!";  
uint64_t address = uint64_t (hello);  
cout << address << endl;
```

- zusätzlich in C++:

implizite und explizite Typumwandlung zwischen Zeigern auf Klassen

```
dynamic_cast<>()  
static_cast<>()  
reinterpret_cast<>()  
const_cast<>()
```

3.12 Typ-Konversionen

<http://www.cplusplus.com/doc/tutorial/typcasting/>

- Zuweisung: Zeiger auf abgeleitete Klasse an Zeiger auf Basisklasse
→ implizite Typumwandlung möglich
- Zuweisung: Zeiger auf Basisklasse an Zeiger auf abgeleitete Klasse
→ nur explizite Typumwandlung möglich:
`dynamic_cast<>()`, `static_cast<>()`
- implizite Typumwandlungen in der Klasse definieren:
 - Initialisierung durch Konstruktor
 - Zuweisungs-Operator
 - Typumwandlungsoperator
- implizite Typumwandlungen ausschalten:
Schlüsselwort `explicit`

3.12 Typ-Konversionen

<http://www.cplusplus.com/doc/tutorial/typecasting/>

- `dynamic_cast<>()`
Zuweisung: Zeiger auf Basisklasse an Zeiger auf abgeleitete Klasse
explizite Typumwandlung mit Prüfung, ggf. Exception
- `static_cast<>()`
Zuweisung: Zeiger auf Basisklasse an Zeiger auf abgeleitete Klasse
explizite Typumwandlung ohne Prüfung
- `reinterpret_cast<>()`
Typumwandlung ohne Prüfung zwischen Zeigern untereinander
und zwischen Zeigern und Integer-Typen
- `const_cast<>()`
„**const**“ ein- bzw. ausschalten

4 Standard-Bibliotheken (STL)

4.1 Container-Templates

array	Array mit fester Größe
bitset	festes Array von Bits (Booleans)
vector	dynamisches Array
vector <bool>	dynamisches Bit-Array
forward_list	einfach-verkettete Liste
list	doppelt-verkettete Liste
set	binärer Baum
multiset	mehrfache Elemente zulässig
unordered_set	Hash-Tabelle
unordered_multiset	mehrfache Elemente zulässig
map	binärer Baum mit separaten Schlüsselwerten
multimap	mehrere Elemente pro Schlüssel
unordered_map	Hash-Tabelle mit separaten Schlüsselwerten
unordered_multimap	mehrere Elemente pro Schlüssel
stack	Stack
queue	FIFO
deque	<i>double-ended queue</i>
priority_queue	geordneter Push-Pop-Container

4 Standard-Bibliotheken (STL)

4.2 Iteratoren

Pointer-Arithmetik:

```
int prime[5] = { 2, 3, 5, 7, 11 };  
for (int *p = prime; p != prime + 5; p++)  
    cout << *p << endl;
```

Iterator als Verallgemeinerung:

```
array<int, 5> prime = { { 2, 3, 5, 7, 11 } };  
for (array<int, 5>::iterator p = prime.begin (); p != prime.end (); p++)  
    cout << *p << endl;
```

5 C++11

5.1 Syntax-Erweiterungen

Allgemeine Syntax-Erweiterung: Datentyp **auto**

Anwendung auf Iterator:

```
array<int, 5> prime = { { 2, 3, 5, 7, 11 } };  
for (auto p = prime.begin (); p != prime.end (); p++)  
    cout << *p << endl;
```

Spezielle Syntax-Erweiterung: **for**-Schleife über Container mit Referenz statt „Zeiger“

```
array<int, 5> prime = { { 2, 3, 5, 7, 11 } };  
for (auto p: prime)  
    cout << p << endl;
```

5 C++11

5.2 Streng typisierte Aufzählungstypen

- `enum class`

5 C++11

5.3 Intelligente Zeiger

Warum?

- bereits freigegebene Zeiger werden u. U. weiterhin verwendet
- Speicherlecks
- uninitialisierte Zeiger
- `shared_ptr`
- `weak_ptr`
- `unique_ptr`
- `move()`

5 C++11

5.4 R-Wert-Referenztypen

- &&
- move()

Literatur:

- http://thbecker.net/articles/rvalue_references/section_01.html
- <http://www.artima.com/cppsource/rvalue.html>

5 C++11

5.5 Lambda-Ausdrücke

Übergabe von Funktionszeigern

```
int is_smaller (int a, int b)
{
    return a < b;
}
```

```
sort (numbers.begin (), numbers.end (), is_smaller);
```

Stattdessen: *Lambda-Ausdrücke*

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

- 1 Einführung
- 2 Wiederholung: Programmieren in C
- 3 Einführung in C++
 - ...
 - 3.12 Typ-Konversionen
- 4 Standard-Bibliotheken (STL)
 - 4.1 Container-Templates
 - 4.2 Iteratoren
- 5 C++11
 - 5.1 Syntax-Erweiterungen
 - 5.2 Streng typisierte Aufzählungstypen
 - 5.3 Intelligente Zeiger
 - 5.4 R-Wert-Referenztypen
 - 5.5 Lambda-Ausdrücke
- 6 Die Boost-Bibliothek
- 7 Plug-In-Architekturen



Änderungen
vorbehalten

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

17. Dezember 2018

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

1 Einführung

2 Wiederholung: Programmieren in C

3 Einführung in C++

...

3.12 Typ-Konversionen

4 Standard-Bibliotheken (STL)

4.1 Container-Templates

4.2 Iteratoren

5 C++11

5.1 Syntax-Erweiterungen

5.2 Streng typisierte Aufzählungstypen

5.3 Intelligente Zeiger

5.4 R-Wert-Referenztypen

5.5 Lambda-Ausdrücke

6 Die Boost-Bibliothek

7 Plug-In-Architekturen



Änderungen
vorbehalten

3.12 Typ-Konversionen

- In C:
`char *hello = "Hello, world!";`
`uint64_t address = (uint64_t) hello;`
`printf ("%s" PRIu64 "\n", address);`
- alternative Syntax in C++:
`char *hello = "Hello, world!";`
`uint64_t address = uint64_t (hello);`
`cout << address << endl;`
- zusätzlich in C++:
implizite und explizite Typumwandlung zwischen Zeigern auf Klassen
`dynamic_cast<>()`
`static_cast<>()`
`reinterpret_cast<>()`
`const_cast<>()`

3.12 Typ-Konversionen

<http://www.cplusplus.com/doc/tutorial/typecasting/>

- Zuweisung: Zeiger auf abgeleitete Klasse an Zeiger auf Basisklasse
→ implizite Typumwandlung möglich
- Zuweisung: Zeiger auf Basisklasse an Zeiger auf abgeleitete Klasse
→ nur explizite Typumwandlung möglich:
`dynamic_cast<>()`, `static_cast<>()`
- implizite Typumwandlungen in der Klasse definieren:
 - Initialisierung durch Konstruktor
 - Zuweisungs-Operator
 - Typumwandlungsoperator
- implizite Typumwandlungen ausschalten:
Schlüsselwort `explicit`

3.12 Typ-Konversionen

<http://www.cplusplus.com/doc/tutorial/typecasting/>

- `dynamic_cast<>()`
Zuweisung: Zeiger auf Basisklasse an Zeiger auf abgeleitete Klasse
explizite Typumwandlung mit Prüfung, ggf. Exception
- `static_cast<>()`
Zuweisung: Zeiger auf Basisklasse an Zeiger auf abgeleitete Klasse
explizite Typumwandlung ohne Prüfung
- `reinterpret_cast<>()`
Typumwandlung ohne Prüfung zwischen Zeigern untereinander
und zwischen Zeigern und Integer-Typen
- `const_cast<>()`
„**const**“ ein- bzw. ausschalten

4 Standard-Bibliotheken (STL)

4.1 Container-Templates

array	Array mit fester Größe
bitset	festes Array von Bits (Booleans)
vector	dynamisches Array
vector <bool>	dynamisches Bit-Array
forward_list	einfach-verkettete Liste
list	doppelt-verkettete Liste
set	binärer Baum
multiset	mehrfache Elemente zulässig
unordered_set	Hash-Tabelle
unordered_multiset	mehrfache Elemente zulässig
map	binärer Baum mit separaten Schlüsselwerten
multimap	mehrere Elemente pro Schlüssel
unordered_map	Hash-Tabelle mit separaten Schlüsselwerten
unordered_multimap	mehrere Elemente pro Schlüssel
stack	Stack
queue	FIFO
deque	<i>double-ended queue</i>
priority_queue	geordneter Push-Pop-Container

4 Standard-Bibliotheken (STL)

4.2 Iteratoren

Pointer-Arithmetik:

```
int prime[5] = { 2, 3, 5, 7, 11 };  
for (int *p = prime; p != prime + 5; p++)  
    cout << *p << endl;
```

Iterator als Verallgemeinerung:

```
array<int, 5> prime = { { 2, 3, 5, 7, 11 } };  
for (array<int, 5>::iterator p = prime.begin (); p != prime.end (); p++)  
    cout << *p << endl;
```

5 C++11

5.1 Syntax-Erweiterungen

Allgemeine Syntax-Erweiterung: Datentyp **auto**

Anwendung auf Iterator:

```
array<int, 5> prime = { { 2, 3, 5, 7, 11 } };  
for (auto p = prime.begin (); p != prime.end (); p++)  
    cout << *p << endl;
```

Spezielle Syntax-Erweiterung: **for**-Schleife über Container mit Referenz statt „Zeiger“

```
array<int, 5> prime = { { 2, 3, 5, 7, 11 } };  
for (auto p: prime)  
    cout << p << endl;
```


5 C++11

5.2 Streng typisierte Aufzählungstypen

- `enum class`

5 C++11

5.3 Intelligente Zeiger

Warum?

- bereits freigegebene Zeiger werden u. U. weiterhin verwendet
- Speicherlecks
- uninitialisierte Zeiger
- `shared_ptr`
- `weak_ptr`
- `unique_ptr`
- `move()`

5 C++11

5.4 R-Wert-Referenztypen

- &&
- move()

Literatur:

- http://thbecker.net/articles/rvalue_references/section_01.html
- <http://www.artima.com/cppsource/rvalue.html>

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

7. Januar 2019

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

- 1 Einführung
- 2 Wiederholung: Programmieren in C
- 3 Einführung in C++
- 4 Standard-Bibliotheken (STL)
- 5 C++11
 - ...
 - 5.3 Intelligente Zeiger
 - 5.4 R-Wert-Referenztypen
 - 5.5 Lambda-Ausdrücke
- 6 Die Boost-Bibliothek
 - 6.1 C++11-Erweiterungen für ältere Compiler
 - 6.2 Spracherweiterungen
 - 6.3 Zeitangaben
 - 6.4 Interprozeßkommunikation
 - 6.5 Asynchroner Input/Output
- 7 Plug-In-Architekturen



Änderungen
vorbehalten

5 C++11

5.4 R-Wert-Referenztypen

- &&
- move()

Literatur:

- http://thbecker.net/articles/rvalue_references/section_01.html
- <http://www.artima.com/cppsource/rvalue.html>

5 C++11

5.5 Lambda-Ausdrücke

Übergabe von Funktionszeigern

```
int is_smaller (int a, int b)
{
    return a < b;
}
```

```
sort (numbers.begin (), numbers.end (), is_smaller);
```

Stattdessen: *Lambda-Ausdrücke*

6 Die Boost-Bibliothek

Überblick: <http://dieboostcppbibliotheken.de/>

6.1 C++11-Erweiterungen für ältere Compiler

C++11

```
for (auto &i : container)
[](int i){ cout << i << endl; }
#include <cstdlib>
#include <random>
...
```

Boost

```
BOOST_FOREACH (int &i, container)
cout << boost::lambda::_1 << "\n"
#include <boost/cstdint.hpp>
#include <boost/random.hpp>
...
```


6 Die Boost-Bibliothek

6.2 Spracherweiterungen

- „Duck Typing“:
`boost::any`

6 Die Boost-Bibliothek

6.3 Zeitangaben

- Kalender (Schaltjahr-Problematik usw.)
- formatierte Ein-/Ausgabe
- Zeitzonen
- Zeitmessungen
- Messung der Code-Ausführungsgeschwindigkeit
- betriebssystemunabhängiges Warten:
`this_thread::sleep_for (chrono::milliseconds (137));`

6 Die Boost-Bibliothek

6.4 Interprozeßkommunikation

- Gemeinsamer Speicher: *Shared Memory*
- Kernel verwaltet benannte Speicherbereiche
- Existenz persistent, unabhängig von Prozessen
- Spezialfall von *Mapped Region*
- in C (Unix): `shm_open()`, `mmap()`, Bibliothek: `-lrt`
- in C++ (Boost): betriebssystemunabhängig
- ... außer, man verwendet MS-Windows-spezifischen Shared Memory, der beim Beenden des Prozesses automatisch entfernt wird

6 Die Boost-Bibliothek

6.4 Interprozeßkommunikation

- Verwalteter gemeinsamer Speicher: *Managed Shared Memory*
- Ein C++-map-Container verwaltet benannte Variable innerhalb des Shared Memory.
- erzeugen mit `construct<>`, verwenden mit `find<>`, beides mit `find_or_construct<>`
- wieder entfernen mit `destroy<>`, `destroy_ptr<>`
- Speicherüberlauf: `bad_alloc`-Exception

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

- 1 Einführung
- 2 Wiederholung: Programmieren in C
- 3 Einführung in C++
- 4 Standard-Bibliotheken (STL)
- 5 C++11
 - ...
 - 5.3 Intelligente Zeiger
 - 5.4 R-Wert-Referenztypen
 - 5.5 Lambda-Ausdrücke
- 6 Die Boost-Bibliothek
 - 6.1 C++11-Erweiterungen für ältere Compiler
 - 6.2 Spracherweiterungen
 - 6.3 Zeitangaben
 - 6.4 Interprozeßkommunikation
 - 6.5 Asynchroner Input/Output
- 7 Plug-In-Architekturen



Änderungen
vorbehalten

Vertiefung Software-Entwicklung in C++

Prof. Dr. rer. nat. Peter Gerwinski

14. Januar 2019

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

- 1 Einführung
- 2 Wiederholung: Programmieren in C
- 3 Einführung in C++
- 4 Standard-Bibliotheken (STL)
- 5 C++11
 - ...
 - 5.4 R-Wert-Referenztypen
 - 5.5 Lambda-Ausdrücke
- 6 Die Boost-Bibliothek
 - 6.1 C++11-Erweiterungen für ältere Compiler
 - 6.2 Spracherweiterungen
 - 6.3 Zeitangaben
 - 6.4 Interprozeßkommunikation
 - 6.5 Asynchroner Input/Output
- 7 Plug-In-Architekturen



Änderungen
vorbehalten

5 C++11

5.5 Lambda-Ausdrücke

Übergabe von Funktionszeigern

```
int is_smaller (int a, int b)
{
    return a < b;
}
```

```
sort (numbers.begin (), numbers.end (), is_smaller);
```

Stattdessen: *Lambda-Ausdrücke*

6 Die Boost-Bibliothek

Überblick: <http://dieboostcppbibliotheken.de/>

6.1 C++11-Erweiterungen für ältere Compiler

C++11

```
for (auto &i : container)
[](int i){ cout << i << endl; }
#include <cstdlib>
#include <random>
...
```

Boost

```
BOOST_FOREACH (int &i, container)
cout << boost::lambda::_1 << "\n"
#include <boost/cstdint.hpp>
#include <boost/random.hpp>
...
```

6 Die Boost-Bibliothek

6.2 Spracherweiterungen

- „Duck Typing“:
`boost::any`

6 Die Boost-Bibliothek

6.2 Spracherweiterungen

- „Duck Typing“:
`boost::any`
- Coroutinen:
kooperatives Multitasking (Alternative zu Threads)
anhalten und Wert-Rückgabe über `push_type`-Objekt,
aufrufen und Wert-Abfrage über `pull_type`-Objekt

6 Die Boost-Bibliothek

6.2 Spracherweiterungen

- „Duck Typing“:
`boost::any`
- Coroutinen:
kooperatives Multitasking (Alternative zu Threads)
anhalten und Wert-Rückgabe über `push_type`-Objekt,
aufrufen und Wert-Abfrage über `pull_type`-Objekt
- Parameter-Namen:
`BOOST_PARAMETER_NAME()`, `BOOST_PARAMETER_FUNCTION()`

6 Die Boost-Bibliothek

6.3 Zeitangaben

- Kalender (Schaltjahr-Problematik usw.)
- formatierte Ein-/Ausgabe
- Zeitzonen
- Zeitmessungen
- Messung der Code-Ausführungsgeschwindigkeit
- betriebssystemunabhängiges Warten:
`this_thread::sleep_for (chrono::milliseconds (137));`

6 Die Boost-Bibliothek

6.4 Interprozeßkommunikation

- Gemeinsamer Speicher: *Shared Memory*
- Kernel verwaltet benannte Speicherbereiche
- Existenz persistent, unabhängig von Prozessen
- Spezialfall von *Mapped Region*
- in C (Unix): `shm_open()`, `mmap()`, Bibliothek: `-lrt`
- in C++ (Boost): betriebssystemunabhängig
- ... außer, man verwendet MS-Windows-spezifischen Shared Memory, der beim Beenden des Prozesses automatisch entfernt wird

6 Die Boost-Bibliothek

6.4 Interprozeßkommunikation

- Verwalteter gemeinsamer Speicher: *Managed Shared Memory*
- Ein C++-map-Container verwaltet benannte Variable innerhalb des Shared Memory.
- erzeugen mit `construct<>`, verwenden mit `find<>`, beides mit `find_or_construct<>`
- wieder entfernen mit `destroy<>`, `destroy_ptr<>`
- Speicherüberlauf: `bad_alloc`-Exception

6 Die Boost-Bibliothek

6.4 Interprozeßkommunikation

- Verwalteter gemeinsamer Speicher: *Managed Shared Memory*
- Ein C++-`map`-Container verwaltet benannte Variable innerhalb des Shared Memory.
- erzeugen mit `construct<>`, verwenden mit `find<>`, beides mit `find_or_construct<>`
- wieder entfernen mit `destroy<>`, `destroy_ptr<>`
- Speicherüberlauf: `bad_alloc`-Exception
- spezielle Typen: `boost::interprocess::string` und Container (enthalten Zeiger → müssen auf Shared Memory verweisen)

6 Die Boost-Bibliothek

6.4 Interprozeßkommunikation

- Verwalteter gemeinsamer Speicher: *Managed Shared Memory*
- Ein C++-`map`-Container verwaltet benannte Variable innerhalb des Shared Memory.
- erzeugen mit `construct<>`, verwenden mit `find<>`, beides mit `find_or_construct<>`
- wieder entfernen mit `destroy<>`, `destroy_ptr<>`
- Speicherüberlauf: `bad_alloc`-Exception
- spezielle Typen: `boost::interprocess::string` und Container (enthalten Zeiger → müssen auf Shared Memory verweisen)
- Synchronisation: *Mutex*-Variable `interprocess_mutex`, `named_mutex`

6 Die Boost-Bibliothek

6.5 Asynchroner Input/Output

- serielle Schnittstelle, TCP/IP, ...
aber auch: Timer
- mit Callbacks
statt mit blockendem I/O und `select()`
- Timer: Zeitpunkt, Wartezeit
- TCP/IP: Resolver, Verbindungsaufbau, Daten
- statt Schleife: Callback re-installiert sich selbst

Vertiefung Software-Entwicklung in C++

<https://gitlab.cvh-server.de/pgerwinski/cpp.git>

- 1 Einführung
- 2 Wiederholung: Programmieren in C
- 3 Einführung in C++
- 4 Standard-Bibliotheken (STL)
- 5 C++11
 - ...
 - 5.4 R-Wert-Referenztypen
 - 5.5 Lambda-Ausdrücke
- 6 Die Boost-Bibliothek
 - 6.1 C++11-Erweiterungen für ältere Compiler
 - 6.2 Spracherweiterungen
 - 6.3 Zeitangaben
 - 6.4 Interprozeßkommunikation
 - 6.5 Asynchroner Input/Output
- 7 Plug-In-Architekturen



Änderungen
vorbehalten