

Programmieren C/C++



Prof. Dr. Dieter Nazareth



HOCHSCHULE LANDSHUT
HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN

© Copyright 2023 Prof. Dr. Dieter Nazareth
Programmieren

Organisatorisches Allgemeines

- Dozent: Prof. Dr. Dieter Nazareth
Raum: J2 15
E-Mail: dieter.nazareth@haw-landshut.de
Sprechstunde: Nach Vereinbarung
Unterlagen: Skript und Lernvideos im Moodle Kursraum
Praktikum

- Leistungsnachweis erforderlich!
- Es herrscht Teilnahmepflicht!
- Termine und Ablauf: siehe Ankündigungen in Moodle

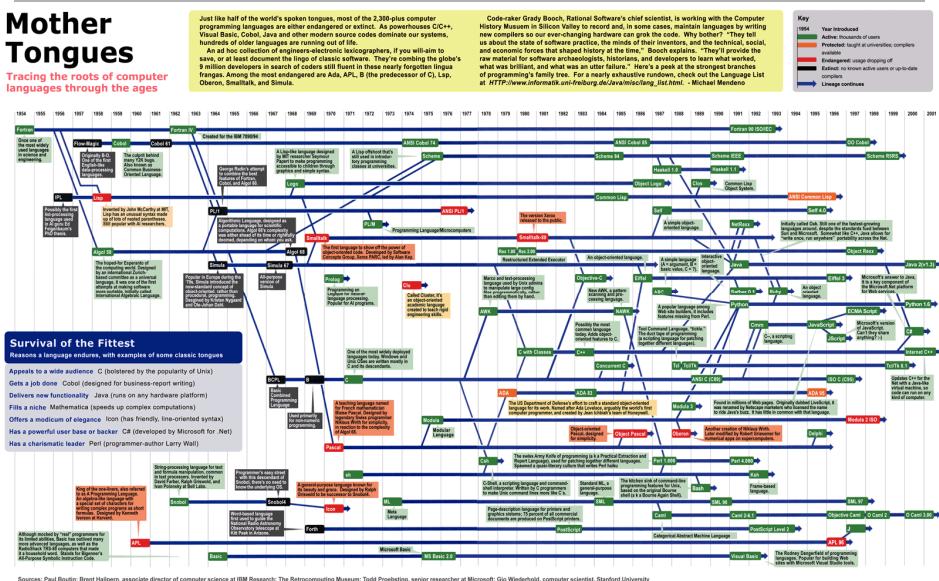
Organisatorisches Inhalt

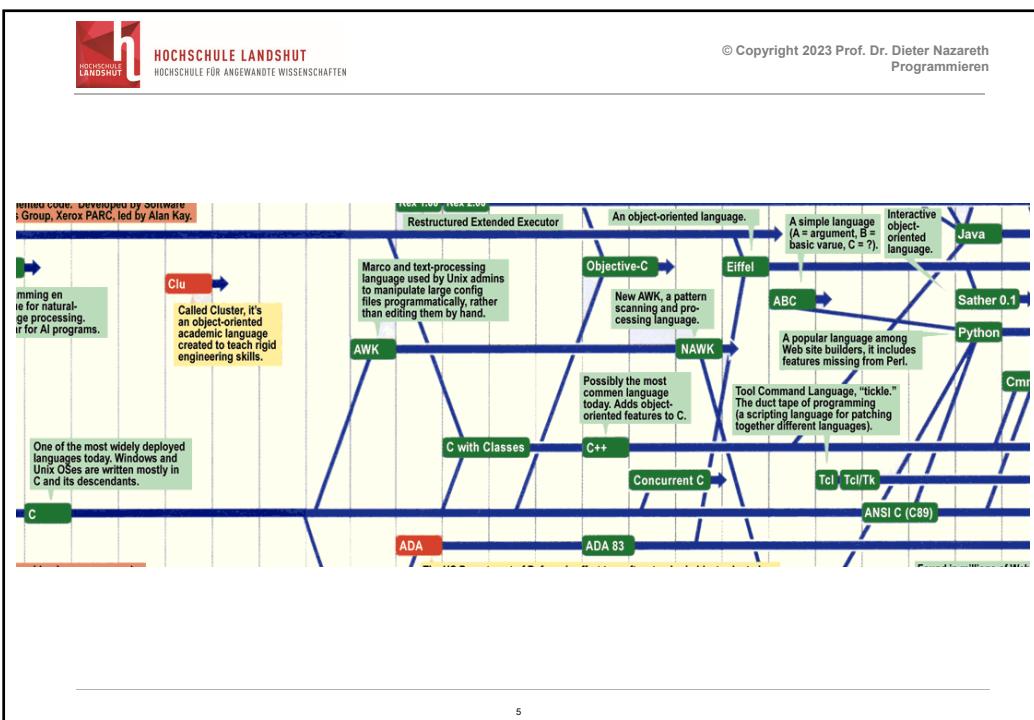
- **Lernziele**

- Sprachkonzepte der Programmiersprache C
 - Lösung von Problemstellungen in der Programmiersprache C
 - Kennenlernen von Softwareentwicklungsmethoden und Algorithmen
 - Elementare Konzepte von objektorientiertem Programmieren in C++

▪ Literatur

- Brian W. Kernighan, Dennis Ritchie: The C Programming Language
 - Jürgen Wolf: C von A bis Z: Das umfassende Handbuch, Galileo Computing
 - Jürgen Wolf: C++: Das umfassende Handbuch





HOCHSCHULE LANDSHUT
HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN

© Copyright 2023 Prof. Dr. Dieter Nazareth
Programmieren

Einführung

Die Programmiersprache C

- Eine der „klassischen“ Programmiersprachen
- Gehört zur Gruppe der imperativen Programmiersprachen
- Früher: **DIE** Standardprogrammiersprache (zusammen mit C++)
- Heute: Immer noch bevorzugte Sprache für systemnahe Programmierung wenn es auf hohe Effizienz ankommt, ansonsten ist Java derzeit dominant
- Warum lernen wir dann C?
 - Der imperative Kern vieler Sprachen ist C sehr nahe (Java z.B.)
 - Das alles kennen Sie dann schon wenn Sie Java lernen
 - Sie lernen grundlegende imperative Konzepte
 - Sie verstehen besser wie ein Computer funktioniert, wenn Sie ihn maschinennah programmieren

6

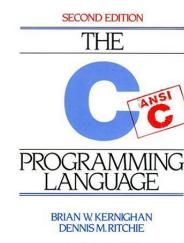
Einführung Ursprung von C

- Das Betriebssystem UNIX wurde 1969 von Bell Labs (AT&T) in Assembler auf einer PDP-7 implementiert.
- UNIX sollte auf andere Rechner portiert werden.
- Gesucht war eine Programmiersprache von der Art eines „Super-Assemblers“ mit folgenden Merkmalen:
 - Möglichkeiten einer hardwarenahen Programmierung vergleichbar mit Assembler
 - Performance des Laufzeitcodes vergleichbar mit Assembler
 - Unterstützung der Sprachmittel der **strukturierten Programmierung**.
- Ken Thompson entwickelte, beeinflusst von der Programmiersprache BCPL, die Sprache B.
- B war eine interpretative Sprache ohne Datentypen.

7

Einführung Ursprung von C

- Dennis Ritchie entwickelte 1971/72 die Sprache C, die diese Nachteile beseitigte.
- 1973 wurde UNIX neu in C realisiert (ca. 10% Assembleranteil)
- Bis 1978 wurde C durch Kernighan und Ritchie weiterentwickelt.
- 1978 erschien die C-Bibel: „The C Programming Language“



8

Einführung Standardisierung

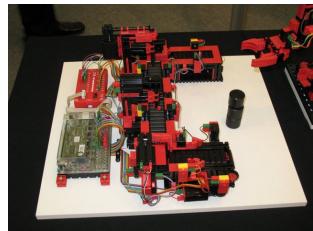
- Im Laufe der Jahre entwickelten sich viele unterschiedliche C Dialekte.
- 1989 wurde die Programmiersprache durch das *American National Standards Institute (ANSI)* standardisiert (C89).
- Ein Jahr später wurde diese Norm mit ein paar Änderungen von der *ISO* übernommen (C90).
- Bis heute ist C90 die Sprachbasis für alle Derivate der Programmiersprache C, unter anderem auch für das modernere C++.
- Ein auf C90 basierendes Programm **sollte** ohne Probleme von jedem C-Compiler übersetzt und ausgeführt werden können.
- Weitere Standards (C95 und C99) werden nicht von allen Compilern unterstützt.
- Verwendung von C90 in der Vorlesung.

Einführung Eigenschaften von C

- C ist eine relativ maschinennahe Sprache.
- C erlaubt über Adressen direkt auf den Speicher zuzugreifen.
- C erlaubt mit Adressen zu rechnen (Zeigerarithmetik).
- C erlaubt Operationen auf der Bitebene.
- C enthält die Elemente der strukturierten Programmierung (imperative Programmiersprache).
- C hat ein Typkonzept das allerdings nicht sehr streng ist.
- Die Sprache C unterstützt eine getrennte Kompilierbarkeit von Programmeinheiten.
- Es ist möglich, dass der in C geschriebene Quellcode eines Programms aus mehreren Dateien bestehen kann.
- Der Übersetzungsvorgang ist mehrstufig (z.B. Präprozessor).

Einführung C versus Java

- „C ist ein offener Geländewagen. Kommt durch jeden Matsch und Schlamm, aber der Fahrer sieht hinterher auch entsprechend aus.“
- „Java ist ein neues experimentelles Fahrzeug auf Luftkissenbasis. Es bewegt sich auf Straßen aller Art, ist allerdings noch schwer zu steuern. Es ist strengstens verboten Umbauten am Fahrzeug vorzunehmen.“



C



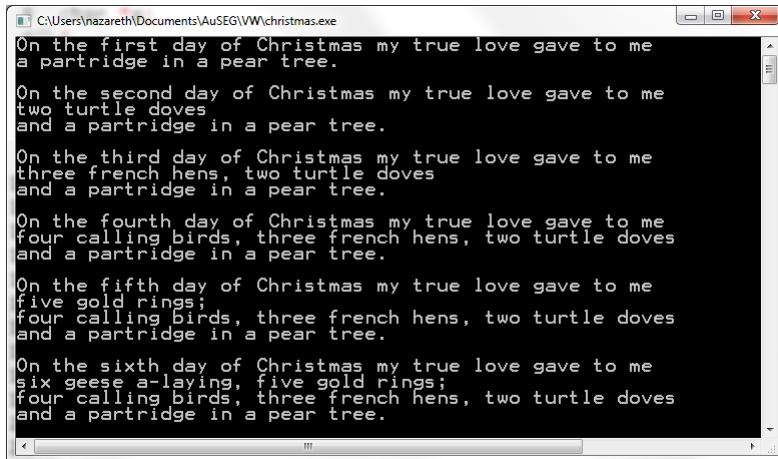
Java

Grundbegriffe Das erste Programm

```
#include <stdio.h>
main(t,_,a)
char *a;
{
    return!0<t?t<3?main(-79,-13,a+main(-87,1-,main(-86,0,a+1)+a)):
    1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
    main(2,_,+"%s %d %d\n"):9:16:t<0?t<-72?main(_,_,
    "@n'+,#'/*{w+/w#cdnr/+,{}r/*de}+,*{*,/w{%,/w#q#n+,/#{1+,/n{n+,/+#n+,/#\
    ;#q#n+,/+k#;*+,/'r : 'd*'3,{w+K w'K:'+}e#';dq#'\l \
    q#'+d'K#/!+k#,q#'\r}eKK#}w'r)eKK{n1}#/;#q#n'{})#w'){}){n1}'/#n';d}rw' i;# \
    )}{n1}]/n{n#'; r{#w'r nc{n1}#/l,+'K {rw' iK{};[{n1}'/w#q#n'wk nw' \
    iwk{KK{n1}!/w{%'l##w#' i;:{n1}'/*{q#'\ld;r'}{nlwb!/*de}'c \
    ;:{n1}'-{r}w'#+,}##'*#nc,',#nw]'/+kd'+e}+;#rdq#w! nr'/' ) }+}{r1#{'n' '})# \
    }'+}##(!/")
    :t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
    :0<t?main(2,2,"%s"):#a=='/'||main(0,main(-61,*a,
    "!ek;dc i@bK'(q)-[w]%"n+r3#1,{}:nuwloca-0;m .vpbks,fxntdCeghiry"),a+1);
}
```

Grundbegriffe

Das erste Programm



```
C:\Users\nazareth\Documents\AuSEG\VW\christmas.exe
On the first day of Christmas my true love gave to me
a partridge in a pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a pear tree.

On the third day of Christmas my true love gave to me
three french hens, two turtle doves
and a partridge in a pear tree.

On the fourth day of Christmas my true love gave to me
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

On the fifth day of Christmas my true love gave to me
five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.

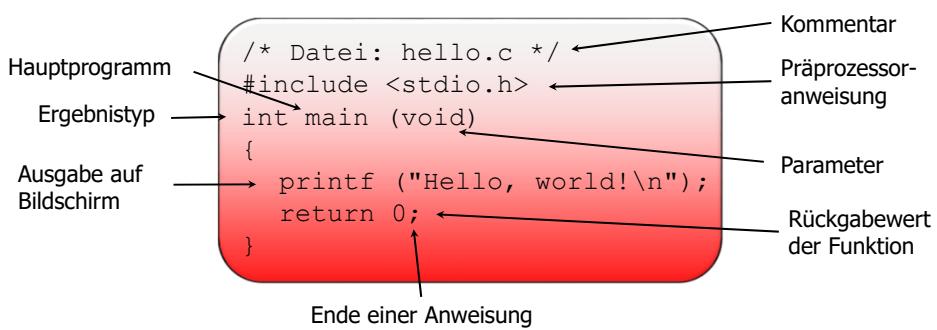
On the sixth day of Christmas my true love gave to me
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.
```

13

Grundbegriffe

Das erste Programm

- Seit dem Lehrbuch von Kernighan und Ritchie über C ist es Usus geworden, als erstes Beispiel in einer neuen Programmiersprache mit dem Hello-World-Programm zu beginnen.
- In C sieht das Hello-World-Programm folgendermaßen aus:



14

Grundbegriffe

Das erste Programm

Kommentare:

- Alles innerhalb der Begrenzer /* und */ ist ein Kommentar.
- Kommentare dienen dem menschlichen Verständnis und werden bei Übersetzen in Maschinensprache entfernt.
- Kommentare können auch über mehrere Zeilen gehen.

Präprozessoranweisung:

- Vor dem eigentlichen Übersetzen des Programms werden in einem vorgeschaltetem Prozess (Präprozess) die Präprozessoranweisungen ausgeführt.
- Alle Präprozessoranweisungen beginnen mit #.
- Die Anweisung #include <datei> bewirkt, dass der Inhalt der Datei datei an diese Stelle kopiert wird.
- #include <stdio.h> stdio.h enthält die Standard Ein- und Ausgabefunktionen wie z.B. printf

Grundbegriffe

Das erste Programm

Hauptprogramm:

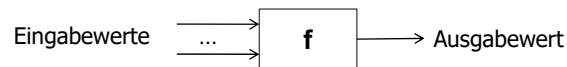
- Dieses Programm besteht aus einer Funktion main().
- In C muss jedes Programm eine Funktion dieses Namens haben.
- Wenn ein Programm ausgeführt werden soll, wird mit der Ausführung der main()-Funktion begonnen.
- Die main()-Funktion wird auch als Hauptprogramm bezeichnet.

Funktion

- Alle Programme in C basieren von ihrem Aufbau her komplett auf **Funktionen**.
- Die C Funktion entspricht einer mathematischen Funktion, die Eingabewerte bekommt und einen Ergebniswert berechnet und zurückgibt.

Grundbegriffe

Das erste Programm



Funktion (cont.):

- Das Schlüsselwort `void` signalisiert, dass die Funktion main keine Eingabeparameter besitzt.
- Das vorangestellte Schlüsselwort `int` definiert den Rückgabetyp.
- Damit muss die Funktion als Rückgabewert einen Integerwert zurückgeben.
- Die geschweiften Klammern `{ }` begrenzen den Rumpf der Funktion.
- Der Rumpf der Funktion besteht aus einer Sequenz von Anweisungen (Befehlen) und dient zu Berechnung des Ergebniswertes.
- Die Anweisungen werden von oben nach unten abgearbeitet.
- Jede Anweisung wird durch ein Semikolon ; beendet.
- Durch das Schlüsselwort `return` wird die Abarbeitung der Funktion beendet und ein Wert als Ergebnis zurückgegeben.

Grundbegriffe

Das erste Programm

Ein-/Ausgabe:

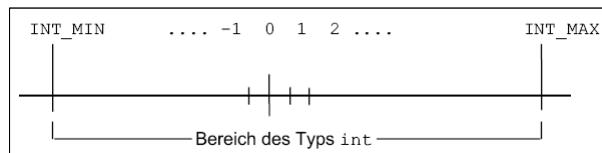
- Die Sprache C kennt keine eingebauten Befehle zur Ein- und Ausgabe von Werten.
- Ein- und Ausgabe erfolgt grundsätzlich über Bibliotheksfunktionen (vordefinierte mitgelieferte Standardfunktionen).
- Die Bibliotheksfunktion `printf()` dient zur Ausgabe auf dem Bildschirm („print formatted“).
- Durch die Präprozessoranweisung `#include <stdio.h>` wird diese Funktion dem Programm bekanntgemacht.
- Durch die Funktion schreibt die Zeichenkette `Hello, World` auf den Bildschirm.
- Durch `\n` wird eine neue Zeile begonnen (newline).

Grundbegriffe

Das erste Programm

Datentyp int:

- Der Datentyp `int` vertritt in C-Programmen die ganzen Zahlen (Integer-Zahlen).
- Es gibt in C jedoch noch weitere Integer-Datentypen. Sie unterscheiden sich vom Datentyp `int` durch ihre Repräsentation und damit auch durch ihren Zahlenbereich.
- Die `int`-Zahlen umfassen auf dem Computer einen endlichen Zahlenbereich, der nicht überschritten werden kann.



19

Grundbegriffe

Das erste Programm

Datentyp int:

- `INT_MIN` und `INT_MAX` sind die Grenzen der `int`-Werte auf einem Rechner. Somit gilt für jede beliebige Zahl x vom Typ `int`:
 x ist eine ganze Zahl, $\text{INT_MIN} \leq x \leq \text{INT_MAX}$
- Umfasst die interne Darstellung von `int`-Zahlen 32 Bit, so entspricht dies einem Zahlenbereich von -2^{31} bis $+2^{31} - 1$.
- Wird eine `int`-Zahl durch 16 Bit dargestellt, so wird ein Wertebereich von -2^{15} bis $+2^{15} - 1$ aufgespannt.
- Die Entscheidung, wie viele Bits letztendlich für die Darstellung von `int`-Zahlen genommen werden, hängt vom Compilerhersteller und vom Prozessor ab, auf dem das C-Programm ablaufen soll.
- `INT_MIN` und `INT_MAX` sind definiert in `<limits.h>`

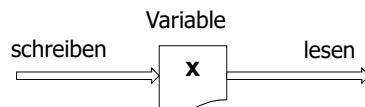
20

Grundbegriffe

Das zweite Programm

Variablen:

- Variablen sind Bezeichner für Speicherzellen.
- Sie dienen der Speicherung von Daten.
- Auf Variablen kann lesend und schreibend zugegriffen werden.
- Eine Variable entspricht einem Zettel mit einem Namen, der beschrieben und wieder gelöscht werden kann.



- Variablen müssen in C vor ihrer Verwendung definiert werden.
- Der Wert einer Variable kann sich während der Programmausführung ändern (wenn sie geschrieben wird).

Grundbegriffe

Das zweite Programm

Typen:

- C ist eine getypte Sprache.
- Typen sind ein Ordnungsprinzip für Daten das festlegt
 - welche Werte möglich sind
 - welche Operationen möglich sind
 - Beispiel: int sind ganze Zahlen mit Ganzzahlarithmetik
- In C müssen alle Variablen einem bestimmten Typ zugeordnet werden.
- Damit kann die Variable nur Daten von diesem Typ aufnehmen.
- Definition einer Variable:

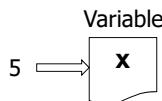
```
int x;
```
- Durch die Definition hat die Variable noch keinen konkreten Wert.
- Sie muss durch eine Zuweisung initialisiert werden.

Grundbegriffe

Das zweite Programm

Zuweisung:

- Eine Zuweisung an eine Variable gibt dieser einen Wert.
- Dies entspricht einem schreibendem Zugriff auf die Variable.
- Als Zuweisungszeichen dient in C das `=`, Bsp.:
 $x = 5;$
- Die Zuweisung erfolgt von rechts nach links. Der Wert der rechten Seite des `=` wird der links stehenden Variable zugewiesen.



- Die erste Zuweisung an eine Variable wird *Initialisierung* genannt.
- Wichtig! `=` bedeutet Zuweisung und hat nichts mit dem aus der Mathematik bekannten „ist gleich“ zu tun

Grundbegriffe

Das zweite Programm

Ausdruck:

- Ein Ausdruck ist ein mathematischer Term, dessen Berechnung einen Wert ergibt.
- In einem Ausdruck werden z.B. Zahlen und Variablen mit Operatoren verknüpft, Bsp.:

$5+4$
 $x+4$
 $x+3*y$

- Auf der rechten Seite einer Zuweisung kann ein Ausdruck stehen.
- Der Wert des Ausdrucks wird der Variable zugewiesen, z.B.:

$x = 5+4;$
 $y = x+4;$
 $y = x+3*y;$

Grundbegriffe

Das zweite Programm

Beispiel:

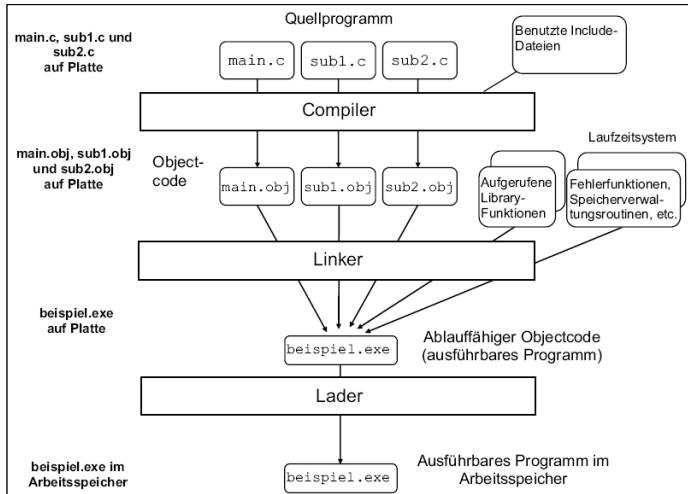
```
/* Datei: simple.c */
#include <stdio.h>
int main (void){
    int x;
    int y;
    x = 5+4;
    y = x+4;
    y = x+3*y;
    printf ("Die Variable y hat den Wert: %d",y);
    return 0;
}
```

Grundbegriffe

Programmerzeugung und -ausführung

- Der Quelltext eines Programms wird mit einem Editor geschrieben und auf der Festplatte des Rechners unter einem Dateinamen mit der Endung `.c` als Datei abgespeichert.
- Einfache Programme bestehen aus einer einzigen Quelldatei, komplexe aus mehreren Quelldateien.
- Das folgende Bild zeigt den Ablauf der Tätigkeiten, die nun durchzuführen sind, um ein C-Programm zu erzeugen und es auszuführen.
- Das Beispielprogramm besteht aus den 3 Quellcode-Dateien `main.c`, `sub1.c` und `sub2.c`.

Grundbegriffe Programmerzeugung und -ausführung



27

Grundbegriffe Programmerzeugung und -ausführung

Compiler:

- Die Aufgabe eines Compilers (Übersetzers) für die Programmiersprache C ist, den Text (Quellcode) eines C-Programms in Maschinencode (Objektcode) zu wandeln.
- Maschinencode ist eine prozessorspezifische Programmiersprache, die ein spezieller Prozessor direkt versteht.

Linker:

- Die Aufgabe eines Linkers ist es, die einzelnen Teile eines Maschinencodes zu einem ablauffähigem Programm zusammenzufügen.
- Zu den Objektdateien werden vom Linker noch die benötigten Bibliotheksfunktionen und ein Laufzeitsystem hinzugefügt.

28

Grundbegriffe

Programmerzeugung und -ausführung

Lader:

- Mit dem Lader (Loader) wird das Programm in den Arbeitsspeicher des Computers geladen, wenn es gestartet wird.
- Unter Windows oder UNIX ist das Laden des Programms relativ einfach: In einer Shell wird man `hello.exe` gefolgt von `Return` ein. Dadurch wird das Programm geladen und es wird ausgeführt.
- Im Embedded Bereich muss das Programm erst mal mit Hilfe eines geeigneten Programms auf den Controller gebracht werden.

Integrierte Entwicklungsumgebung (IDE):

- Um die Programmierung komfortabler zu gestalten, gibt es sogenannte integrierte Entwicklungsumgebungen (z.B. Eclipse).
- Diese enthalten innerhalb eines Programmsystems Compiler, Linker, Lader, Debugger sowie einen Editor zur Eingabe der Programmtexte.

Datentypen und Variablen

Datentypen

Einfache Datentypen:

- Der Datentyp einer Variablen legt die erlaubten Operationen und möglichen Werte fest.
- Der Compiler legt – abhängig von der Hardware für die kompiliert wird – die Repräsentation und Größe fest
- Durch die Darstellung (Repräsentation) wird zum einen der Speicherbedarf festgelegt, d.h. durch wie viele Bits die Variable dargestellt wird, zum anderen aber auch der Wertebereich.
- Alle elementaren (einfachen) Datentypen sind *arithmetische Typen*. Sie umfassen:
 - Ganzzahltypen (Integertypen)
 - Gleitpunkttypen.

Datentypen und Variablen

Datentypen

Einfache Datentypen (cont.):

- Der Typ `void` ist der ungültige (leere) Typ. Er bezeichnet wird beispielsweise verwendet, wenn eine Funktion keinen Rückgabewert oder keinen Übergabeparameter hat.



31

Datentypen und Variablen

Datentypen

Integertypen:

- `char` mit dem Integertyp `char` (character) werden Zeichen aus dem Zeichensatz abgelegt. Zeichen werden intern als ganzzahlige Werte gespeichert.
- `int` üblicher Standardtyp für ganzzahlige Werte.
- `short int` für kleine ganzzahlige Werte.
- `long int` für große ganzzahlige Werte.
- Statt `short int` reicht es auch, `short` zu schreiben, und statt `long int` entsprechend `long`.
- Für die Integertypen stehen noch die Modifizierer `signed` (vorzeichenbehaftet) bzw. `unsigned` (nicht vorzeichenbehaftet, positiv) zur Verfügung.

32

Datentypen und Variablen

Datentypen

Integertypen:

Bit 7 6 5 4 3 2 1 0

1	0	1	1	0	1	0	0
---	---	---	---	---	---	---	---

 Beispiel für eine Zahl vom Typ unsigned char
Stellenwert $2^7 2^6 2^5 2^4 2^3 2^2 2^1 2^0$

Der Wert der Zahl in Bild 5-1 berechnet sich zu:

$$1*2^7 + 0*2^6 + 1*2^5 + 1*2^4 + 0*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = 128 + 32 + 16 + 4 = 180$$

Achtung: Die Wahl des richtigen Datentyps ist wichtig. Wird ein zu kleiner Datentyp gewählt, dann kommt es zu einem Überlauf. Dieser wird in C nicht geprüft oder abgefangen!

Datentypen und Variablen

Datentypen

Integertypen:

- Die Größe einer Variablen vom Typ `char`, `unsigned char` oder `signed char` ist 1 Byte.
- Das Wort `char` ist die Abkürzung von `character` (Schriftzeichen).
- Gewöhnlich wird der Datentyp `char` dafür verwendet, um einzelne Zeichen aus dem Zeichensatz zu verarbeiten, wie z.B. '`c`', oder Steuerzeichen, wie '`\n`'.
- Der Wert eines Zeichens ist eine ganze Zahl entsprechend dem Zeichensatz auf der Maschine (ASCII).
- Der Datentyp `char` kann auch zur Darstellung bzw. zur Verarbeitung von ganzen Zahlen mit einem kleinen Wertebereich verwendet werden.
- Die Interpretation, ob Zahl oder Zeichen, hat durch den Anwender zu erfolgen.

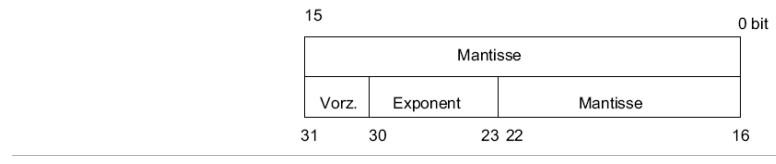
Datentypen und Variablen

Datentypen

Fließkommatypen:

- float Gleitpunktzahl (Fließkommazahl)
 - double Gleitpunktzahl mit einer höheren Genauigkeit der Darstellung
 - long double Gleitpunktzahl mit einer noch höheren Genauigkeit
- Fließkommazahlen werden auf dem Rechner als Exponentialzahlen in der Form Mantisse * Basis ^{Exponent} dargestellt.

float: 1 Vorzeichenbit (Bit 31)
8 Bits für Exponenten (Bit 23 - 30)
23 Bits für Mantisse (Bit 0 - 22)



35

Datentypen und Variablen

Datentypen

Datentyp	Anzahl Bytes	Wertebereich (dezimal)
char	1	-128 bis +127 oder 0 bis +255 (maschinenabhängig)
unsigned char	1	0 bis +255 (erweiterter ASCII-Zeichensatz)
signed char	1	-128 bis +127 (alle ASCII-Zeichen bis 127)
int	4 in der Regel	-2 147 483 648 bis +2 147 483 647
unsigned int	4 in der Regel	0 bis +4 294 967 295
short int	2 in der Regel	-32 768 bis +32 767
unsigned short int	2 in der Regel	0 bis +65 535
long int	4 in der Regel	-2 147 483 648 bis +2 147 483 647
unsigned long int	4 in der Regel	0 bis +4 294 967 295
float	4 in der Regel	$-3.4 \cdot 10^{38}$ bis $+3.4 \cdot 10^{38}$
double	8 in der Regel	$-1.7 \cdot 10^{308}$ bis $+1.7 \cdot 10^{308}$
long double	10 in der Regel	$-1.1 \cdot 10^{4932}$ bis $+1.1 \cdot 10^{4932}$

36

Datentypen und Variablen

Variablen

Deklaration und Definition:

- Eine *Deklaration* legt den Namen eines Objektes und seinen Typ fest (macht diesen Bezeichner bekannt). Damit weiß der Compiler, mit welchem Typ er einen Namen verbinden muss.
- Eine *Definition* von Variablen dient dazu, Variablen im Speicher anzulegen, d.h. Platz im Speicher zu reservieren. Hierbei ist automatisch eine Deklaration mit eingeschlossen.
- Definition = Deklaration + Reservierung des Speicherplatzes.
- Separate Deklarationen werden bei größeren Projekten benötigt, wenn ein Projekt aus mehreren Dateien besteht (siehe später).

Datentypen und Variablen

Variablen

Definition einer Variable:

Eine einzige Variable wird definiert durch eine Vereinbarung der Form

```
datentyp name;
```

also beispielsweise durch

```
int x;
```

Vom selben Typ können mehrere Variablen in einer einzigen Vereinbarung definiert werden, indem man die Variablennamen durch Kommata trennt:

```
int x, y, z;
```

- Die Namen der Variablen müssen die Namenskonventionen einhalten.
- Ein Variablenname darf nicht identisch mit einem Schlüsselwort sein.
- Jede Variable in einer Folge von Definitionen von Variablen muss ihren eigenen, eindeutigen Namen erhalten.

Datentypen und Variablen

Variablen

Namenskonventionen:

- Ein Name besteht aus einer Zeichenfolge aus Buchstaben und Ziffern, die mit einem Buchstaben beginnt. In C zählt auch der Unterstrich _ zu den Buchstaben.
- Nach ANSI-C sind mindestens 31 Zeichen für interne Namen und mindestens 6 Zeichen für externe Namen relevant. Das heißt, dass ein Compiler in der Lage sein muss, mindestens so viele Zeichen eines Namens bewerten zu können.
- Beispiele für zulässige Namen sind:
`summe, x_quadrat`
- Beispiele für unzulässige Namen sind:
`1x (beginnt mit Ziffer), x-quadrat (Sonderzeichen -),
ärger (kein zulässiges Zeichen: ä)`

Datentypen und Variablen

Variablen

Initialisierung:

- Lokale Variablen werden **nicht automatisch initialisiert**.
- Der Wert einer Variable nach der Definition ist **undefiniert** (der Wert, der zufällig an der Speicherstelle steht, an der die Variable abgelegt wird).
- Jede einfache Variable kann bei ihrer Definition initialisiert werden, indem man einfach den Zuweisungsoperator = gefolgt von einer Konstanten des passenden Typs an den Namen der Variablen anhängt, z.B.:
`int a=9;
int b=1, c=2;`
- **Achtung:** Der Compiler überprüft nicht, ob eine Variable initialisiert worden ist!

Datentypen und Variablen

Konstanten

In C gibt es zwei Arten von Konstanten:

- *literale Konstanten*
- *symbolische Konstanten*.

Literele Konstanten:

- Literale haben keinen Namen. Sie werden durch ihren Wert dargestellt.
- Ganzzahlige Konstanten: 0 1 2 ... 2345 ... -1 -2 ... -328
- Gleitkommakonstanten: 1.0 300.0 3e2 3.E2 .3E3
- Eine Zeichenkonstante ist ein Zeichen eingeschlossen in einfachen Hochkommas: 'a' 'b' '\n'
- Eine Zeichenkettenkonstante (string) ist eine Zeichenreihe, die in Anführungszeichen eingeschlossen wird: „Hello, World\n“

Datentypen und Variablen

Konstanten

Symbolische Konstanten:

- Symbolische Konstanten (benannte Konstanten) haben einen Namen, der ihren Wert repräsentiert.
- In C gibt es 2 Möglichkeiten symbolische Konstanten zu definieren.

Präprozessorbefehl #define:

- Symbolische Konstanten können mit dem Präprozessor-Befehl #define eingeführt werden:

```
#define PI 3.1415
```

- Damit wird eine symbolische Konstante mit dem Namen PI eingeführt, die als Wert die literale Konstante 3.1415 hat.
- Garantiert konsistente Mehrfachverwendung und Austauschbarkeit.

Datentypen und Variablen

Konstanten

Konstante Variable:

- Mit dem Schlüsselwort `const` ist es möglich, eine Variable anzulegen, die nur einmal initialisiert werden kann und anschließend vom Compiler vor Schreibzugriffen geschützt wird, z.B.:

```
const double PI = 3.1415927;
```

- Garantiert konsistente Mehrfachverwendung und Austauschbarkeit.
- Höhere Typsicherheit verglichen mit `#define`.

Ausdrücke und Operatoren

Ausdruck

Definition:

- Ein Ausdruck ist ein mathematischer Term, dessen Berechnung einen Wert ergibt.
- Ein Ausdruck ist in C im einfachsten Fall der Bezeichner (Name) einer Variablen oder einer Konstante.
- Durch Verknüpfung von Variablen und Konstanten mit Operatoren und Funktionen entsteht ein komplexer Ausdruck.
- Ausdrücke können durch Klammern strukturiert werden.
- Jeder Ausdruck hat einen Wert, der durch die Ausführung des Ausdrucks berechnet wird.
- Auf der rechten Seite einer Zuweisung muss immer ein Ausdruck stehen.

Ausdrücke und Operatoren

Operatoren:

Als Operatoren werden die in C fest eingebauten Funktionen bezeichnet:

```
( )   [ ]   ->   .
!     ~     ++   --   +     -     *     &     (Typname)   sizeof
/     %     <<   >>   <     <=   >     >=   ==   !=   ^     |     &&
||    ?:    =     +=   -=   *=   /=   %=   &=   ^=   |=   <<=   >>=
```

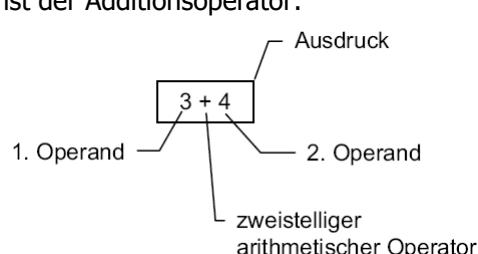
Es gibt in C die folgenden Klassen von Operatoren:

- einstellige (unäre, monadische)
- zweistellige (binäre, dyadische)
- und einen einzigen dreistelligen (ternären, tryadischen)

Ausdrücke und Operatoren

Operatoren:

Benötigt ein Operator 2 Operanden für die Verknüpfung, so spricht man von einem zweistelligen (binären) Operator. Ein vertrautes Beispiel für einen binären Operator ist der Additionsoperator:



Bsp.: int a=9;
printf("%d", a+3);

Ausdrücke und Operatoren

Operatoren

Auswertungsreihenfolge:

Wie in der Mathematik spielt es bei C eine wichtige Rolle, in welcher Reihenfolge ein Ausdruck berechnet wird. Genau wie in der Mathematik gilt auch in C die Regel "Punkt vor Strich", weshalb $5 + 2 * 3$ gleich 11 und nicht 21 ist. Allerdings gibt es in C sehr viele Operatoren. Durch eine Tabelle ist die Priorität der einzelnen Operatoren festgelegt.

Assoziativität:

Unter Assoziativität versteht man die Reihenfolge, wie Operatoren und Operanden verknüpft werden, wenn mehrstellige Operatoren der gleichen Priorität (Vorrangstufe) über ihre Operanden miteinander verkettet sind.

Ist ein Operator in C *rechtsassoziativ*, so wird eine Verkettung von Operatoren dieser Art von rechts nach links abgearbeitet, bei *Linksassoziativität* dementsprechend von links nach rechts.

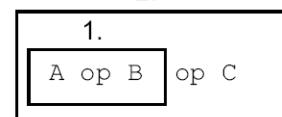
Ausdrücke und Operatoren

Operatoren

Assoziativität (cont.):

Unter Assoziativität versteht man die Reihenfolge, wie Operatoren und Operanden verknüpft werden, wenn mehrstellige Operatoren der gleichen Priorität (Vorrangstufe) über ihre Operanden miteinander verkettet sind.

Ist ein Operator in C *rechtsassoziativ*, so wird eine Verkettung von Operatoren dieser Art von rechts nach links abgearbeitet, bei *Linksassoziativität* dementsprechend von links nach rechts.



Da Additions- und Subtraktionsoperator linksassoziativ sind und dieselbe Priorität haben, wird beispielsweise der Ausdruck $A - B + C$ wie $(A - B) + C$ verknüpft und nicht wie $A - (B + C)$.

Ausdrücke und Operatoren

Operatoren

Vorrangtabelle

Typ	Prio	Operatoren	Assoziativität
Einstellig	15	[] . -> ()	links → rechts
	14	! ~ ++ -- & * (TYP) sizeof + -	rechts → links
Arithmetisch	13	* / %	links → rechts
	12	+ -	links → rechts
Bitshift	11	<< >>	links → rechts
Vergleich	10	< <= > >=	links → rechts
	9	== !=	links → rechts
Bitweise	8	&	links → rechts
	7	^	links → rechts
	6		links → rechts
Logisch	5	&&	links → rechts
	4		links → rechts
Bedingter Ausdruck	3	? :	rechts → links
Zuweisung	2	= += -= *= /= %= &= ^= = <=>=	rechts → links
Sequenz	1	,	links → rechts

Ausdrücke und Operatoren

Operatoren

Auswertungsreihenfolge:

1. Wie in der Mathematik werden als erstes Teilausdrücke in Klammern ausgewertet.
2. Dann werden Ausdrücke mit unären Operatoren ausgewertet. Unäre Operatoren werden von rechts nach links angewendet. Dies bedeutet, dass
 - a. zuerst die Postfix-Operatoren auf ihre Operanden
 - b. und dann die Präfix-Operatoren auf ihre Operanden angewendet werden.
3. Abschließend werden Teilausdrücke mit mehrstelligen Operatoren ausgewertet.

Bsp.: `int a = 9;
printf("%d", -a++);`

Ausdrücke und Operatoren

Operatoren

Einstellige Operatoren:

Ein *einstelliger (unärer) Operator* hat einen einzigen Operanden wie z.B. der Minusoperator als Vorzeichenoperator. So ist in -3 das $-$ ein Vorzeichenoperator, der auf die positive Konstante 3 angewandt wird.

Bei einstelligen Operatoren unterscheidet man zwischen *Präfixoperatoren* und *Postfixoperatoren*.

Präfixoperator: Der Operator steht vor dem Operanden

Bsp.: `int a=9;
printf("%d", -a);`

Postfixoperator: Der Operator steht hinter dem Operanden

Bsp.: Inkrementoperator `++`
`int a=9;
a++;
printf("%d", a);`

51

Ausdrücke und Operatoren

Operatoren

Zweistellige arithmetische Operatoren:

- Additionsoperator: $A+B$
- Subtraktionsoperator: $A-B$
- Multiplikationsoperator: $A*B$
- Divisionsoperator: A/B
- Restwertoperator: $A \% B$

Achtung: Die arithmetischen Operatoren sind alle *überladen*, d.h. sie stehen für alle arithmetischen Datentypen zur Verfügung (Ausnahme: Restwertoperator). Ihr Verhalten ist dabei auf unterschiedlichen Datentypen anders!

52

Ausdrücke und Operatoren

Operatoren

Divisionsoperator:

Bei der Verwendung des Divisionsoperators mit ganzzahligen Operanden ist das Ergebnis wieder eine ganze Zahl, d.h. es wird eine Ganzzahldivision durchgeführt. Der Nachkommanteil des Ergebnisses wird abgeschnitten. Ist mindestens ein Operand eine Fließkommazahl (z.B. float), so ist das Ergebnis eine Fließpunktzahl, d.h. es wird die übliche mathematische Division durchgeführt.

Beispiele:

5 / 5

Ergebnis: 1

5 / 3

Ergebnis: 1

5 / 0

dieser Ausdruck ist nicht zulässig, liefert undefinierten Wert

11.0 / 5

Ergebnis: 2.2

Ausdrücke und Operatoren

Operatoren

Restwertoperator:

Der Restwertoperator gibt den Rest bei der ganzzahligen Division des Operanden A durch den Operanden B an. Er ist nur für ganzzahlige Operanden (z.B. int) anwendbar!

Beispiele:

5 % 3

Ergebnis: 2

10 % 5

Ergebnis: 0

2 % 0

dieser Ausdruck ist nicht zulässig, liefert undefinierten Wert

Ausdrücke und Operatoren

Operatoren

Einstellige arithmetische Operatoren:

Neben den zweistelligen Operatoren gibt es in C auch noch folgende einstellige Operatoren:

- positiver Vorzeichenoperator: +A
- negativer Vorzeichenoperator: -A
- Postfix-Inkrementoperator: A++
- Präfix-Inkrementoperator: ++A
- Postfix-Dekrementoperator: A--
- Präfix-Dekrementoperator: --A

Ausdrücke und Operatoren

Operatoren

Postfix-Inkrementoperator: A++

Der **Rückgabewert** des Ausdrucks ist der unveränderte Wert des Operanden. Als **Nebeneffekt** wird der Wert des Operanden um 1 inkrementiert.

Prefix-Inkrementoperator: ++A

Der **Rückgabewert** des Ausdrucks ist der um 1 inkrementierte Wert des Operanden. Als **Nebeneffekt** wird der Wert des Operanden um 1 inkrementiert.

Beispiele:

```
var x = 5;  
printf(„%d“, x++) ;      Ausgabe: 5  
printf(„%d“, ++x) ;     Ausgabe: 7
```

Ausdrücke und Operatoren

Operatoren

Zuweisungsoperatoren:

Zu den Zuweisungsoperatoren gehören der

- einfache Zuweisungsoperator: $a = b$
- Additions-Zuweisungsoperator: $a += b$
- Subtraktions-Zuweisungsoperator: $a -= b$
- Multiplikations-Zuweisungsoperator: $a *= b$
- Divisions-Zuweisungsoperator: $a /= b$
- Restwert-Zuweisungsoperator: $a \% = b$

Sowie weitere logische Zuweisungsoperatoren (siehe später).

Ausdrücke und Operatoren

Operatoren

Einfacher Zuweisungsoperator:

Der Zuweisungsoperator wird in C als binärer, rechtsassoziativer Operator betrachtet und liefert als Rückgabewert den Wert des rechten Operanden. Es handelt sich bei einer Zuweisung also um einen **Ausdruck**. Dieses Konzept ist typisch für C. In anderen Sprachen (z.B. Pascal oder in GDI) ist eine Zuweisung kein Ausdruck, sondern eine Anweisung. In C können Zuweisungen wiederum in Ausdrücken weiter verwendet werden.

Beispiel Mehrfachzuweisung:

```
int a, b;  
a = b = 5; /* entspricht a = (b = 5) */  
printf(“%d”, a);
```

Ausdrücke und Operatoren

Operatoren

Additions-Zuweisungsoperator:

Der Additions-Zuweisungsoperator ein zusammengesetzter Operator. Zum einen werden die beiden Werte addiert. Zum anderen erhält die Variable auf der linken Seite als Nebeneffekt den Wert dieser Addition zugewiesen. Damit entspricht der Ausdruck

$$A += B$$

semantisch genau dem Ausdruck

$$A = A + (B).$$

Außer der kurzen Schreibweise kann der Additions-Zuweisungsoperator gegenüber der konventionellen Schreibweise noch einen Vorteil für den Compiler bringen.

Ausdrücke und Operatoren

Operatoren

Additions-Zuweisungsoperator (cont):

Beispiel:

```
int a=1, b=2;  
a+=b;           /* entspricht a=a+b) */  
printf(„%d“, a); /* Ergebnis: 3 */
```

Für die sonstigen kombinierten Zuweisungsoperatoren gilt das Gleiche wie bei dem Additions-Zuweisungsoperator. Außer der konventionellen Schreibweise:

$$A = A \text{ op } (B)$$

gibt es die zusammengesetzte kurze Schreibweise:

$$A \text{ op=} B$$

Ausdrücke und Operatoren

Operatoren

Relationale Operatoren

In C gibt es folgende zweistellige relationale Operatoren (Vergleichsoperatoren)

- Gleichheitsoperator: ==
- Ungleichheitsoperator: !=
- Größeroperator: >
- Kleineroperator: <
- Größergleichoperator: >=
- Kleinergleichoperator: <=

Der Rückgabewert von Vergleichsoperationen ist immer vom Datentyp int. Wenn ein Vergleich falsch ist, ist der Rückgabewert 0, wenn er wahr ist, ist der Rückgabewert 1.

Ausdrücke und Operatoren

Operatoren

Gleichheitsoperator: A == B

Mit dem Gleichheitsoperator wird überprüft, ob der Wert des linken Operanden mit dem Wert des rechten Operanden übereinstimmt. Ist das der Fall – sprich, ist der Vergleich wahr – hat der Rückgabewert den Wert 1. Andernfalls, d.h. wenn der Vergleich falsch ist, hat der Rückgabewert den Wert 0.

Beispiel:

```
printf ("Der Wert des Ausdruckes 1 + 2 == 3 ist: %d",
       1 + 2 == 3);
```

Ein folgeschwerer Fehler ist in C, statt des Gleichheitsoperators == versehentlich den Zuweisungsoperator = anzuschreiben. Ein solches Programm ist oft kompilier- und lauffähig, erzeugt aber andere Ergebnisse als erwartet.

Ausdrücke und Operatoren

Operatoren

Wahrheitstafel für logisches UND:

A	B	A&B
falsch	falsch	falsch
falsch	wahr	falsch
wahr	falsch	falsch
wahr	wahr	wahr

Beispiele:

```
0 && 1          /* Ergebnis: 0 (falsch) */  
5 && 6          /* Ergebnis: 1 (wahr) */  
3 < 5 && 5 > 3  /* Ergebnis: 1 (wahr) */
```

Ausdrücke und Operatoren

Operatoren

Wahrheitstafel für logisches ODER:

A	B	A B
falsch	falsch	falsch
falsch	wahr	wahr
wahr	falsch	wahr
wahr	wahr	wahr

Beispiele:

```
0 || 1          /* Ergebnis: 1 (wahr) */  
0 || (1 && 0)    /* Ergebnis: 0 (falsch) */  
'b' == 'a' + 1 || 0  /* Ergebnis: 1 (wahr) */
```

Ausdrücke und Operatoren

Operatoren

Wahrheitstafel für logischen Negationsoperator:

A	!A
wahr	falsch
falsch	wahr

Beispiele:

```
!0      /* Ergebnis: 1 (wahr) */  
!!5     /* Ergebnis: 1 (wahr) */  
!0 == 1 /* Ergebnis: 1 (wahr) */
```

Die Operatoren UND/ODER haben eine sehr geringe Priorität. Die Vergleichsoperatoren haben eine höhere Priorität als die logischen Operatoren. Deshalb sind Klammern für die Bewertung der Ausdrücke oft nicht notwendig. So entspricht `(a < b) && (c == d)` dem Ausdruck `a < b && c == d`.

Ausdrücke und Operatoren

Operatoren

Der Bedingungsoperator: A ? B : C

Der Bedingungsoperator ist der einzige Operator, der drei Operanden verarbeitet. In einem bedingten Ausdruck `A ? B : C` wird zuerst der Ausdruck A ausgewertet. Ist der Rückgabewert von Ausdruck A ungleich 0, also wahr, so wird der Ausdruck B ausgewertet. Das Ergebnis von B ist dann der Rückgabewert des Bedingungsoperators. Ist jedoch der Ausdruck A gleich 0, also falsch, so wird der Ausdruck C ausgewertet.

Beispiele:

```
1 == 1 ? 0 : 1      /* Rückgabewert: 0 */  
0 ? 0 : 1          /* Rückgabewert: 1 */  
printf ("Die Zahl %d ist %s\n", x,  
       x < 0 ? "negativ": "positiv" );
```

Kontrollstrukturen

Sequenz und Block

Die einfachste Kontrollstruktur ist die Sequenzbildung. In C können Anweisungen einfach hintereinander geschrieben werden. Im Gegensatz zu anderen Sprachen ist der ; keine Begrenzer zwischen zwei Anweisungen, sondern er begrenzt eine Anweisung. Durch die geschweiften Klammern werden die Anweisungen zu einem Block zusammengefasst und bilden damit wieder eine Anweisung.

```
{  
    Anweisung_1  
    Anweisung_2  
    ...  
    Anweisung_n  
}
```

Die Anweisungen werden dabei von oben nach unten abgearbeitet.

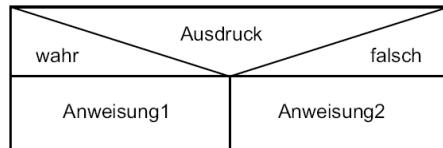
Kontrollstrukturen

Selektion

Bedingte Anweisung:

Die Syntax der einfachen Alternative ist:

```
if (Ausdruck)  
    Anweisung1  
else  
    Anweisung2
```



Der Ausdruck in Klammern wird berechnet. Trifft die Bedingung zu (hat also Ausdruck einen von 0 verschiedenen Wert), so wird Anweisung1 ausgeführt. Trifft die Bedingung nicht zu (hat also Ausdruck den Wert 0), so wird Anweisung2 ausgeführt. Der else-Zweig ist optional.

Kontrollstrukturen

Selektion

Bedingte Anweisung:

Beispiel:

```
if (a<b)
    printf(„Die größere Zahl ist %d“,b);
else
    if (b < a)
        printf(„Die größere Zahl ist %d“,a);
    else
        printf(„Beide Zahlen sind gleich groß: %d“,b);
```



Ein else bezieht sich immer auf das letzte if das noch ohne else ist! Empfehlung aus der Praxis: Immer { verwenden

69

Kontrollstrukturen

Selektion

Empfohlene Verwendung von {:

```
if (a<b) {
    printf(„Die größere Zahl ist %d“, b);
}
else {
    if (b < a) {
        printf(„Die größere Zahl ist %d“, a);
    }
    else {
        printf(„Beide Zahlen sind gleich groß: %d“,b);
    }
}
```

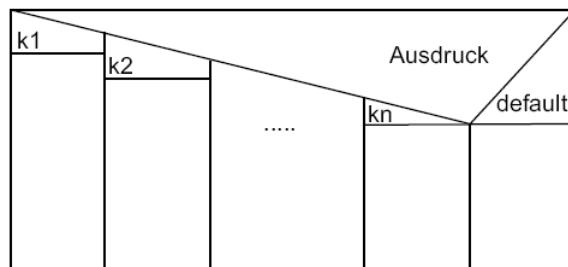
70

Kontrollstrukturen

Selektion

Mehrfachselektion:

Für eine Mehrfach-Selektion, d.h. eine Selektion unter mehreren Alternativen, steht die switch-Anweisung zur Verfügung. Die Alternativen müssen dabei konstante Ausdrücke von einem Integer-Typ sein.



71

Kontrollstrukturen

Selektion

Mehrfachselektion (cont.):

```
switch (Ausdruck) {
    case k1:
        Anweisungen_1
        break; /* ist optional */
    case k2:
        Anweisungen_2
        break; /* ist optional */
    ...
    case kn:
        Anweisungen_n
        break; /* ist optional */
    default: /* ist */
        Anweisungen_default /* optional */
}
```

72

Kontrollstrukturen

Selektion

Mehrfachselektion (cont.):

- Jeder Alternative geht eine – oder eine Reihe – von case-Marken mit **ganzahligen Konstanten** k₁, ..., k_n oder konstanten Ausdrücken voraus.
- Hat der konstante Ausdruck der switch-Anweisung den gleichen Wert wie einer der Ausdrücke der case-Marken, wird die Ausführung des Programms mit der Anweisung hinter dieser case-Marke weitergeführt. Stimmt keiner der Ausdrücke mit dem switch-Ausdruck überein, wird zu default gesprungen.
- Wird durch die switch-Anweisung eine passende case-Marke gefunden, werden die anschließenden Anweisungen bis zum break ausgeführt. break springt dann zu der auf die switch-Anweisung folgenden Anweisung.
- Fehlt die break-Anweisung, so werden die nach der nächsten case-Marke folgenden Anweisungen abgearbeitet. Dies geht so lange weiter, bis ein break gefunden wird oder bis das Ende der switch-Anweisung erreicht ist.

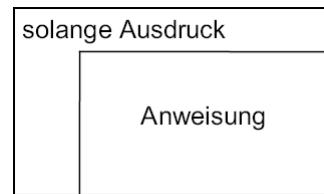
Kontrollstrukturen

Iteration

Abweisende Schleife mit while:

Die Syntax der while-Schleife lautet:

```
while (Ausdruck)
      Anweisung
```



In einer while-Schleife kann eine Anweisung in Abhängigkeit von der Bewertung eines Ausdrucks wiederholt ausgeführt werden. Da der Ausdruck vor der Ausführung der Anweisung bewertet wird, spricht man auch von einer **abweisenden Schleife**.

Der Ausdruck wird berechnet und die Anweisung nur dann ausgeführt, wenn der Ausdruck wahr ist. Danach wird die Berechnung des Ausdrucks und die eventuelle Ausführung der Anweisung wiederholt.

Kontrollstrukturen

Iteration

Beispiel:

```
unsigned erg = 1;
unsigned x = 5;
while (x > 1) {
    erg *= x;
    x--;
}
printf(„Fakultät von 5 ist %u“, erg);
while (1) { /* Endlosschleife */
    /*. . .*/
}
```

Kontrollstrukturen

Iteration

Gezählte Wiederholung:

Die for-Schleife ist wie die while-Schleife eine abweisende Schleife, da erst geprüft wird, ob die Bedingung für ihre Ausführung zutrifft.

Die Syntax der for-Schleife lautet:

```
for (Anweisung_1; Ausdruck_2; Anweisung_3)
    Anweisung
```

Die for-Anweisung ist äquivalent zu

```
Anweisung_1;
while (Ausdruck_2) {
    Anweisung
    Anweisung_3;
}
```

Kontrollstrukturen

Iteration

Gezählte Wiederholung (cont.):

Die for-Schleife enthält 3 Schritte:

- Initialisierung einer Laufvariablen, welche die Anzahl der Schleifendurchläufe zählt, in Ausdruck_1.
- Prüfung der Schleifenbedingung durch Ausdruck_2,
- gegebenenfalls Ausführung von Anweisung und Erhöhung des Wertes der Laufvariablen in Ausdruck_3, falls kein Abbruch erfolgte.

In einer gebräuchlichen Form wird die for-Schleife so verwendet, dass die Ausdrücke Ausdruck_1 und Ausdruck_3 Zuweisungen an die Laufvariable sind und Ausdruck_2 eine Bedingung über den Wert der Laufvariablen ist. In folgendem Beispiel wird dies veranschaulicht:

77

Kontrollstrukturen

Iteration

Gezählte Wiederholung (cont.):

- Typischerweise wird die for-Schleife als Zählschleife verwendet:

```
for (i = 0; i < 10; i++)  
    Anweisung
```
- Aber es sind auch ganz allgemeine Ausdrücke und sogar leere möglich

```
for (;;) {  
    Anweisung  
}
```

78

Kontrollstrukturen

Iteration

Gezählte Wiederholung (cont.):

Beispiel:

```
unsigned erg = 1;
unsigned i;
for (i = 2; i <= 5; i++) {
    erg *= i;
}
printf(„Fakultät von 5 ist %u“, erg);
```

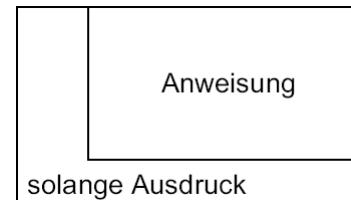
Kontrollstrukturen

Iteration

Annehmende Schleife mit do while:

Die Syntax der do while-Schleife ist:

```
do {
    Anweisung
} while (Ausdruck);
```



Die do while-Schleife ist eine „annehmende Schleife“. Zuerst wird die Anweisung der Schleife einmal ausgeführt. Danach wird der Ausdruck bewertet. Die Anweisung und die Bewertung des Ausdrucks werden solange fortgeführt, wie der Ausdruck wahr ist.

Im Gegensatz zu allen anderen Schleifen, wird bei der do while-Schleife ein Semikolon nach der Schleifenbedingung verlangt, weil hier das Ende der Anweisung erreicht ist.

Kontrollstrukturen

Iteration

Annehmende Schleife mit do while (cont.):

Beispiel:

```
int zahl;  
  
long summe = 0;  
  
do {  
    scanf ("%d", &zahl);  
    summe += zahl;  
} while (zahl); /* Hier muss ein Strichpunkt stehen ! */
```

Kontrollstrukturen

Iteration

Die **break** Anweisung kann verwendet werden um eine Schleife vorzeitig zu verlassen. Sie sollte immer mit Vorsicht verwendet werden, weil dasselbe Verhalten auch immer ohne break erreicht werden kann.

Beispiel:

```
int main(void) {  
    int i;  
    for (i = 0; i < 10; i++) {  
        /* do something */  
        if (cond)  
            break;  
    }  
}
```

Kontrollstrukturen

Iteration

Die **continue** Anweisung kann verwendet werden um vorzeitig mit der nächsten Schleifeniteration fortzufahren. Auch sie sollte aus dem selben Grund immer mit Vorsicht verwendet werden.

Beispiel:

```
int main(void) {  
    int i;  
    for (i = 0; i < 10; i++) {  
        /* do something */  
        if (cond)  
            continue;  
        /* do something */  
    }  
}
```

83

Funktionen

Funktionsdefinition

Syntax

Die Definition einer Funktion besteht in C aus dem **Funktionskopf** und dem **Funktionsrumpf**. Der Funktionskopf legt die Aufrufschnittstelle der Funktion fest. Der Funktionsrumpf enthält lokale Vereinbarungen und die Anweisungen der Funktion:

```
rueckgabetyp funktionsname (typ_1 formaler_param_1,  
                           typ_2 formaler_param_2,  
                           ... ,  
                           typ_n formaler_param_n)  
{  
    . . .  
}
```

Funktionsrumpf

Funktionskopf

Die formalen Parameter werden innerhalb der Funktion wie Variablen behandelt.

84

Funktionen

Funktionsdefinition

Syntax:

Hat eine Funktion keinen Rückgabewert (im mathematisches Sinne ist sie dann auch keine Funktion), dann wird als Rückgabetyp `void` verwendet. Funktionen mit Rückgabewert `void` werden in vielen anderen Sprachen wie z.B. Pascal als **Prozedur** bezeichnet.

Wird der Rückgabetyp weggelassen, was man sich nicht angewöhnen sollte, so wird als Default-Wert vom Compiler der Rückgabetyp `int` verwendet.

Beispiel:

```
void ausgebenPLZ (int plz) {  
    printf ("Die Postleitzahl ist: ");  
    printf ("%05d\n", plz); /* Ausgabe 5 Stellen */  
    /* links mit 0en aufgefüllt */  
}
```

85

Funktionen

Funktionsdefinition

Rückgabe:

Hat eine Funktion einen von `void` verschiedenen Rückgabetyp, so **muss** sie mit `return` einen Wert zurückgeben.

Nach `return` kann ein beliebiger Ausdruck stehen:

```
    return ausdruck;
```

Die Abarbeitung der Funktion ist dann beendet. Code der nach dieser Anweisung steht wird nicht mehr ausgeführt (dead code).

Steht hinter `return` kein Ausdruck so wird die Funktion verlassen, ohne einen Wert zurückzugeben. Dies ist möglich, wenn die Funktion den Rückgabetyp `void` hat. Enthält ein Funktionsrumpf einer Funktion mit dem Rückgabetyp `void` keine `return`-Anweisung, so wird die Funktion beim Erreichen der den Funktionsrumpf abschließenden geschweiften Klammer beendet.

86

Funktionen

Funktionsdefinition

Beispiel:

```
int square (int x) {  
    return x*x;  
}  
  
unsigned fak (unsigned x) {  
    unsigned i, erg=1;  
    for (i = 2; i <= x; i++) {  
        erg *= i;  
    }  
    return erg;  
}
```

87

Funktionen

Funktionsdefinition

Deklaration:

Jede Funktion muss **vor** ihrer Verwendung **deklariert** (bekannt gemacht) werden. Die Deklaration erfolgt entweder durch die Definition der Funktion, oder durch eine separate Deklarationsanweisung.

Beispiel:

```
unsigned fak (unsigned);
```

Durch die Deklarationsanweisung wird die **Signatur** der Funktion bekanntgegeben:

- Name der Funktion
- Anzahl der Parameter
- Typen der Parameter
- Rückgabetyp

88

Funktionen

Funktionsaufruf

Syntax:

Hat eine Funktion **formale Parameter**:

```
rueckgabetyp funktionsname ( typ_1 formaler_parameter_1,
                                typ_2 formaler_parameter_2,
                                ...
                                typ_n formaler_parameter_n) {  
    ...  
}
```

so muss beim Aufruf an jeden formalen Parameter ein **aktueller Parameter** übergeben werden:

```
funktionsname (aktueller_parameter_1,
                aktueller_parameter_2,
                ...
                aktueller_parameter_n)
```

89

Funktionen

Funktionsaufruf

Beispiel:

```
#include <stdio.h>
unsigned fak (unsigned x) {
    unsigned i, erg=1;
    for (i = 2; i <= x; i++) {
        erg *= i;
    }
    return erg;
}
int main (void) {
    unsigned zahl, erg;
    printf ("Bitte Zahl eingeben:\n");
    scanf("%d",&zahl);
    erg = fak(zahl); /* Funktionsaufruf */
    printf("Die Fakultät von %d ist %d!\n", zahl,erg);
    return 0;
}
```

90

Funktionen

Funktionsaufruf

Ein aktueller Parameter kann dabei ein beliebig komplexer Ausdruck sein.
Beispiele:

```
erg = fak(zahl*2);  
erg = fak(zahl*zahl+10);  
erg = fak((fak(zahl));  
erg = fak(fak(zahl+1)*2);
```

Übergeben wird an die Funktion immer der **Wert** des Ausdrucks (**call-by-value**), d.h. der Ausdruck wird ausgewertet, dann wird der Wert an die Funktion übergeben (d.h. dem formale Parameter der Funktion wird der Wert zugewiesen) und dann erst wird die Funktion ausgeführt.

Funktionen können beliebig viele Parameter haben. Die Anzahl der aktuellen Parameter und der Typ muss immer mit der Anzahl der formalen Parameter übereinstimmen.

Funktionen

Funktionsaufruf

Beispiel:

```
int power(unsigned base, unsigned exponent) {  
    unsigned i, result = 1;  
    for (i = 1; i <= exponent; i++) {  
        result *= base;  
    }  
    return result ;  
}  
int main (void) {  
    unsigned b, e;  
    printf („Bitte Basis und Exponent eingeben:\n“);  
    scanf(„%d“,&b);  
    scanf(„%d“,&e);  
    printf(„%d hoch %d ist %d!\n“,b,e,power(b,e));  
    return 0;  
}
```

Funktionen

Funktionsaufruf

Achtung: Die Reihenfolge der Auswertung der aktuellen Parameter ist in C nicht festgelegt! Kann von links nach rechts oder von rechts nach links erfolgen.

Beispiel:

```
result = power(b*2,e+5);
```

Im Normalfall spielt das aber auch keine Rolle. Aber in C können Ausdrücke sog. **Seiteneffekte** haben, d.h. es erfolgt zusätzlich zur Wertberechnung eine Veränderung von Variablenwerten.

Beispiel:

```
result = power(--x,++x);
```

Achtung: Schreiben Sie nie(!!) solchen Code. Das Ergebnis ist kaum vorhersehbar und kann von Compiler zu Compiler unterschiedlich sein.

Funktionen

Rekursive Funktionen

Ruft sich eine Funktion selbst im Rumpf wieder auf (direkt oder indirekt), so spricht man von einer rekursiven Funktion.

Beispiel (direkte Rekursion):

```
unsigned fak (unsigned n) {
    if (n == 0) {
        return 1;
    }
    else {
        return n * fak(n-1);
    }
}
```

Funktionen

Rekursive Funktionen

Beispiel (indirekte Rekursion):

```
int iseven(unsigned);  
int isodd(unsigned);  
  
int iseven (unsigned x) {  
    if (x == 0) { return 1; }  
    else { return isodd(x-1); }  
}  
  
int isodd (unsigned x) {  
    if (x == 0) { return 0; }  
    else { return iseven(x-1); }  
}
```

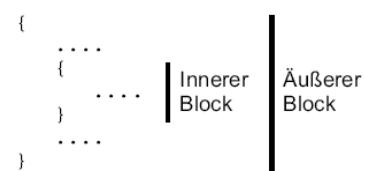
Blöcke

Definition

Blöcke sind ein Strukturierungselement für Anweisungen. Ein Block hat den folgenden Aufbau:

```
{  
    Vereinbarungen  
    Anweisungen  
}
```

- Vereinbarungen umfassen Definitionen von Variablen und Deklarationen.
- Vereinbarungen müssen immer **vor** den Anweisungen stehen (ANSI C).
- Der Rumpf einer Funktionen ist immer ein Block.
- Da eine Anweisung eines Blocks selbst wieder ein Block sein kann, können Blöcke geschachtelt werden.



Blöcke Definition

- In C können in jedem Block – auch in inneren Blöcken – Vereinbarungen durchgeführt werden.
- Variablen, die außerhalb vom äußersten Block definiert werden, werden **globale Variablen** genannt.
- Variablen, die innerhalb eines Blockes definiert sind werden **lokale Variablen** genannt.
- Achtung: C erlaubt es nicht lokale Funktionen, also z.B. Funktionen innerhalb einer anderen Funktion zu definieren. Sie müssen immer ganz außen definiert werden.

Blöcke Definition

Beispiel:

```
int global = 10; /* globale Variable */

int main(void) {
    int local1 = 1; /* lokale Variable */
    if (local1 > 0) {
        int local2 = 4; /* lokale Variable */
        printf ("y hat den Wert %d\n", local2);
    }
    else{
        int local3 = 4; /* lokale Variable */
        printf ("\nx hat den Wert %d\n", local3);
    }
}
```

Blöcke Lebensdauer und Gültigkeit

Die **Lebensdauer** einer Variable ist die Zeitspanne, in der das Laufzeitsystem des Compilers der Variablen einen Platz im Speicher zur Verfügung stellt.

Für die Lebensdauer von Variablen gilt:

- Globale Variablen leben solange wie das Programm.
 - Lokale Variablen werden beim Betreten des Blockes angelegt und beim Verlassen des Blockes wieder ungültig.
 - Lokale Variablen leben auch in allen weiter innen liegenden Blöcken.

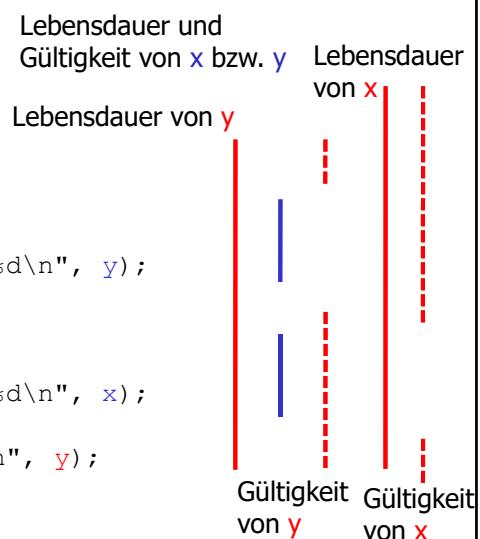
Die **Gültigkeit/Sichtbarkeit** einer Variable ist die Lebensdauer der Variable abzüglich aller eingeschlossenen Blöcke, in denen der Bezeichner erneut definiert ist. Ist eine Variable sichtbar, dann kann über den Namen auf diese Variable zugegriffen werden. Durch Namensgleichheit werden Variablen **verschattet**.

Blöcke

Definition

Beispiel:

```
int x = 10;
int main (void) {
    int y = 1;
    if (x > 0) {
        int y = 4;
        printf ("y hat den Wert %d\n", y);
    }
    else{
        int x = 4;
        printf ("x hat den Wert %d\n", x);
    }
    printf ("y hat den Wert %d\n", y);
}
```



Blöcke

Lebensdauer und Gültigkeit

- Formale Parameter verschatten globale Variable.
- Lokale Bezeichner verschatten Funktionsbezeichner.

Beispiel:

```
int quadrat (int n){  
    return n * n;  
}  
int main (void){  
    int resultat;  
    int quadrat;  
    int x = 5;  
    resultat = quadrat (x);  
    printf ("%d", resultat);  
    return 0;  
}
```

101

Zeiger

Zeigertypen und Zeigervariable

Der Arbeitsspeicher eines Rechners ist in Speicherzellen eingeteilt, die durchnummeriert sind. Die Nummer einer Speicherzelle wird als **Adresse** bezeichnet. Ist der Speicher eines Rechners byteweise (1 Byte = 8 Bits) ansprechbar, so sagt man, er sei **byteweise adressierbar**.

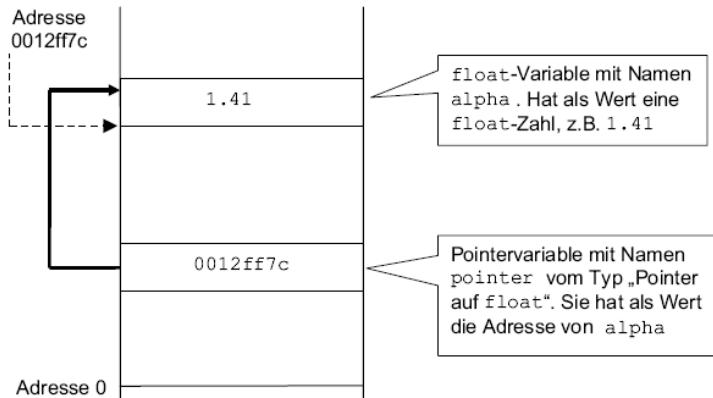
Ein **Zeiger (Pointer)** ist eine Variable, welche die Adresse einer im Speicher befindlichen Variablen oder Funktion aufnehmen kann. Damit verweist eine Zeigervariable mit ihrem Variablenwert auf die jeweilige Adresse.

Zeigervariable werden in C genauso behandelt wie „normale“ Variable.

102

Zeiger Zeigertypen und Zeigervariable

Arbeitsspeicher



103

Zeiger Zeigertypen und Zeigervariable

Definition einer Zeigervariable:

Ein Zeiger wird formal wie eine Variable definiert – dem Zeigernamen ist lediglich ein Stern vorangestellt:

Typname * Zeigername;

Zeigername ist der Name des Zeigers. Typname * ist der Datentyp des Zeigers. Diese Definition wird von rechts nach links gelesen, wobei man den * als „ist Zeiger auf“ liest. Die Definition wird also gelesen zu:

„Zeigername ist Zeiger auf Typname“.

Der Datentyp des Zeigers heißt:

„Zeiger auf Typname“.

Beispiel:

```
int* zeiger1;
```

104

Zeiger

Zeigertypen und Zeigervariable

Achtung:

```
int * pointer, alpha;  
entspricht
```

```
int * pointer;  
int alpha;
```

Der Wertebereich einer Zeigervariable vom Typ „Zeiger auf Typname“ ist die Menge aus allen Zeigern, die auf Speicherobjekte vom Typ Typname zeigen können, und dem NULL-Pointer. Der **Pointer NULL** ist ein vordefinierter Pointer und zeigt auf die Adresse 0 und damit auf kein gültiges Speicherobjekt:

```
int * pointer = NULL;  
/* pointer zeigt auf die Adresse NULL */
```

Zeiger

Zeigertypen und Zeigervariable

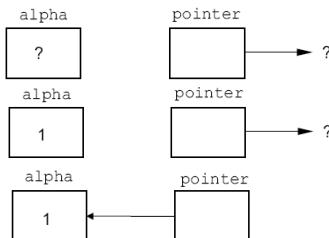
Adressoperator:

Die einfachste Möglichkeit, einen Zeiger auf ein Objekt zeigen zu lassen, besteht darin, auf der rechten Seite des Zuweisungsoperators den Adressoperator & auf eine Variable anzuwenden, denn es gilt:

Ist x eine Variable vom Typ Typname, so liefert der Ausdruck $\&x$ einen Zeiger auf das Objekt x vom Typ „Zeiger auf Typname“.

Beispiel:

```
int alpha;  
int *pointer;  
  
alpha = 1;  
pointer = &alpha;
```



Zeiger

Zeigertypen und Zeigervariable

Dereferenzierung

Wurde einem Zeiger ein Wert zugewiesen, so will man natürlich auch auf das referenzierte Objekt, d.h. auf das Objekt, auf das der Zeiger zeigt, zugreifen können. Dazu gibt es in C den **Inhaltsoperator** *. Wird er auf einen Zeiger angewandt, so greift er auf das Objekt zu, auf das der Zeiger verweist. Statt Inhaltsoperator sagt man oft auch **Dereferenzierungsoperator**. Mit Hilfe des Dereferenzierungsoperators erhält man aus einem Zeiger, also einer Referenz auf ein Objekt, das Objekt selbst. Ist

```
int alpha = 1;
```

und

```
int * pointer = &alpha;
```

dann wird mit

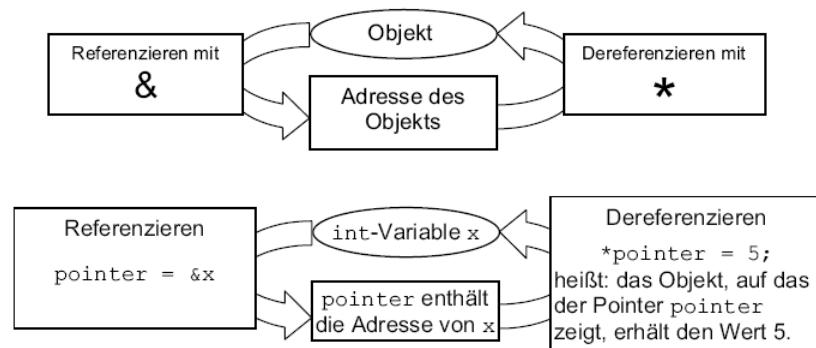
```
*pointer = 2; /* Dereferenzierung */
```

der Variablen alpha der Wert 2 zugewiesen.

Zeiger

Zeigertypen und Zeigervariable

Referenzierung und Dereferenzierung



Zeiger

Zeigertypen und Zeigervariable

Beispiel:

```
int main (void) {  
    float zahl = 3.5f;  
    printf("Adresse von zahl: %p\n", &zahl);  
    printf("Wert von zahl: %f\n", zahl);  
    printf("Wert von *&zahl = %f\n", *&zahl);  
    return 0;  
}
```

Ein häufiger Fehler ist im folgenden Beispiel zu sehen:

```
int * pointer;  
*pointer = 6;
```

Zeiger

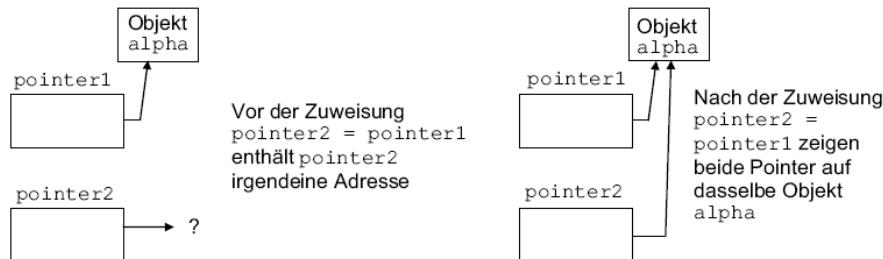
Zeigertypen und Zeigervariable

Beispiel:

```
int main (void) {  
    int alpha;  
    int * pointer1, * pointer2;  
    pointer1 = &alpha; /* pointer1 wird initialisiert */  
    /* und zeigt auf alpha */  
    *pointer1 = 5; /* alpha wird 5 zugewiesen */  
    printf ("\n%d", *pointer1); /* 5 wird ausgegeben */  
    *pointer1 = *pointer1 + 1; /* alpha wird um 1 inkr. */  
    pointer2 = pointer1; /* pointer2 wird initialisiert */  
    /* und zeigt auch auf alpha */  
    printf ("\n%d", *pointer2); /* 6 wird ausgegeben */  
    return 0;  
}
```

Zeiger Zeigertypen und Zeigervariable

Beispiel (cont.):



Zeiger Zeigertypen und Zeigervariable

Zusammenfassung:

Zum Arbeiten mit Zeigern stehen die beiden Operatoren & und * zur Verfügung:

- & wird als Adressoperator bezeichnet. Mit ihm erhält man die Adresse eines Objekts.
- * wird als Dereferenzierungsoperator bezeichnet. Mit ihm erhält man den Inhalt des Objektes, auf das ein Zeiger zeigt.

Zeiger Felder

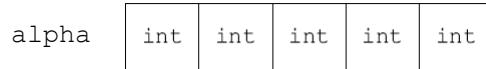
Eindimensionale Felder

Unter einem Feld (Array) versteht man die Zusammenfassung von mehreren Variablen des gleichen Typs unter einem gemeinsamen Namen. Die allgemeine Form der Definition eines eindimensionalen Feldes (Vektor) ist:

```
Typname Feldname [Anzahl];
```

Konkrete Beispiele hierfür sind:

```
int alpha [5]; /* Feld aus 5 Elementen vom Typ int */  
char beta [6]; /* Feld aus 6 Elementen vom Typ char */
```



Zeiger Felder

Eindimensionale Felder

- In C gibt es im Gegensatz zu anderen Sprachen kein Schlüsselwort **array**. Der C Compiler erkennt ein Array an den **eckigen Klammern**, die bei der Definition die Anzahl der Elemente enthalten.
- Die Anzahl der Elemente muss immer eine positive ganze Zahl sein. Sie kann gegeben sein durch eine **Konstante** oder einen **konstanten Ausdruck**, nicht aber durch eine Variable.
- Dies bedeutet, dass die Größe **nicht dynamisch** zugeordnet werden kann.
- Der Zugriff auf ein Element eines Feldes erfolgt über den Feldindex. Hat man ein Feld mit n Elementen definiert, so ist darauf zu achten, dass in C die Indizierung der Arrayelemente mit **0 beginnt und bei n-1 endet**.

Zeiger Felder

Eindimensionale Felder

Das folgende Beispiel zeigt ein Array aus 5 int-Elementen:

```
int alpha [5]; /* Arraydefinition */
alpha[0]=1; /* das 1. Element alpha[0] hat den Index 0 */
alpha[1]=2; /* das 2. Element alpha[1] hat den Index 1 */
alpha[2]=3; /* das 3. Element alpha[2] hat den Index 2 */
alpha[3]=4; /* das 4. Element alpha[3] hat den Index 3 */
alpha[4]=5; /* das letzte Element hat den Index 4 */
```

alpha[0] int	alpha[1] int	alpha[2] int	alpha[3] int	alpha[4] int
1	2	3	4	5

Zeiger Felder

Eindimensionale Felder

Eine tückische Besonderheit von Arrays in C ist, dass beim Überschreiten des zulässigen Indexbereiches kein Kompilier- bzw. Laufzeitfehler erzeugt wird. So würde die folgende Anweisung einfach die Speicherzelle direkt nach `alpha[4]` mit dem Wert 6 überschreiben:

```
alpha[5] = 6;
```

Manche Compiler können so eingestellt werden, dass sie vor solchen einfachen Fehlern warnen.

Das Werkzeug **lint**, welches auf UNIX-Rechnern in der Regel mit dem C-Compiler ausgeliefert wird und erweiterte Prüfungen des C-Quellcodes durchführt, erkennt diesen Fehler jedoch sofort.

Ist der Arrayindex allerdings eine Variable, also z.B. `alpha[i]` kann im allgemeinen Fall der Fehler nicht zur Übersetzungszeit erkannt werden.

Zeiger Felder

Eindimensionale Felder

Der Vorteil von Arrays gegenüber mehreren einfachen Variablen ist, dass Arrays sich leicht mit Schleifen bearbeiten lassen. Im Gegensatz zur Größe des Arrays, die konstant ist, kann der Index einer Array-Komponente eine Variable sein:

```
int index;
float summe = 0;
for (index = 0; index < 5; index = index + 1) {
    printf ("\n%d", alpha[index]);
    summe = summe + alpha[index];
}
printf ("\nDer Durchschnitt ist: %f\n", summe / index);
```

117

Zeiger Felder

Eindimensionale Felder

Es ist erlaubt, innerhalb einer einzigen Vereinbarung sowohl einfache Variablen, als auch Arrays zu definieren, z.B.

```
float alpha [10], beta, gamma [5];
```

Hierdurch wird definiert:

ein Array alpha mit 10 Elementen vom Typ float, eine einfache float-Variable mit Namen beta, ein Array namens gamma mit 5 Elementen vom Typ float.

Auf jeden Fall sollte man es sich bei Arrays zur Gewohnheit machen, immer mit symbolischen Konstanten und nie mit literalen Konstanten zu arbeiten:

```
#define MAX 40
int alpha[MAX];
for (index = 0; index < MAX; index = index + 1) ...
```

118

Zeiger Felder

Initialisierung von Feldern

Bei der **manuellen** Initialisierung eines Arrays ist nach der eigentlichen Definition des Arrays ein Zuweisungsoperator gefolgt von einer Liste von Initialisierungswerte anzugeben. Diese Liste ist begrenzt durch geschweifte Klammern. Sie enthält die einzelnen Werte getrennt durch Kommata. Als Werte können Konstanten oder Ausdrücke aus Konstanten angegeben werden.

Beispiel:

```
int alpha [3] = {1, 2 * 5, 3};
```

Bei der Initialisierung mit **impliziter Längenbestimmung** wird die Größe des Feldes nicht bei der Definition angegeben. Die Größe wird vom Compiler durch Abzählen der Anzahl der Elemente in der Initialisierungsliste festgelegt.

Beispiel:

```
int alpha [] = {1, 2, 3, 4};
```

Zeiger Felder

Zeichenketten

Eine konstante Zeichenkette (String) besteht aus einer Folge von Zeichen, die in Anführungszeichen eingeschlossen sind, wie z.B. "hello". Eine konstante Zeichenkette wird vom Compiler intern als ein Array von Zeichen gespeichert. Dabei wird als letztes Element des Arrays automatisch ein zusätzliches Zeichen, das Zeichen '\0' (Nullzeichen) angehängt, um das Stringende anzuzeigen.

Wie jede andere Konstante auch, stellt eine konstante Zeichenkette einen Ausdruck dar. Der Rückgabewert einer konstanten Zeichenkette ist ein Pointer auf das erste Zeichen der Zeichenkette. Der Typ des Rückgabewertes ist `char*`.

Beispiel:

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

Zeiger Felder

Zeichenketten

Zeichenketten können auch wie normale Felder initialisiert werden.

```
char buffer [20]={'Z','e','i','c','h','e','n',
                   'k','e','t','t','e','r','\0'};
```

Bei der Speicherung von Zeichenketten muss stets ein Speicherplatz für das Nullzeichen vorgesehen werden. So hat beispielsweise

```
char vorname [15];
```

nur Platz für 14 Buchstaben einer Zeichenkette und das abschließende '\0'-Zeichen.

Eine direkte Zuweisung einer Zeichenkette an einen Vektor kann nur bei der Initialisierung erfolgen. Eine Zuweisung einer Zeichenkette mit Hilfe des Zuweisungsoperators ist im weiteren Programmverlauf **nicht** möglich!

Zeiger Felder

Zeichenketten

```
int index;
char eingabe [MAX+1];
...
for (index=0; eingabe[index]!='\0'; index= index+1){
    if (eingabe[index] == 'a'){
        break;
    }
}
if (eingabe[index] == '\0'){
    printf ("\nIhr String enthaelt kein 'a'\n");
}
else{
    printf("\nDas a befand sich an der %d. Stelle\n",
           index + 1);
}
```

Zeiger Parameterübergabe

Die meisten Programmiersprachen kennen zwei Arten von Funktionsparametern:

- **Wertparameter:** Beim Aufruf der Funktion wird der Wert des Ausdrucks an die Funktion übergeben (call by value).
- **Referenzparameter:** Beim Aufruf der Funktion wird nicht der Wert einer Variable übergeben, sondern die Variable selbst (Variablenparameter), bzw. eine Referenz auf diese Variable (call by reference). Als aktuelle Parameter können nur Variablen verwendet werden.

In C ist eine call by reference-Schnittstelle als Sprachmittel nicht vorgesehen. Man kann das Verhalten einer call by reference-Schnittstelle auch mit der call by value-Schnittstelle erreichen, indem man einen Pointer auf den aktuellen Parameter mit call by value übergibt.

123

Zeiger Parameterübergabe

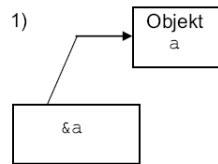
Beispiel:

```
void init (int *alpha) {  
    *alpha = 10;  
}  
  
int main (void) {  
    int a;  
    init (&a);  
    printf ("Der Wert von a ist %d", a);  
    return 0;  
}
```

124

Zeiger Parameterübergabe

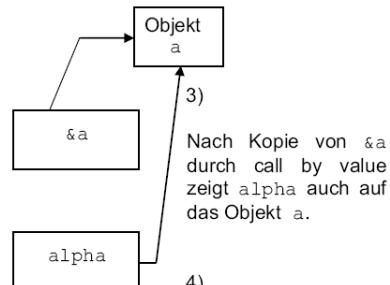
Beispiel:



Vor dem Funktionsaufruf von init() wird der aktuelle Parameter von init() berechnet.

2)

Der Wert des aktuellen Parameters &a wird an den formalen Parameter, den Pointer alpha zugewiesen. In anderen Worten, alpha wird mit der Adresse von a initialisiert.



3)
Nach Kopie von &a durch call by value zeigt alpha auch auf das Objekt a.

4)

*alpha = 10
Dem Objekt, auf das alpha zeigt, wird der Wert 10 zugewiesen.

Zeiger Parameterübergabe

Referenzparameter:

- Können verwendet werden um Ergebnisse an den Aufrufer zurückzugeben. Damit können neben dem Funktionsergebnis noch weitere Berechnungsergebnisse zurückgegeben werden.
- Können verwendet werden, um die Übergabe von Parametern zu beschleunigen (z.B. von großen Datentypen).
- Müssen immer bei der Übergabe von Feldern und Strings verwendet werden.

Zeiger

Parameterübergabe

Referenzparameter:

Durch das Schlüsselwort `const` kann ein Referenzparameter vor dem Überschreiben geschützt werden. Damit kann auf einen solchen Parameter nur lesend zugegriffen werden. Der Aufrufer einer solchen Funktion kann damit sicher sein, dass seine Variable nicht verändert wird.

Beispiel:

```
void f (const int * pointer) {  
    *pointer = 15; /* Fehler ! */  
    ...  
}
```

Achtung: Nicht der Zeiger ist damit konstant, sondern der Speicher auf den der Zeiger verweist wird als Konstante verwendet!

Zeiger

Parameterübergabe

```
#include <stdio.h>  
  
void fake(const int* arg) {  
    int* sneaky = arg;  
    *sneaky = 1;  
}  
  
int main() {  
    int x = 0;  
    fake(&x);  
    printf("x = %d", x);  
    return 0;  
}
```

Dies ist aber keine 100% Garantie!



Mann kann einen `const` Zeiger in einen normalen Zeiger umwandeln und dann doch schreibend zugreifen. Der Compiler erzeugt zwar eine Warnung aber der C Code ist korrekt, wenn auch normalerweise schlechter Stil.

Zeiger

Zeigerarithmetik

Unter dem Begriff **Zeigerarithmetik** fasst man die Menge der zulässigen Operationen auf Zeigern zusammen:

- **Zuweisung:**

Zeiger dürfen Zeigervariablen zugewiesen werden. Dabei müssen die beiden Datentypen übereinstimmen. Zeiger vom Typ void * dürfen Zeigern eines anderen Datentyps zugewiesen werden und Zeiger eines beliebigen Datentyps dürfen Zeigern vom Typ void * zugewiesen werden. Der NULL-Pointer kann ebenfalls jedem anderen Zeiger zugewiesen werden.

- **Addition und Subtraktion:**

Zeiger können unter bestimmten Voraussetzungen (z.B. Verweis auf Elemente desselben Feldes) voneinander abgezogen werden. Zu einem Zeiger kann eine ganze Zahl addiert oder von ihm abgezogen werden.

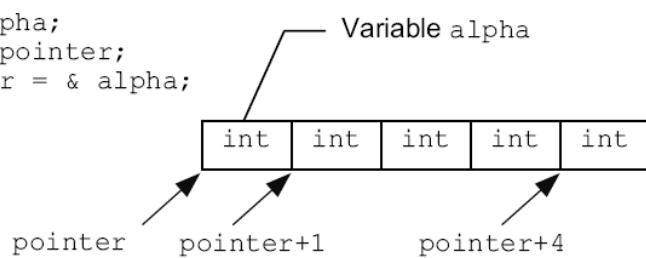
Zeiger

Zeigerarithmetik

- **Addition und Subtraktion (cont)**

Wird ein Pointer vom Typ int * um 1 erhöht, so zeigt er um ein int-Objekt weiter. Wird ein Pointer vom Typ float * um 1 erhöht, so zeigt er um ein float-Objekt weiter. Die Erhöhung um 1 bedeutet, dass der Pointer immer um ein Speicherobjekt vom Typ, auf den der Pointer zeigt, weiterläuft.

```
int alpha;
int * pointer;
pointer = & alpha;
```



Zeiger

Zeigerarithmetik

- **Vergleiche**

Zwei Zeiger können auf Gleichheit bzw. Ungleichheit verglichen werden, wenn beide Zeiger denselben Typ haben oder einer der beiden der NULL-Pointer ist. Ein „größer“ oder „kleiner“ ist unter bestimmten Voraussetzungen (z.B. Verweis auf Elemente desselben Feldes) möglich.

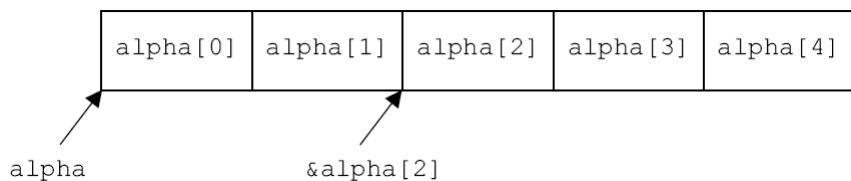
Wenn zwei Pointer auf dasselbe Speicherobjekt zeigen oder beide NULL sind, so ergibt der Test auf Gleichheit (==) den Booleschen Wert „wahr“.

Zeiger

Äquivalenz von Array- und Pointer-Notation

Der Name eines Arrays kann als konstanter Zeiger auf das erste Element des Arrays verwendet werden. Damit gibt es für das erste Element zwei gleichwertige Schreibweisen:

- $\alpha[0]$
- und mit Verwendung des Dereferenzierungsoperators auch $*\alpha$.



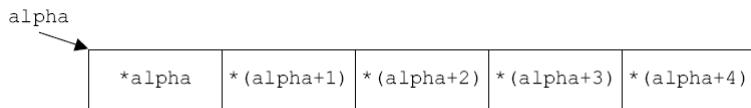
Zeiger

Äquivalenz von Array- und Pointer-Notation

alpha sei ein Vektor. Dann gilt:

$$\text{alpha}[i] = *(\text{alpha} + i).$$

Dies bedeutet, dass die Arraynotation äquivalent ist zu einer Pointernotation bestehend aus Dereferenzierungsoperator, Pointer und Abstand.



Diese Äquivalenz gilt auch in der anderen Richtung: pointer sei ein Pointer. Dann kann man statt

$$*(\text{pointer} + i) \text{ auch } \text{pointer}[i]$$

schreiben. Der Compiler arbeitet intern nicht mit Indizes. Erhält er eine Array-Komponente, so rechnet er den Index sofort in einen konstanten Pointer um.

Zeiger

Übergabe von Feldern und Zeichenketten

Bei der Übergabe eines Feldes an eine Funktion wird als aktueller Parameter der Feldname übergeben. Der Feldname stellt dabei einen Pointer auf das erste Element des Feldes dar.

Ein Feld wird deshalb immer als Referenzparameter übergeben, d.h. das Feld wird nicht kopiert!

Der formale Parameter für die Übergabe eines eindimensionalen Feldes kann ein offenes Array sein – oder wegen der Pointereigenschaft des Arraynamens – auch ein Pointer auf den Komponententyp des Arrays.

Da Zeichenketten vom Compiler intern als char-Arrays gespeichert werden, ist die Übergabe von Zeichenketten identisch mit der Übergabe von char-Arrays. Der formale Parameter einer Funktion, die eine Zeichenkette übergeben bekommt, kann vom Typ `char *` oder `char[]` sein.

Zeiger

Übergabe von Feldern und Zeichenketten

Beispiel:

```
void init (int alpha[], int x) { /* Referenzübergabe */
    /* (int *alpha, */
    int i;
    for (i=0; i<MAX; i++) {
        alpha[i]=x;
    }
}
int main (void) {
    int feld[MAX];
    init (feld, x);
    ...
    return 0;
}
```

135

Zeiger

Übergabe von Feldern und Zeichenketten

Da Felder und Zeichenreihen immer per Referenz übergeben werden ist in diesem Zusammenhang das Schlüsselwort `const` sehr wichtig um zu signalisieren, dass die Zeichenreihe bzw. die Feldinhalte nicht verändert werden.

Im folgenden Beispiel

```
printf ("Hier bin ich\n");
```

bekommt die Funktion `printf()` einen Pointer auf die konstante Zeichenkette "Hier bin ich\n" als Argument übergeben. Der Funktionsprototyp von `printf()` zeigt durch den formalen Parameter in der Definition

```
int printf (const char * formatstring, ...);
```

dass nur lesend und nicht schreibend auf die konstante Zeichenkette zugegriffen werden kann.

136

Zeiger

Äquivalenz von Array- und Pointer-Notation

Vergleich von Arrays

In C ist es mit dem Vergleichsoperator nicht möglich, zwei Vektoren auf identischen Inhalt zu überprüfen, wie z.B. durch

```
arr1 == arr2
```

Der Grund dafür ist, dass die Vektornamen äquivalent sind zur Speicheradresse des ersten Vektorelements. Es wird mit `arr1 == arr2` also nur verglichen, ob `arr1` und `arr2` auf dieselbe Adresse zeigen.

Für einen echten Vergleich gibt es zwei Möglichkeiten:

- Überprüfung der einzelnen Vektorelemente in einer Schleife.
- Verwendung von standardisierten Funktionen aus einer C-Library (z.B. `strcmp`).

Zeiger

Funktionen zur Bearbeitung von Strings

Im Folgenden werden aus der Menge der Stringverarbeitungsfunktionen die Funktionen

- `strlen()` /* Zum Ermitteln der Stringlänge */
- `strcpy()`, /* Zum Kopieren von Strings */
- `strcat()`, /* Zum Anhängen eines Strings an einen anderen */
- `strcmp()`, /* Zum Vergleichen von Strings */
- `strncmp()` /* Zum Vergleichen von Strings */

vorgestellt. Nach dem ISO-Standard wird als Include-Datei für die Funktionen zur Stringbearbeitung die Datei `string.h` benötigt.

Zeiger

Funktionen zur Bearbeitung von Strings

Die Funktion `strlen()`

Syntax:

```
size_t strlen (const char * s);
```

Beschreibung:

Die Funktion `strlen()` liefert als Rückgabewert die Anzahl der Zeichen des Strings, auf den `s` zeigt. Das Stringende-Zeichen '`\0`' wird dabei nicht mitgezählt.

Beispiel:

```
char string [100] = "So lang ist dieser String:";  
printf ("So gross ist das Array: %d\n", sizeof (string));  
printf ("%s %d\n", string, strlen (string));
```

139

Zeiger

Funktionen zur Bearbeitung von Strings

Die Funktion `strcpy()`

Syntax:

```
char * strcpy (char * dest, const char * src);
```

Beschreibung:

Die Funktion `strcpy()` kopiert den Inhalt des Strings, auf den `src` zeigt, an die Adresse, auf die `dest` zeigt. Kopiert wird der gesamte Inhalt einschließlich des Stringende-Zeichens '`\0`'.

Die Funktion `strcpy()` überprüft dabei nicht, ob der Puffer, dessen Adresse übergeben wurde, genügend Platz zur Verfügung stellt. Hierfür muss der Programmierer selbst Sorge tragen.

Die Funktion `strcpy()` gibt als Rückgabewert den Pointer `dest` zurück.

140

Zeiger

Funktionen zur Bearbeitung von Strings

Die Funktion `strcpy()`

Beispiel:

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

int main (void) {
    char string1 [25];
    char string2 [] = "Zu kopierender String";
    printf ("Der kopierte String ist: %s\n", strcpy
            (string1,string2));
    return 0;
}
```

141

Zeiger

Funktionen zur Bearbeitung von Strings

Die Funktion `strcat()`

Syntax:

```
char * strcat (char * dest, const char * src);
```

Beschreibung:

Die Funktion `strcat()` hängt an den String, auf den `dest` zeigt, den String an, auf den `src` zeigt. Dabei wird das Stringende-Zeichen '\0' des Strings, auf den `dest` zeigt, vom ersten Zeichen des Strings, auf den `src` zeigt, überschrieben. Angehängt wird der gesamte String, auf den `src` zeigt, einschließlich des Zeichens '\0'. Die Funktion `strcat()` prüft dabei nicht, ob genügend Speicher im String, auf den `dest` zeigt, vorhanden ist.

Die Funktion `strcat()` gibt als Rückgabewert den Pointer `dest` zurück.

142

Zeiger

Funktionen zur Bearbeitung von Strings

Die Funktion `strcat()`

Beispiel:

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

int main (void) {
    char string [50] = "concatenate";
    printf ("%s\n", string);
    printf ("%s\n",
           strcat (string, " = zusammenfuegen"));
    return 0;
}
```

143

Zeiger

Funktionen zur Bearbeitung von Strings

Die Funktion `strcmp() / strncmp()`

Syntax:

```
int strcmp (const char * s1, const char * s2);
int strncmp (const char * s1, const char * s2, size_t n);
```

Beschreibung:

Die Funktion `strcmp()` führt einen zeichenweisen Vergleich der beiden Strings, auf die `s1` und `s2` zeigen, durch. Die beiden Strings werden solange verglichen, bis ein Zeichen unterschiedlich oder bis ein Stringende-Zeichen '`\0`' erreicht ist.

Die Funktionsweise entspricht der Funktion `strcmp()`. Hinzu kommt aber noch das zusätzliche Abbruchkriterium `n`. Spätestens nach `n` Zeichen oder gegebenenfalls schon früher beim Erkennen eines Nullzeichens '`\0`' oder zweier unterschiedlicher Zeichen in den beiden Strings wird der Vergleich beendet.

144

Zeiger

Funktionen zur Bearbeitung von Strings

Die Funktion `strcmp()`

Beschreibung (cont.):

Die Funktion `strcmp()` gibt folgende Rückgabewerte zurück:

- < 0 wenn der String, auf den s1 zeigt, lexikografisch kleiner ist als der String, auf den s2 zeigt,
- == 0 wenn der String, auf den s1 zeigt, lexikografisch gleich dem String ist, auf den s2 zeigt,
- > 0 wenn der String, auf den s1 zeigt, lexikografisch größer ist als der String, auf den s2 zeigt.

Zeiger

Funktionen zur Bearbeitung von Strings

Die Funktion `strcmp()`

Beispiel:

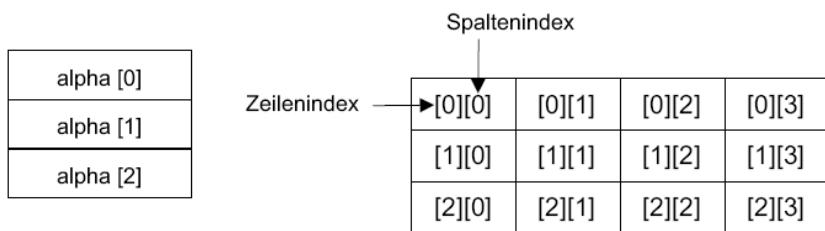
```
#include <stdio.h>
#include <string.h>
int main (void) {
    printf ("%d\n", strcmp ("abcde", "abCde")); /* 1 */
    printf ("%d\n", strcmp ("abcde", "abcde")); /* 0 */
    printf ("%d\n", strcmp ("abcd", "abcde")); /* -1 */
    return 0;
}
```

Zeiger Mehrdimensionale Felder

In C ist es wie in anderen Programmiersprachen möglich, mehrdimensionale Arrays zu verwenden. Mehrdimensionale Arrays entstehen durch das Anhängen zusätzlicher eckiger Klammern, wie im folgenden Beispiel:

```
int alpha [3] [4];
```

Interpretiert man `alpha[3][4]` in Gedanken als einen Vektor so hat jedes dieser Vektorelemente selbst wieder 4 Elemente.



147

Zeiger Mehrdimensionale Felder

Jedes Element kann durch eine Zuweisung wie z.B.

```
alpha[1][3] = 6;
```

initialisiert werden. Wie ein eindimensionales Array kann auch ein mehrdimensionales Array bereits bei seiner Definition initialisiert werden, beispielsweise:

148

Zeiger Mehrdimensionale Felder

```
int alpha [3][4] =  
{  
    {1, 3, 5, 7},  
    {2, 4, 6, 8},  
    {3, 5, 7, 9},  
};
```

1	3	5	7
2	4	6	8
3	5	7	9

Theoretisch können beliebig viele Felder ineinander geschachtelt werden, wodurch n-dimensionale Strukturen entstehen.

Für Verarbeitung von mehrdimensionalen Feldern bieten sich geschachtelte for-Schleifen mit einer Laufvariable für jede Dimension an.

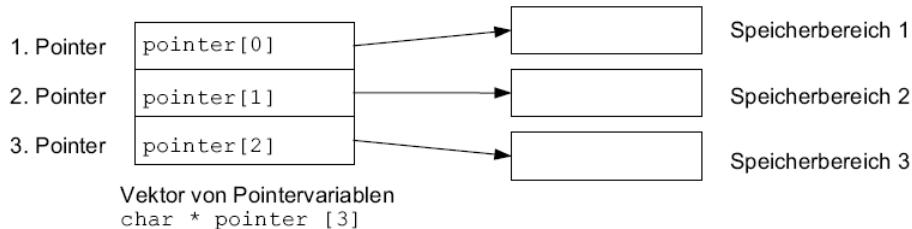
Zeiger Mehrdimensionale Felder

```
void plusplus (int matrix[3][4]) {  
    int i,j;  
    for (i=0; i<3; i++) {  
        for (j=0; j<4; j++) {  
            matrix[i][j]++;
        }
    }
}  
void print (const int matrix[3][4]) {  
    int i,j;  
    for (i=0; i<3; i++) {  
        for (j=0; j<4; j++) {  
            printf("%d\t",matrix[i][j]);
        }
        printf("\n");
    }
}
```

Zeiger

Vektoren von Zeigern

Ein Zeiger ist eine Variable, in der die Adresse eines anderen Speicherobjektes gespeichert ist. Entsprechend einem eindimensionalen Vektor von gewöhnlichen Variablen kann natürlich auch ein eindimensionaler Vektor von Zeigervariablen gebildet werden.

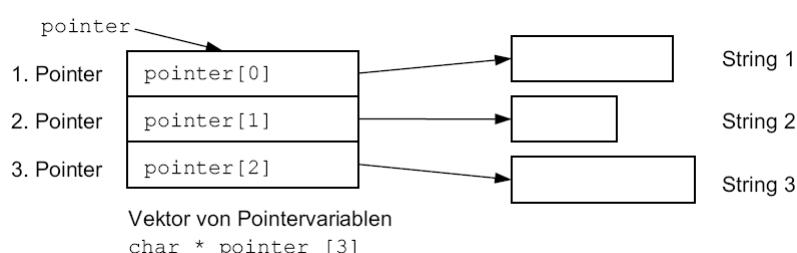


In der Praxis wird häufig mit Vektoren von Zeigern im Zusammenhang mit Strings von unterschiedlicher Länge gearbeitet (vgl. argv).

Zeiger

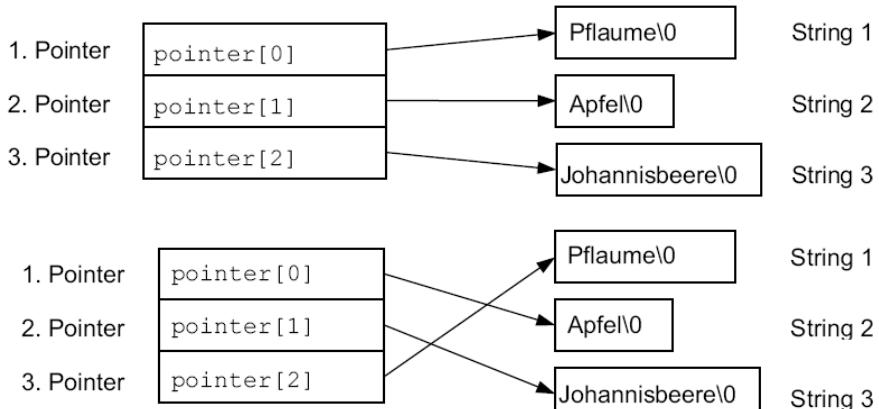
Vektoren von Zeigern

Arbeitet man mit mehreren Zeichenketten, deren Länge nicht von vornherein bekannt ist, so verwendet man ein Array von Zeigern auf char .



Will man beispielsweise diese Strings sortieren, so muss dies nicht mit Hilfe von aufwändigen Kopieraktionen durchgeführt werden. Es werden lediglich die Zeiger so verändert, dass die geforderte Sortierung erreicht wird.

Zeiger Vektoren von Zeigern



153

Zeiger Mehrdimensionale Felder und Vektoren von Zeigern

Der Unterschied zwischen einem eindimensionalen Vektor von Zeigern und einem mehrdimensionalen Array ist, dass bei mehrdimensionalen Arrays die Anzahl der Elemente fest vorgegeben ist, bei Vektoren von Zeigern hingegen nur die Anzahl an Zeigern. So ist

```
int vektor_2d [5][10];
```

ein Vektor mit insgesamt 50 int-Elementen und

```
int * pointer_vektor [5];
```

ein Vektor von 5 Zeigern auf int. Der Vorteil des pointer_vektor besteht darin, das die '2-te Dimension' der einzelnen Elemente des Vektors unterschiedlich groß sein kann. D.h. im Gegensatz zum vektor_2d muss nicht jedes Element 10 int-Werte haben.

Die häufigste Anwendung besteht deshalb darin, einen Vektor unterschiedlich langer Strings zu bilden.

154

Zeiger

Mehrdimensionale Felder und Vektoren von Zeigern

```
char * err_desc [] = {  
    "Fehlercode existiert nicht", /* 27 Bytes */  
    "Fehlertext 1", /* 13 Bytes */  
    "Fehlertext 2" }; /* 13 Bytes */
```

Mit dieser Definition werden insgesamt 53 Bytes für die Zeichen der Zeichenketten und $3 * \text{sizeof}(\text{char} *)$ Bytes für die Zeiger benötigt, d.h. in der Regel 12 Bytes. Zum Vergleich:

```
char err_desc1 [] [27] = {  
    "Fehlercode existiert nicht", /* 27 Bytes */  
    "Fehlertext 1", /* 27 Bytes */  
    "Fehlertext 2" }; /* 27 Bytes */
```

Hier muss die Anzahl an Elementen reserviert werden, die der längste String benötigt. Dies sind 27 Bytes für "Fehlercode existiert nicht". Somit benötigt err_desc1 insgesamt 81.

Komplexe Datentypen

Aufzählungstyp

Datentyp mit endlicher Datenmenge können durch Aufzählen der Elemente definiert werden. Solche Datentypen werden **Aufzählungstypen** (engl. enumeration types) genannt.

Syntax:

```
enum type {const1, const2, ... , constn}
```

Beispiel:

```
enum Monate {JAN, FEB, MAE, APR, MAI, JUN, JUL,  
            AUG, SEP, OKT, NOV, DEZ};
```

Der Typname des definierten Typs ist enum Monate. Zulässige Werte für Variablen eines Aufzählungstyps sind die Werte der Aufzählungskonstanten in der Liste der Definition des Aufzählungstyps.

Komplexe Datentypen

Aufzählungstyp

Die erste Aufzählungskonstante in der Liste hat den Wert 0, die zweite den Wert 1 und so fort. Es ist aber auch möglich, für jede Aufzählungskonstante einen Wert explizit anzugeben. Werden einige Werte in der Liste nicht explizit belegt, so wird der Wert ausgehend vom letzten explizit belegten Wert jeweils um 1 bis zum nächsten explizit angegebenen Wert hochgezählt. Dies ist in den folgenden Beispielen zu sehen:

```
enum test {ALPHA, BETA, GAMMA}; /* ALPHA = 0, BETA = 1,  
                                     GAMMA = 2 */  
  
enum test {ALPHA=5, BETA=3, GAMMA=7}; /* ALPHA = 5,  
                                         BETA = 3, GAMMA = 7 */  
  
enum test {ALPHA=4, BETA, GAMMA=3}; /* ALPHA = 4,  
                                         BETA = 5, GAMMA = 3 */
```

Komplexe Datentypen

Aufzählungstyp

Explizite Wertzuweisung sollte nur dann erfolgen, wenn sie sinnvoll ist, wie im folgenden Beispiel:

```
enum Monate {JAN = 1, FEB, MAE, APR, MAI, JUN, JUL,  
             AUG, SEP, OKT, NOV, DEZ};
```

Hier wird vollkommen automatisch der Februar (FEB) zum Monat 2, der März (MAE) zum Monat 3, usw.

Aufzählungstypen sind geeignet, um Konstanten zu definieren. Sie stellen damit eine Alternative zu der Definition von Konstanten mit Hilfe der Präprozessor-Anweisung `#define` dar.

Man kann in C zwar Variablen eines Aufzählungstyps definieren. Dennoch wird von einem C-Compiler nicht verlangt, zu prüfen, ob einer Variablen ein zulässiger Wert (eine Aufzählungskonstante) zugewiesen wird. Es findet hier also **keine** Typprüfung statt:

Komplexe Datentypen

Strukturtyp

Der **Strukturtyp** entspricht dem **Produktyp** aus GDI. In Pascal wird dieser auch **Record** genannt. Der Vorteil einer Struktur ist, dass es damit möglich ist, logisch zusammengehörige Daten in einem Datentyp zusammenzufassen und zur Speicherung von diesen Daten nur eine Variable verwenden zu müssen.

Ein Strukturtyp ist ein selbst definierter zusammengesetzter Datentyp, welche aus einer festen Anzahl von Komponenten besteht, die einen Namen haben. Im Unterschied zu einem Array können die Komponenten verschiedenartige Typen haben.

Syntax:

```
struct name {  
    komponententyp_1 komponente_1;  
    komponententyp_2 komponente_2;  
    . . .  
    komponententyp_n komponente_n;  
};
```

159

Komplexe Datentypen

Strukturtyp

Der hier selbst definierte Typname ist `struct name`. Ist die Struktur definiert, dann kann eine Variable von diesem neuen Typ folgendermaßen definieren:

```
struct name variable;
```

Beispiel:

```
struct adresse {  
    char strasse [20];  
    int hausnummer;  
    int postleitzahl;  
    char stadt [20];  
};  
struct student {  
    int matrikelnummer;  
    char name [20];  
    char vorname [20];  
    struct adresse wohnort;  
};
```

160

Komplexe Datentypen

Strukturtyp

Beispiele für die Definition von Variablen sind:

```
struct student meyer, mueller;
struct student semester [50]; /* ein Array aus 50 Studenten*/
```

Initialisierung:

Eine Initialisierung einer Strukturvariablen kann direkt bei der Definition der Strukturvariablen mit Hilfe einer Initialisierungsliste durchgeführt werden:

```
struct student stud = {
    66202,
    "Fuchs",
    "Max",
    {
        "Schillerplatz",
        20,
        84036,
        "Landshut"
    }
};
```

161

Komplexe Datentypen

Strukturtyp

Selektion:

Die Selektion erfolgt durch den Punktoperator. Ist eine Variable `x` vom Typ `struct name` definiert, dann kann folgendermaßen auf die einzelnen Komponenten der Struktur zugegriffen werden:

```
x.komponente_i
```

Beispiel:

```
printf("Straße: %s\n", x.strasse);
```

Konstruktion:

In C gibt es keinen vorgegebenen Konstruktor. Die Konstruktion einer Variable erfolgt einzelnen über die Selektoren:

```
x.komponente_i = ausdruck;
```

162

Komplexe Datentypen

Strukturtyp

Beispiele:

```
stud.matrikelnummer = 716347;  
stud.wohnort.postleitzahl = 84036;  
strcpy (stud.name, „Fuchs”);  
strcpy (stud.wohnort.stadt, „Landshut”);  
struct kartesische_koordinaten {  
    float x;  
    float y;  
} punkt1;  
struct kartesische_koordinaten punkt2, punkt3;  
struct {  
    float x;  
    float y;  
} punkt1, punkt2, punkt3;
```

Komplexe Datentypen

Strukturtyp

Zuweisung:

Liegen zwei Strukturvariablen a und b vom gleichen Strukturtyp vor, so kann der Wert der einen Variablen der anderen zugewiesen werden, z.B. durch

```
a = b;
```

Bei dieser Zuweisung werden alle Komponentenwerte der Variablen b den jeweiligen Komponenten der Variablen a zugewiesen.

Größenberechnung:

Die Größe einer Strukturvariablen im Arbeitsspeicher kann nicht aus der Größe der einzelnen Komponenten berechnet werden, da Compiler die Komponenten oft auf bestimmte Wortgrenzen (Alignment) legen. Zur Ermittlung der Größe muss der Operator `sizeof` verwendet werden, z.B.

```
sizeof stud oder sizeof (struct student).
```

Komplexe Datentypen

Strukturtyp

Vergleich:

Zwei Strukturvariablen a und b können nicht durch den eingebauten Vergleichsoperator == verglichen werden. Ein Vergleich muss immer komponentenweise erfolgen!

Beispiel:

```
struct kartesische_koordinaten{
    float x;
    float y;
}

struct kartesische_koordinaten punkt1 = {1,0};
struct kartesische_koordinaten punkt2 = punkt1;
if (punkt1 == punkt2) { ... }
```

Komplexe Datentypen

Strukturtyp

Beispiel (cont):

```
if (punkt1.x == punkt2.x && punkt1.y == punkt2.y) {
...
}
```

Es wird empfohlen, sich für jeden Strukturtyp eine eigene Gleichheitsfunktion zu definieren!

Strukturen und Funktionen:

Strukturen werden als zusammengesetzte Variablen komplett an Funktionen übergeben. Es gibt hier keinen Unterschied zu Variablen von einfachen Datentypen wie float oder int. Man muss nur einen formalen Parameter vom Typ der Struktur einführen und als aktuellen Parameter eine Strukturvariable dieses Typs übergeben. Auch die Rückgabe einer Strukturvariablen unterscheidet sich nicht von der Rückgabe einer einfachen Variablen.

Komplexe Datentypen

Strukturtyp

Beispiel (cont):

```
boolean equal_kk(struct kartesische_koordinaten punkt1,
                  struct kartesische_koordinaten punkt2) {
    return punkt1.x == punkt2.x && punkt1.y == punkt2.y;
}
```

Pointer und Strukturen:

Die Adresse einer Strukturvariablen `a` wird wie bei Variablen von einfachen Datentypen mit Hilfe des Adressoperators ermittelt, d.h. durch `&a`. Im Folgenden werde ein Pointer `pointer_auf_punkt` auf Variablen des Datentyps `struct kartesische_koordinate` definiert:

```
struct kartesische_koordinate *pointer_auf_punkt;
```

Komplexe Datentypen

Strukturtyp

Dieser Pointer `pointer_auf_punkt` soll nun auf den Punkt `punkt1` zeigen. Dies wird erreicht, indem dem Pointer `pointer_auf_punkt` die Adresse `&punkt1` zugewiesen wird:

```
pointer_auf_punkt = &punkt1;
```

Dann kann auf die Komponenten des Punktes `punkt1` zugegriffen werden über:

```
punkt1.x bzw. (*pointer_auf_punkt).x
```

```
punkt1.y bzw. (*pointer_auf_punkt).y
```

Da Strukturen oft zusammen mit Pointern verwendet werden gibt es hierfür folgende Abkürzung:

```
(*pointer_auf_punkt).x entspricht pointer_auf_punkt->x
```

Komplexe Datentypen

Strukturtyp

Anwendungen:

Strukturen werden verwendet, um logisch zusammengehörende Daten „am Stück“ weiterzugeben. Viele Systemfunktionen liefern logisch zusammengehörige Daten in Form von Strukturen oder Pointern auf Strukturen zurück. Die benötigten Daten erhält man dann durch Zugriff auf die einzelnen Komponenten der Struktur.

Die Systemzeit eines Computers wird in der Regel als Anzahl der Sekunden seit Mitternacht des 1. Januars 1970 ausgegeben. Da sich aus der Anzahl der Sekunden sowohl die Uhrzeit, das Datum und weitere Zusatzinformationen ableiten lassen, müssten mehrere Funktionen geschrieben werden, die jeweils die gesuchte Information in einem bestimmten Format liefern. Um dies zu vermeiden, nützt man Strukturen, um dort alle zusammengehörigen Informationen zu speichern. Folgende Struktur tm (siehe time.h) enthält alle wichtigen Daten, die sich aus der Sekundenanzahl berechnen lassen:

Komplexe Datentypen

Strukturtyp

```
struct tm {  
    int tm_sec;          /* Sekunden - [0,59] */  
    int tm_min;          /* Minuten - [0,59] */  
    int tm_hour;         /* Stunden - [0,23] */  
    int tm_mday;         /* Tag - [1,31] */  
    int tm_mon;          /* Monat - [0,11] */  
    int tm_year;         /* Jahre seit 1900 */  
    int tm_wday;         /* Tage seit Sonntag - [0,6] */  
    int tm_yday;         /* Tage seit 1. Januar -[0,365] */  
    int tm_isdst;        /* Daylight Saving Time */  
};
```

Komplexe Datentypen

Vereinigungstyp

Ein Vereinigungstyp wird in C **Union** genannt und besteht wie eine Struktur aus einer Reihe von Komponenten mit unterschiedlichen Datentypen. Im Gegensatz zur Struktur werden bei einer Union die Komponenten nicht hintereinander im Speicher abgebildet, sondern alle Alternativen beginnen bei derselben Adresse. Der Speicherplatz wird vom Compiler so groß angelegt, dass der Speicherplatz auch für die größte Alternative reicht.

Syntax:

```
union name {  
    komponententyp_1 komponente_1;  
    komponententyp_2 komponente_2;  
    . . .  
    komponententyp_n komponente_n;  
};
```

171

Komplexe Datentypen

Vereinigungstyp

Der hier selbst definierte Typname ist `union name`. Ist die Union definiert, dann kann eine Variable von diesem neuen Typ folgendermaßen definieren:

```
union name variable;
```

Beispiel:

```
union vario {  
    char charnam;  
    int intnam;  
    float floatnam;  
};  
union vario variante;
```

172

Komplexe Datentypen

Vereinigungstyp

Initialisierung:

Bei einer Union kann nur eine Initialisierung der ersten Alternative erfolgen. Diese Initialisierung erfolgt durch einen in geschweiften Klammern stehenden konstanten Ausdruck.

Beispiel:

```
union vario variante = {'a'};
```

Selektion:

Die Selektion erfolgt wie bei Strukturen durch den Punktoperator. Ist eine Variable `x` vom Typ `union name` definiert, dann kann folgendermaßen auf die einzelnen Varianten der Union zugegriffen werden:

```
x.komponente_i
```

Komplexe Datentypen

Vereinigungstyp

Beispiel:

```
printf("Zeichen: %c\n", variante.charnam);
```

Achtung:

Wird in C eine Union definiert, dann steht nicht automatisch ein Variantentest zur Verfügung. Man kann also vor der Selektion nicht prüfen welche Variante vorliegt!

Konstruktion:

Wie bei den Strukturen gibt es auch bei den Unions keinen vorgegebenen Konstruktor. Die Konstruktion einer Variable erfolgt einzelnen über die Selektoren:

```
x.komponente_i = ausdruck;
```

Komplexe Datentypen

Vereinigungstyp

Zuweisung:

Liegen zwei Variablen a und b vom gleichen Vereinigungstyp vor, so kann der Wert der einen Variablen der anderen zugewiesen werden, z.B. durch

```
a = b;
```

Größenberechnung:

Auch hier kann die Größe einer Unionvariable im Arbeitsspeicher durch den Operator sizeof berechnet werden, z.B.

```
sizeof variante oder sizeof (union vario).
```

Vergleich:

Zwei Unionvariablen a und b können wie Strukturvariable nicht durch den eingebauten Vergleichsoperator == verglichen werden, man muss auch hier einen Komponentenvergleich durchführen.

Komplexe Datentypen

Vereinigungstyp

Pointer und Unions:

Die Adresse einer Unionvariablen a wird wie bei Variablen von einfachen Datentypen mit Hilfe des Adressoperators ermittelt, d.h. durch &a. Auch hier steht, wie bei Strukturen der Pfeiloperator als Abkürzung zur Verfügung.

Beispiel:

```
union vario * ptr;  
  
ptr = &variant;  
  
x = ptr->intnam;
```

Komplexe Datentypen

Vereinigungstyp

Variantentest:

Bei C fehlt der implizite Variantentest. Dieser kann und sollte auch immer explizit modelliert werden. Dies erfolgt durch Einbetten der Union in einen speziellen Strukturtyp, wie folgendes Beispiel zeigt:

Beispiel:

```
enum varart {CHARVAR, INTVAR, FLOATVAR};  
struct vario {  
    enum varart             art;  
    union {  
        char charnam;  
        int intnam;  
        float floatnam;  
    }                         inhalt;  
};
```

177

Komplexe Datentypen

Vereinigungstyp

Variantentest (cont):

Bei der Initialisierung und Zuweisung muss dann immer explizit die richtige Variante gesetzt werden. Bei der Selektion kann dann die vorliegende Variante abgeprüft werden.

Beispiel:

```
struct vario variante = { CHARVAR, 'a');  
...  
if (variante.art == CHARVAR) {  
    printf("%c\n", variante.inhalt.charnam);  
}  
...  
variante.inhalt.intnam = 1;  
variante.art = INTVAR;
```

178

Komplexe Datentypen

Bitfelder

Bitfelder sind eine Möglichkeit eine kleine Anzahl von Daten kompakt zu speichern. Hierbei wird neben dem Datentyp auch noch die Länge in Bits explizit angegeben.

Die Datentypen, die in einem Bitfeld verwendet werden dürfen, sind eingeschränkt. Nach dem Standard dürfen lediglich die Typen int, signed int oder unsigned int verwendet werden. Bei manchen Compilern wie z.B. beim Visual C++ Compiler sind auch die Typen char, short und long jeweils signed und unsigned erlaubt.

Bitfelder können nur in Zusammenhang **mit Strukturen** verwendet werden.

Syntax:

```
type var:zahl;
```

Komplexe Datentypen

Bitfelder



Bitfeld der Größe 4
unsigned a : 4; Wertebereich: 0 bis 15

a = 3;

0	0	1	1
---	---	---	---

 a ist 3

a = 19;

0	0	1	1
---	---	---	---

 a ist 3 (Bereichsüberschreitung)

Stellenwert: 2³ 2² 2¹ 2⁰



Bitfeld der Größe 4
signed b : 4; Wertebereich: -8 bis +7

b = 3;

0	0	1	1
---	---	---	---

 b ist 3

b = 9;

1	0	0	1
---	---	---	---

 b ist -7 (Bereichsüberschreitung)

Stellenwert: -2³ +2² +2¹ +2⁰

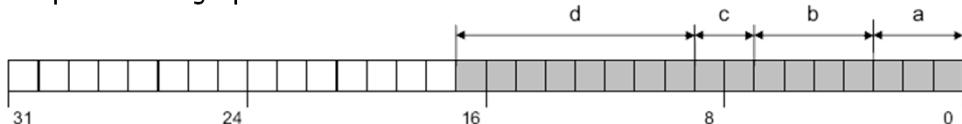
Komplexe Datentypen

Bitfelder

Beispiel:

```
struct Bitfeld_Struktur_1 {  
    unsigned a:3;  
    unsigned b:4;  
    unsigned c:2;  
    unsigned d:8;  
};
```

Ein Bitfeld wird wie eine normale Komponente einer Struktur mit dem Punkt-Operator . angesprochen.



181

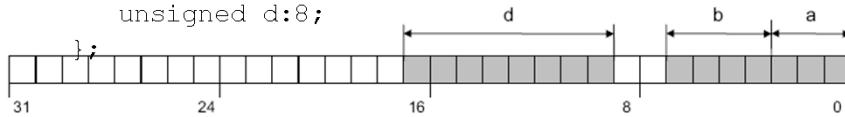
Komplexe Datentypen

Bitfelder

Es ist auch erlaubt, ein Bitfeld ohne Namen einzuführen. Dies führt zu einer Belegung der entsprechenden Bits, ohne dass man diese ansprechen kann. Dies wird oft im Zusammenhang mit der Abbildung von **Hardware-Registern** benutzt, bei denen in vielen Fällen solche ungenutzten Bits auftauchen.

Beispiel:

```
struct Bitfeld_Struktur_1 {  
    unsigned a:3;  
    unsigned b:4;  
    unsigned :2;  
    unsigned d:8;  
};
```



182

Komplexe Datentypen

Bitfelder

Durch den ISO-Standard – der zur Speicherbelegung von Bitfeldern nichts aussagt – ist es dem Compiler-Hersteller überlassen, wie der Speicher von mehreren aufeinanderfolgenden Bitfeldern belegt wird. So ist es z.B. möglich, dass zwei Bitfelder in demselben Byte, Wort oder Langwort untergebracht werden. Somit ist aber ein Ansprechen des Bitfeldes über einen Pointer in der Regel unmöglich, da über Adressen nur ganze Speicherzellen angesprochen werden können.

Anwendung:

Bitfelder werden oft dazu verwendet um Boolesche Informationen kompakt abzulegen. So wird für eine Binäre Information lediglich 1 Bit statt üblicherweise 4 Byte benötigt.

Komplexe Datentypen

Bitfelder

Beispiel:

```
struct bitfeld_8 {  
    unsigned int bit0 : 1;  
    unsigned int bit1 : 1;  
    unsigned int bit2 : 1;  
    unsigned int bit3 : 1;  
    unsigned int bit4 : 1;  
    unsigned int bit5 : 1;  
    unsigned int bit6 : 1;  
    unsigned int bit7 : 1; } bitfeld;  
bitfeld.bit0 = 0;  
bitfeld.bit1 = 1;  
bitfeld.bit2 = 1;  
...
```

Komplexe Datentypen

Vereinbarung eigener Typnamen

Eigene Typnamen als Aliasnamen zu bestehenden Datentypnamen können in C mit Hilfe von **typedef** vereinbart werden. Ein Aliasname ist ein zweiter gültiger Name. Eigene Typnamen (Aliasnamen) sind besonders bei zusammengesetzten Datentypen nützlich – es funktioniert aber auch für einfache Datentypen.

Syntax:

```
typedef existierender_typ neuer_typ;
```

Beispiel:

```
typedef int integer;
integer len, maxLen;
integer * array[8];
```

Der Typname `integer` ist synonym zu `int`. Der Typ `integer` kann dann bei Vereinbarungen, Parametern usw. genauso verwendet werden wie der Typ `int`.

Komplexe Datentypen

Vereinbarung eigener Typnamen

Es ist auch möglich, auf einmal sowohl einen neuen Datentyp als auch einen zusätzlichen Typnamen einzuführen.

Beispiel:

```
typedef struct point {
    int x;
    int y;
} punkt;
```

Innerhalb der geschweiften Klammern wird der neue Datentyp definiert. Sein Typname ist `struct point`. Der zusätzliche Typname ist `punkt`. Damit kann man Punkte sowohl über

```
struct point p1;
```

als auch kürzer über

```
punkt p2;
```

definieren.

Komplexe Datentypen

Vereinbarung eigener Typnamen

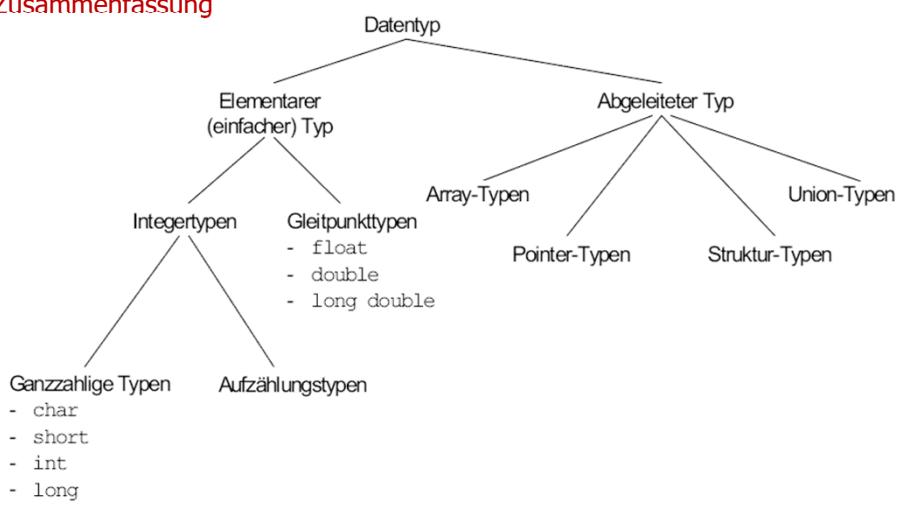
Beispiel:

```
struct student {  
    int matrikelnummer;  
    char name [20];  
    char vorname [20];  
    struct adresse wohnort;  
};  
typedef struct student STUDENT;  
STUDENT semester [50]; /* Vektor mit 50 Strukturen */
```

187

Komplexe Datentypen

Zusammenfassung



188

Der Präprozessor

Aufgaben

Beim Aufruf eines C-Compilers wird vor den eigentlichen Compiler-Läufen wie Syntaxprüfung sowie der Codegenerierung der **Präprozessor** (englisch Preprocessor) gestartet. Die wichtigsten Aufgaben des Präprozessors sind:

- Einfügen von Dateien
- Ersetzen von Text
- bedingte Kompilierung

Die allgemeine Syntax für einen Präprozessor-Befehl (Präprozessor-Direktive, Präprozessor-Anweisung) lautet:

```
#Direktive Text
```

Achtung: Der Präprozessor-Befehl endet mit dem Zeilenende-Zeichen, während eine normale C-Anweisung durch einen Strichpunkt abgeschlossen wird.

189

Der Präprozessor

Symbolische Konstanten und Makros mit Parametern

Mit der Präprozessor-Direktive define wird das Ersetzen von Text festgelegt:

```
#define Bezeichner Ersatztext
```

Der String Bezeichner wird infolge dieser Direktive in der kompletten Datei durch die beliebige Folge von Zeichen in Ersatztext ersetzt. Der Ersatztext ist der Text bis zum Zeilenende. Das Ersetzen von Text findet während des Präprozessor-Laufs ab der Stelle der define-Direktive bis zum Dateiende statt. Ersetzt werden nur komplett Namen.

Nicht ersetzt wird:

- Text, der als Substring in einem Namen enthalten ist
- Text innerhalb von Zeichenketten
- Text innerhalb einer anderen Präprozessor-Anweisung

190

Der Präprozessor

Symbolische Konstanten und Makros mit Parametern

Generell wird das, was durch define definiert wird, als **Makro** bezeichnet. Einem Makro können wie einer Funktion **Parameter** übergeben werden. Ein Makro ohne Parameter wird auch als **symbolische Konstante** bezeichnet. So ist in

```
#define PI 3.1415
```

PI eine symbolische Konstante, die Zahl 3.1415 eine literale Konstante. Es hat sich eingebürgert, die Namen von symbolischen Konstanten stets groß zu schreiben.

Durch die Präprozessor-Direktive

```
#undef Bezeichner
```

kann eine zuvor durch define eingeführte Makrodefinition wieder aufgehoben werden.

191

Der Präprozessor

Symbolische Konstanten und Makros mit Parametern

Makros mit Parametern:

Um einem Makro einen oder auch mehrere Parameter übergeben zu können, muss eine Parameterliste nach dem Bezeichner in folgender Form aufgeführt werden:

```
#define Bezeichner(Param1, ..., ParamN) Ersatztext
```

Wichtig ist dabei, dass zwischen dem Bezeichner und der öffnenden Klammer kein Whitespace-Zeichen stehen darf, sonst wird die Parameterliste als Ersatztext interpretiert.

Beispiel:

```
#define square(x) x*x
```

192

Der Präprozessor

Symbolische Konstanten und Makros mit Parametern

Makros mit Parametern (cont):

Das Makro kann ab dem Zeitpunkt der Definition (mit ein paar Ausnahmen) wie eine Funktion verwendet werden:

```
y = square(x);  
if (square(x) <= 16) { ...}
```

Nach dem Präprozessorlauf sieht die Datei dann folgendermaßen aus:

```
y = x*x;  
if (x*x <= 16) { ...}
```

Der Vorteil liegt darin, dass die Geschwindigkeit gegenüber einem Funktionsaufruf höher ist, weil keine Parameter übergeben werden müssen.

Nachteil: Der Code wird aufgeblasen.

Der Präprozessor

Symbolische Konstanten und Makros mit Parametern

Makros mit Parametern (cont):

Bei dem Ersetzen von Parametern können aber einige Komplikationen auftreten, z.B.:

- ungewollte Bindungen in einem Ausdruck
- ungewollte Blockstruktur

Lösung:

Vollständige Klammerung verwenden!

Beispiel:

```
square(3 + x) wird ersetzt durch 3 + x * 3 + x  
Stattdessen #define square(x) ((x) * (x))
```

Der Präprozessor

Bedingte Kompilierung

Unter **bedingter Kompilierung** versteht man, dass zur Kompilierzeit anhand von Ausdrücken und Symbolen entschieden wird, welcher Teil des Source-Codes kompiliert werden soll. Diese Aufgabe übernimmt der Präprozessor in der Weise, dass nicht in Frage kommender Source-Code entfernt wird. Folgende Direktiven stehen zur Verfügung:

```
#if konstanter_Ausdruck
#elif konstanter_Ausdruck
#else
#endif
#ifndef Symbol
#endif
```

Der Präprozessor

Bedingte Kompilierung

Die bedingte Kompilierung wird z.B. für Programme benutzt, die auf unterschiedlichen Betriebssystemen bzw. Prozessoren laufen sollen oder wenn man eine Testversion des Programms generieren will.

Die Präprozessor-Direktiven `if`, `elif`, `else`, `endif` entsprechen einer if-else-Anweisung in C. `elif` und `else` sind dabei optional. `if` und `elif` müssen einen konstanten Ausdruck als Bedingung erhalten.

Der Wert des **konstanten Ausdrucks** wird wie in der if-Anweisung in C als true oder false interpretiert (false ist 0, true ist ungleich 0). Der Präprozessor **entfernt** alle Programmteile, die in einem Zweig enthalten sind, der als false interpretiert wird.

Mit den Direktiven `ifdef` bzw. `ifndef` kann geprüft werden, ob ein bestimmtes Makro definiert wurde oder nicht.

Der Präprozessor Bedingte Kompilierung

Beispiel:

```
#define TESTVERSION 1

#if TESTVERSION
    printf ("Testausgabe: Index i = %d\n", i);
#endif
#if INT_MAX > 32767
    int i;
#else
    long i;
#endif

#ifdef TESTVERSION
    printf ("Testausgabe: Index i = %d\n", i);
#endif
```

197

Der Präprozessor Einfügen von Dateien in den Source-Code

Das Einfügen einer beliebigen Datei kann mit Hilfe der Präprozessor-Direktive

```
#include "filename"
```

oder

```
#include <filename>
```

erfolgen. Beim Einfügen der Datei wird die Zeile mit der Präprozessor-Direktive entfernt und der Quelltext der Datei eingefügt. Befindet sich innerhalb der eingefügten Datei ebenfalls eine include-Anweisung, so wird diese ebenfalls ausgeführt. Das Einfügen wird dabei in einer temporären, nur für den Kompilierlauf angelegten Datei durchgeführt. Die ursprüngliche Datei bleibt erhalten.

198

Der Präprozessor

Einfügen von Dateien in den Source-Code

Das Einfügen von Dateien wird meistens verwendet, um Konstanten (define-Anweisung), Deklarationen von Typen und Funktionen, Makros und gegebenenfalls weitere include-Direktiven in einen Quelltext einzubinden. Als Beispiel kann hier die Datei stdio.h genommen werden. Unter anderem enthält die Datei stdio.h:

- Definitionen von Typnamen wie `size_t`, `FILE`, ...
- Konstantendefinitionen für die Konstanten `EOF`, `stdin`, `stdout`, ...
- Funktionsdeklarationen für die Funktionen `printf()`, `scanf()`, ...
- und Makros wie z.B. `getc()`, `putc()`, ...

Der Präprozessor

Einfügen von Dateien in den Source-Code

Ist der Filename innerhalb der include-Direktive in spitzen Klammern `<>` eingeschlossen, so handelt es sich um eine Systemdatei. Diese sucht der Compiler in den für ihn typischen **Include-Verzeichnissen**. Unter UNIX ist das Standard-Include-Verzeichnis meist

`/usr/include`

und unter dem Dev-C++ GNU Compiler

`\Dev-Cpp\include.`

Die meisten Compiler erlauben auch eine Erweiterung der Standard-Include-Verzeichnisse mit Hilfe von Compiler-Optionen.

Beispiel:

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

Der Präprozessor

Einfügen von Dateien in den Source-Code

Stellt man in der include-Direktive den Filenamen in Anführungszeichen, so wird zuerst in dem aktuellen Arbeitsverzeichnis bzw. in dem in der include-Direktive aufgeführten Verzeichnis gesucht. Wird die entsprechende Include-Datei dort nicht gefunden, so wird die Suche in den Standard-Include-Verzeichnissen fortgesetzt.

Beispiel:

```
#include „bool.h“  
#include „includes/bool.h“
```

Damit können selbst geschriebene Header-Dateien eingebunden werden. Im Sinne der Wiederverwendung sollten Makros, die man immer wieder benötigt in eigene Header-Dateien ausgelagert werden (Software Engineering).

Bit-Operatoren

Logische Bit-Operatoren

Im Gegensatz zu anderen Programmiersprachen besitzt die Programmiersprache C Operatoren zur Bitmanipulation. Mit solchen Operatoren wird eine hardwarenahe Programmierung unterstützt. C kennt folgende vier **logischen Bit-Operatoren**:

- UND-Operator für Bits: &
- ODER-Operator für Bits: |
- Exklusives-ODER-Operator für Bits: ^
- Negationsoperator für Bits: ~

Die logischen Bit-Operatoren dürfen nur für ganzzahlige Datentypen benutzt werden. Bei vorzeichenlosen Datentypen ist die Verwendung der Bitoperatoren problemlos, ansonsten können implementierungsabhängige Aspekte auftreten.

Bit-Operatoren

Logische Bit-Operatoren

UND-Operator für Bits: A & B

Die Operation bitweises UND findet auf allen Bits der Operanden statt. Dabei werden jeweils die Bits der entsprechenden Position folgendermaßen miteinander verknüpft:

Bit n von A	Bit n von B	Bit n von A&B
0	0	0
0	1	0
1	0	0
1	1	1

Beispiele:

```
0 & 1      /* 0 & 1 = 0 */  
14 & 1     /* 1110 & 0001 = 0000 */  
var & var   /* var & var = var */
```

203

Bit-Operatoren

Logische Bit-Operatoren

ODER-Operator für Bits: A | B

Die Operation bitweises ODER findet auf allen Bits der Operanden statt. Dabei werden jeweils die Bits der entsprechenden Position folgendermaßen miteinander verknüpft:

Bit n von A	Bit n von B	Bit n von A B
0	0	0
0	1	1
1	0	1
1	1	1

Beispiele:

```
0 | 1      /* 0 | 1 = 1 */  
14 | 1     /* 1110 | 0001 = 1111 */  
var | 0     /* var | 0 = var */
```

204

Bit-Operatoren

Logische Bit-Operatoren

Exklusives-ODER-Operator für Bits: A ^ B

Die Operation bitweises Exklusives-ODER (Circumflex) findet auf allen Bits der Operanden statt. Dabei werden jeweils die Bits der entsprechenden Position folgendermaßen miteinander verknüpft.

Bit n von A	Bit n von B	Bit n von A^B
0	0	0
0	1	1
1	0	1
1	1	0

Beispiele:

```
var ^ 0      /* var ^ 0 = var */  
14 ^ 3      /* Bit 0 und Bit 1 der Zahl 14 werden  
              invertiert: 1110 ^ 0011 = 1101 */
```

Bit-Operatoren

Logische Bit-Operatoren

Negationsoperator für Bits: ~A

Die Operation bitweise Negation findet folgendermaßen auf allen Bits des Operanden statt:

Bit n von A	Bit n von ~A
0	1
1	0

Beispiele:

```
unsigned char a, b;  
a = 9; /* a = 0000 1001 */  
b = ~a; /* b = 1111 0110 */
```

Bit-Operatoren

Logische Bit-Operatoren

Flags

Mit **Flag** wird eine binäre Variable bezeichnet, welcher als Hilfsmittel zur Kennzeichnung bestimmter Zustände benutzt werden kann. Ein Flag kann gesetzt, gelöscht oder gelesen werden. Flags werden in Variablen vom Typ `unsigned x` zusammengefasst. Die logischen Bit-Operatoren können dann im Zusammenhang mit sog. **Masken** zum Zugriff auf die einzelnen Flags verwendet werden.

Setzen eines Flags:

Mit Hilfe des logischen ODER-Operators können Flags gesetzt werden. Alle Bits, welche in der Bitmaske '1' sind, werden auf '1' gesetzt. Alle Bits, die in der Maske auf '0' gesetzt sind, bleiben unverändert.

207

Bit-Operatoren

Logische Bit-Operatoren

Beispiel:

```
unsigned char a=0;  
a |= 8;           /* 0000 1000 setzt das 4 Bit */
```

Löschen eines Flags:

Mit Hilfe des logischen UND-Operators können einzelne Flags gelöscht werden. Alle Bits, welche in der Bitmaske '0' sind, werden auf '0' gesetzt. Alle Bits, die in der Maske auf '1' gesetzt sind, bleiben unverändert.

Beispiel:

```
unsigned char a=255;  
a &= 247;        /* 1111 0111 setzt das 4. Bit auf 0 */
```

208

Bit-Operatoren

Logische Bit-Operatoren

Lesen eines Flags:

Mit Hilfe des bitweisen UND-Operators können einzelne Flags auch gelesen werden. Die Bitmaske muss an den Stellen der zu prüfenden Bits eine '1' haben, an allen anderen eine '0'. Die nicht interessanten Bits werden dabei **maskiert** (ausgeblendet).

Beispiel:

```
unsigned char a=255;  
if (a & 8) ... /* 0000 1000 maskiert alle Bits  
                  bis auf das 4. Bit */
```

209

Bit-Operatoren

Logische Bit-Operatoren

Invertieren von Flags:

Mit Hilfe des logischen XOR-Operators können einzelne Flags invertiert werden (toggeln). Alle Bits, welche in der Bitmaske '1' sind, werden invertiert. Alle Bits, die in der Maske auf '0' gesetzt sind, bleiben unverändert.

Beispiel:

```
unsigned char a=255;  
a ^= 9;          /* 0000 1001 invertiert Bit 1  
                  und Bit 4 */
```

210

Bit-Operatoren Shift-Operatoren

Mit den Shift-Operatoren werden Bits nach links << oder nach rechts >> verschoben (engl. shift). Es darf nur um ganzzahlige positive Stellen von Bits verschoben werden. Shift-Operationen wirken wie **Multiplikationen** bzw. **Divisionen** mit Potenzen von 2.

Rechtsshift-Operator: A >> B

Mit dem Rechtsshift-Operator A >> B werden die Bits von A um B Bitstellen nach rechts geschoben. Dabei gehen die B niedrigwertigen Bits von A verloren. Wenn der Operand A von einem unsigned-Typ ist, werden die höherwertigen Bits mit Nullen aufgefüllt, was einer **Division durch 2^B** entspricht. Bei einem Operanden A eines signed-Typs mit einem negativen Wert ist das Ergebnis vom Compiler abhängig – beim Visual C++ Compiler bleibt das Vorzeichenbit erhalten.

Bit-Operatoren Shift-Operatoren

Beispiele:

```
unsigned char a;  
a = 8;           /* 0000 1000 Bitmuster von 8 */  
a = a >> 3;     /* 0000 0001 Bitmuster von 1 */
```

Die Verschiebung um 3 Bits nach rechts entspricht einer Division durch 2^3 .

Linksshift-Operator: A << B

Bei dem Linksshift-Operator A << B werden die Bits von A um B Bitstellen nach links geschoben. Dabei gehen die B höherwertigen Bits von A verloren. Die nachrückenden niedrigwertigen Bits werden mit Nullen aufgefüllt. Falls kein Überlauf eintritt entspricht dies bei einem unsigned-Operanden einer Multiplikation mit 2^B .

Bit-Operatoren Shift-Operatoren

Beispiele:

```
unsigned char a = 128; /* 1000 0000 Bitmuster von 128 */  
a = a << 1; /* Overflow. Ergebnis: 0000 0000 */  
a = 8; /* 0000 1000 Bitmuster von 8 */  
a = a << 3; /* 0100 0000 Bitmuster von 64 */
```

Die Verschiebung um 3 Bits nach rechts entspricht einer Multiplikation mit 2^3 .

Dynamische Speicherzuweisung Statische und dynamische Variablen

Prinzipiell unterscheidet man bei Programmiersprachen zwischen **statischen und dynamischen Variablen**.

Eine **statische Variable** ist eine Variable, die in einem Programm vereinbart wird. Dabei müssen der Name und der Typ der Variablen angegeben werden wie in folgendem Beispiel:

```
int x;
```

Bei der Vereinbarung erhält die Variable einen Bezeichner, einen Variablenamen, hier den Variablenamen x. Nach der Vereinbarung kann auf die Variable über ihren Namen zugegriffen werden. Eine solche Variable heißt statisch, weil ihr Gültigkeitsbereich und ihre Lebensdauer durch die statische Struktur des Programms festgelegt ist.

Dynamische Speicherzuweisung

Statische und dynamische Variablen

Dynamische Variablen sind anonyme Variablen. Sie erscheinen nicht explizit in einer Variablenvereinbarung und tragen keinen Namen. Daher kann auf sie nicht über einen Bezeichner zugegriffen werden. Dynamische Variablen werden bei Bedarf zur Laufzeit des Programms angelegt.

Die Gültigkeit und Lebensdauer einer dynamischen Variablen wird nicht durch die Blockgrenzen, d.h. nicht durch die statische Struktur des Programms bestimmt. Sie werden, im Gegensatz zu statischen Variablen im sog. **Heap** abgelegt.

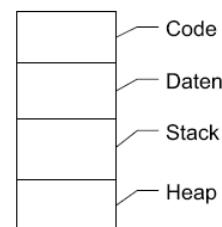
Dynamische Speicherzuweisung

Adressraum

Der **Adressraum** eines ablauffähigen C-Programms – also einer Datei programm.exe – besteht aus den vier Segmenten (Bereichen):

- Code (Programmtext),
- Daten
- Stack
- Heap.

Im Codesegment liegt das Programm in Maschinencode. Lokale (statische) Variablen werden vom C-Compiler auf dem Stack angelegt, globale (statische) Variablen im Datensegment und dynamische Variablen auf dem Heap. Auf dem Stack werden ferner die Rücksprungadresse und die Parameter einer Funktion abgelegt.



Dynamische Speicherzuweisung Reservierung von Speicher

In C erfolgt die Reservierung von Speicher im Heap mit der Funktion `malloc()`. Der Name `malloc()` kommt von memory allocation.

Syntax:

```
#include <stdlib.h>
void * malloc (size_t size);
```

Als Parameter erhält die Funktion `malloc()` wie viele Bytes reserviert werden sollen. Da der Programmierer oft selbst nicht weiß, wie viele Bytes eine Variable eines bestimmten Typs umfasst – und dies auf einem anderen Rechner wieder anders sein könnte – sollte er die Zahl der Bytes mit Hilfe des `sizeof`-Operators bestimmen.

Die Funktion `malloc()` gibt einen Pointer auf den reservierten Speicherbereich zurück, wenn die Speicherreservierung durchgeführt werden konnte, ansonsten wird `NULL` zurückgegeben.

Dynamische Speicherzuweisung Reservierung von Speicher

Der Pointer ist vom Typ `void *` und wird bei einer Zuweisung **implizit** in den gewünschten Typ gewandelt.

Beispiel:

```
int * pointer;
if ((pointer = malloc (sizeof (int))) != NULL) {
    *pointer = 3;
    printf("pointer zeigt auf eine Zahl mit Wert %d\n",
           *pointer);
}
else {
    printf ("Nicht genuegend Speicher verfuegbar\n");
}
```

Dynamische Speicherzuweisung Reservierung von Speicher

Eine weitere Funktion zur Reservierung von Speicher ist die Funktion `calloc()`.

Syntax:

```
#include <stdlib.h>
void * calloc (size_t num, size_t size);
```

Als Übergabeparameter erwartet die Funktion `calloc()` zwei Parameter. Als erster Parameter wird die Anzahl der benötigten Variablen erwartet. Der zweite Parameter entspricht der Größe einer einzelnen Variablen in Byte. Der Rückgabewert entspricht dem von `malloc()`.

Beispiel:

```
int * pointer;
if ((pointer = calloc (1, sizeof (int))) != NULL) {
    *pointer = 3;
    ...
```

219

Dynamische Speicherzuweisung Reservierung von Speicher

Um den mit `malloc()` bzw. `calloc()` erzeugten dynamischen Speicherbereich auch nachträglich noch in seiner Größe ändern zu können, existiert die Funktion `realloc()`.

Syntax:

```
#include <stdlib.h>
void * realloc (void * memblock, size_t size);
```

Diese Funktion bekommt einen Pointer auf einen bereits existierenden dynamischen Speicherbereich übergeben und zusätzlich noch die Größe des gewünschten neuen Speicherbereiches. Kann der angeforderte Speicher nicht mehr an der bisherigen Adresse angelegt werden, weil kein ausreichend großer zusammenhängender Speicherbereich mehr frei ist, dann verschiebt `realloc()` den vorhandenen Speicherbereich an eine Stelle im Speicher, an der noch genügend Speicher frei ist.

220

Dynamische Speicherzuweisung

Freigeben von Speicher

In C muss man den nicht mehr benötigten Speicher, der mit `malloc()`, `calloc()` oder `realloc()` beschafft wurde, mit Hilfe der Funktion `free()` freigeben. Der entsprechende Speicherbereich kann dann von der Heapverwaltung erneut vergeben werden.

Syntax:

```
#include <stdlib.h>
void free (void * pointer);
```

Der formale Parameter von `free()` ist vom Typ `void *`. Damit kann ein Pointer auf einen beliebigen Datentyp übergeben werden. Die Funktion ist mit dem Pointer, den `malloc()`, `calloc()` oder `realloc()` geliefert hat, aufzurufen, ansonsten ist das Ergebnis undefiniert. Nach dem Aufruf von `free()` darf über diesen Pointer natürlich nicht mehr auf ein Speicherobjekt zugegriffen werden, da er auf kein gültiges Speicherobjekt mehr zeigt.

Dynamische Speicherzuweisung

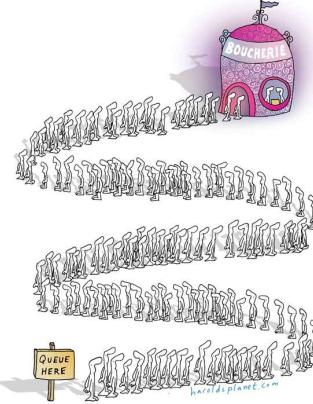
Freigeben von Speicher

- Wird vergessen nicht mehr benötigter Speicher mittels `free` freizugeben, spricht man von einem **memory leak**. Das Programm braucht dann je länger es läuft oft immer mehr Speicher und wird dadurch oft langsamer und stürzt u.U. ganz ab, weil kein Speicher mehr vorhanden ist.
- Verwendet man Zeiger auf bereits freigegebenen Speicher, spricht man von sogenannten **dangling pointers**. Mögliche Folgen:
 - Werte ändern sich scheinbar aus dem Nichts und unerwartet
 - Das Programm stürzt plötzlich mit einem „segmentation fault“ ab
 - Oft manifestiert sich der Fehler zeitlich viel später und ist daher im Programm oft sehr schwer zu finden
- Tipp: Nach einem `free(p)`, sofort `p = NULL` setzen (verhindert dangling pointer).

Dynamische Speicherzuweisung

Dynamischer Datentyp Liste

- Synonyme Begriffe: *Sequenz, lineare Liste*
- Linear geordnete Sammlung von Elementen. Lineare Ordnung bezieht sich auf Position in der Liste
- Jedes Listenelement hat einen *Vorgänger*, ein *Nachfolger*
- Liste ansonsten nicht weiter strukturiert
- Listengröße nicht statisch festgelegt, dynamisch nach Bedarf änderbar
- Alle Listenelemente haben denselben Typ



223

Dynamische Speicherzuweisung

Dynamische Datentypen Liste

Vorgehen (informell): Liste ist...

- entweder leer oder
- besteht aus einem ersten Element und einer restlichen Liste.

Definition des Datentyps:

Der Datentyp der Listen wird durch eine rekursive Definition unter Verwendung eines Zeigers definiert:

```
typedef struct list {  
    int x;  
    struct list * next;  
} list;
```

Die leere Liste wird durch den NULL Zeiger repräsentiert.

224

Dynamische Speicherzuweisung

Dynamische Datentypen Liste

Unter Verwendung der Funktion `malloc()` kann jetzt folgendermaßen eine Funktion `append()` definiert werden, die an eine Liste vorne ein Element dranhängt:

```
list * append (list *l, int x) {
    list *nl;
    if ((nl = (list*) malloc(sizeof(list))) != NULL) {
        nl->x = x;
        nl->next = l;
    }
    return nl;
}
```

225

Dynamische Speicherzuweisung

Dynamische Datentypen Liste

Der Datentyp `list` bildet zusammen mit seinen charakteristischen Funktionen die Datenstruktur der Listen. Folgende Funktionen werden üblicherweise durch die Struktur zur Verfügung gestellt:

<code>list* make(int x);</code>	macht einelementige Liste
<code>list* append(list *l, int x)</code>	hängt Element vorne dran
<code>list* stock(list *l, int x)</code>	hängt Element hinten dran
<code>int first(list *l)</code>	liefert erstes Element
<code>list* rest(list *l)</code>	liefert restliche Liste
<code>int last(list *l)</code>	liefert letztes Element
<code>unsigned length(list *l)</code>	liefert Länge der Liste
<code>bool is_in(int x, list *l)</code>	prüft, ob Element in Liste

226

Dynamische Speicherzuweisung

Dynamische Datentypen Liste

```
int first(list *l){  
    return l->x;  
}  
  
list* rest(list *l){  
    return l->next;  
}  
  
unsigned length(list *l) {  
    if (l == NULL) return 0;  
    else return 1+length(rest(l));  
}
```

227

Modulares Programmieren

Praktische Aspekte

- In der Praxis sind Programme meist viele 1000 oder sogar Millionen Programmzeilen
 - Libre Office ca. 12 Millionen Zeilen C++ Code!
 - Schätzungen von Microsoft Office ca. 40 Millionen Zeilen!
- Es ist nicht praktikabel das ganze Programm in einer einzigen Datei abzuspeichern:
 - Jede Änderung würde eine Rekompilierung von Millionen Programmzeilen verursachen
 - Navigation über mehr als 100 Zeilen in einer Datei ist unpraktisch (Ein Editor für 10 Millionen Programmzeilen muss erst noch geschrieben werden)
 - Mehrere Entwickler könnten nicht gleichzeitig an der selben Datei arbeiten

228

Modulares Programmieren

Praktische Aspekte

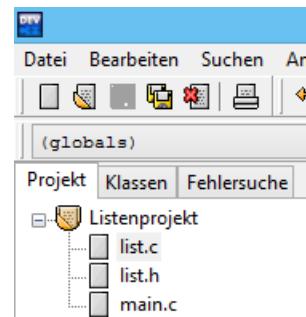
- Eine nützliche Abstraktion ist die Implementierung von der Schnittstelle zu trennen
 - Abstrakter Datentyp
 - Konkrete Implementierung
- Vorteile
 - Verschiedene Implementierungen derselben Funktionalität können ausgetauscht werden
 - Wenn dasselbe Interface (derselbe ADT) verwendet wird, sind die Änderungen minimal, z.B. nur eine andere Bibliothek verwenden
- Beispiel: Datenstruktur Liste
 - Operationen make, append, stock, ...
 - Verschiedene Implementierung sind möglich (z.B. doppelt verkettete Liste)

229

Modulares Programmieren

Header Datei

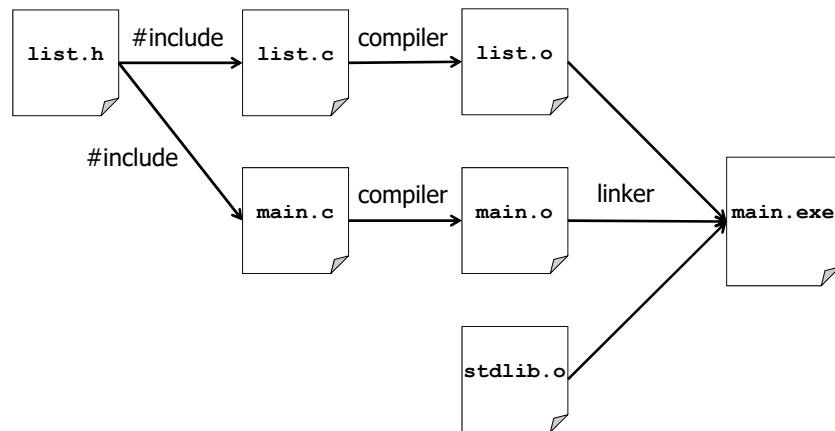
- In C werden die abstrakten Datenstrukturen (Datentyp + Schnittstelle der charakteristischen Funktionen) in einer Header Datei (*.h-Datei) abgelegt.
- Die Implementierung erfolgt in einer C-Datei, die die Headerdatei inkludiert.
- In einer main-Datei werden dann die Funktionen einer abstrakten Datenstruktur verwendet. Dazu muss ebenso die Header Datei eingebunden werden.
- Durch den Linker wird dann die Implementierung des ADTs zum Programm hinzugefügt.



230

Modulares Programmieren

Header Datei



231

Modulares Programmieren

Header Datei

```
/* list.h */
/* Datentyp */
typedef struct list {
    int x;
    struct list *next;
} list;
/* Funktionen */
list* append (int, list *);
int first(list *);
list* rest(list *l);
unsigned length(list *);
```

232

Modulares Programmieren

Header Datei

```
/* list.c */  
#include <stdlib.h>  
#include "list.h"  
int first(list *l){  
    return l->x;  
}  
list* rest(list *l){  
    return l->next;  
}  
unsigned length(list *l) {  
    if (l == NULL) return 0;  
    ...
```

233

Modulares Programmieren

Header Datei

```
/* main.c */  
#include <stdio.h>  
#include <stdlib.h>  
#include "list.h"  
  
int main(int argc, char *argv[]) {  
    list *l;  
    l = append(1,append(2,NULL));  
    printf("%d\n",length(l));  
    return 0;  
}
```

234

Modulares Programmieren

Header Datei

Die Headerdatei sollte vor doppelter Inkludierung geschützt werden:

```
/* list.h */  
ifndef _LIST_H_  
define _LIST_H_  
  
/* Datentyp */  
typedef struct list {  
    int x;  
    struct list *next;  
} list;  
/* Funktionen */  
...
```

235

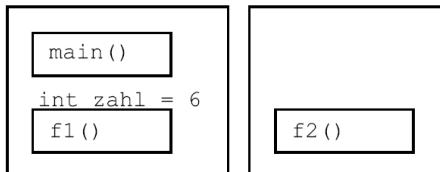
Modulares Programmieren

Globale Variablen

Im Folgenden soll als Beispiel ein Programm aus zwei Dateien betrachtet werden.

Datei ext1.c

Datei ext2.c



Möchte man von der Funktion `f2` auf die globale Variable `zahl` zugreifen, so muss diese in der Datei `ext2.c` folgendermaßen als **extern deklariert** werden:

```
extern int zahl;
```

Dadurch wird die globale Variable in dieser Datei bekanntgemacht.

236

Modulares Programmieren

Globale Variablen/Funktionen

Generell kann ein Programmierer alle globalen Variablen und Funktionen aus einer anderen Datei benutzen, indem er in seiner Datei entsprechende extern-Deklarationen aufnimmt. Um seine Definitionen vor nicht erwünschter Benutzung zu schützen, ist es in C möglich, globale Daten und Funktionen innerhalb einer Datei so zu kapseln, dass diese nur innerhalb der eigenen **Datei** verwendet werden können.

Funktionen und globale Variablen, die als `static` definiert werden, sind nur in ihrer eigenen Datei sichtbar. Funktionen aus anderen Dateien können auf diese Funktionen und globalen Variablen nicht zugreifen.

Damit können mit Hilfe des Schlüsselworts `static` die Prinzipien des **Information Hiding** in C-Programmen realisiert werden.

Speicherklassen

Lokale Variablen

Bei lokalen Variablen gibt es drei verschiedene Speicherklassen:

- Speicherklasse `auto`
- Speicherklasse `register`
- Speicherklasse `static`

Die Speicherklasse wird optional bei der Deklaration einer Variablen vor dem Typ geschrieben:

```
Speicherklasse typ variablenbezeichner;
```

Beispiel:

```
auto int x;  
static short y;  
register int z;
```

Speicherklassen

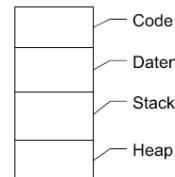
Lokale Variablen

Automatische Variablen

Automatische Variablen sind lokale Variablen **ohne Angaben einer Speicherklasse**, mit Angabe der Speicherklasse `auto`, sowie mit Angabe der Speicherklasse `register`. Auch die formalen Parameter werden zu den automatischen Variablen gezählt.

Automatische Variablen werden so bezeichnet, weil sie automatisch angelegt werden und automatisch verschwinden. Sie sind nur innerhalb des Blockes sichtbar, in dem sie definiert sind, und leben vom Aufruf des Blockes bis zum Ende des Blockes.

Automatische Variablen werden beim Betreten eines Blockes auf dem **Stack** angelegt. Am Ende Blockes wird der Speicherplatz wieder freigegeben.



239

Speicherklassen

Lokale Variablen

Automatische Variablen (cont.)

Eine automatische Variable wird bei jedem Blockeintritt erneut angelegt und initialisiert. Es ist deshalb nicht möglich, mit automatischen Variablen Werte von einem Aufruf des Blockes oder der Funktion bis zum nächsten Aufruf zu retten.

Speicherklasse register

Mit Hilfe des Schlüsselwortes `register` kann man dem Compiler empfehlen, formale Parameter und lokale Variablen in Registern des Prozessors statt im Arbeitsspeicher abzulegen. Damit kann die Zugriffszeit auf eine Variable verkürzt werden. Gut optimierende Compiler erkennen jedoch solche Variablen durch geeignete Analyseverfahren und legen diese Variablen selbstständig – wenn immer möglich – in Registern ab. Daher ist die Verwendung von `register` in den meisten Fällen überflüssig.

240

Speicherklassen

Lokale Variablen

Statische Variablen

Wird also eine lokale Variable mit dem Schlüsselwort `static` versehen, so bezeichnet man sie als **statische Variable**. Statische lokale Variablen werden vom Compiler nicht auf dem Stack angelegt, sondern im Speicherbereich der globalen Variablen (Datenbereich). Sie werden dadurch permanent und behalten ihren Wert auch zwischen zwei Funktionsaufrufen bzw. Betreten eines Blockes bei. Eine manuelle Initialisierung wird nur beim ersten Aufruf ausgeführt. Wird keine manuelle Initialisierung durchgeführt, dann wird automatisch mit 0 initialisiert.

Statische lokale Variablen haben also eine andere Lebensdauer, bezüglich der Sichtbarkeit verhalten sie sich aber nach wie vor wie lokale Variablen. Funktionen können über statische Variable ein „Gedächtnis“ bekommen.

Speicherklassen

Globale Variablen

Wird also eine Variable außerhalb einer Funktion definiert, so bezeichnet man sie als **globale Variable**.

Lebensdauer

Externe Variablen leben so lange wie das ganze Programm. Sie werden zu Programmbeginn im Datenbereich angelegt und werden erst vernichtet, wenn das Programm beendet wird.

Sichtbarkeit

Globale Variablen sind in allen Funktionen sichtbar, die in derselben Quelldatei nach der Variablendefinition definiert werden. Hierbei kann es allerdings zu Verschattungen kommen.

Initialisierung

Globale Variablen werden, wenn sie nicht manuell initialisiert werden, zu Beginn des Programmes automatisch mit 0 initialisiert.

Speicherklassen Zusammenfassung

Speicher-klasse	Gültigkeit	Lebensdauer	automat. Initialisierung	Segment
register	Block	Block	nein	Stack oder in Register
auto	Block	Block	nein	Stack
static lokale Variable	Block	Programm	mit 0	Daten
static externe Variable	in Datei ab Definition, in anderen Dateien nicht	Programm	mit 0	Daten
extern (nicht static)	in Programm ab Definition bzw. ab extern-Deklaration	Programm	mit 0	Daten

Typumwandlung Implizite Typumwandlung

In C ist es nicht notwendig, dass die Operanden eines Ausdrucks vom selben Typ sind. Genauso wenig muss bei einer Zuweisung der Typ übereinstimmen. Auch bei der Übergabe von Werten an Funktionen und bei Rückgabewerten von Funktionen können übergebene Ausdrücke bzw. der rückzugebende Ausdruck von den formalen Parametern bzw. dem Rückgabetyp verschieden sein. In solchen Fällen kann der Compiler **selbsttätig implizite (automatische) Typumwandlungen** durchführen, die nach einem von der Sprache vorgeschriebenen Regelwerk ablaufen.

Arithmetische Konversionen

Bei binären Operatoren werden arithmetische Operanden in einen gemeinsamen Typ umgewandelt. D.h. in

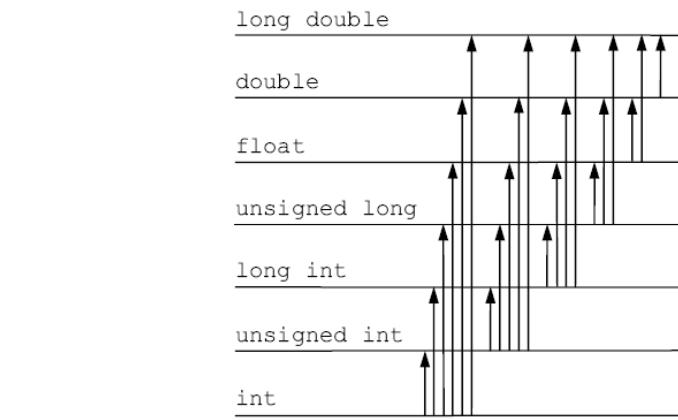
Ausdruck1 Operator Ausdruck2

werden Ausdruck1 und Ausdruck2 auf den gleichen Typ gebracht. Von diesem Typ ist auch das Ergebnis.

Typumwandlung

Implizite Typumwandlung

Die Umwandlung erfolgt in den höheren Typ der folgenden Hierarchie:



245

Typumwandlung

Implizite Typumwandlung

Die Integer-Erweiterung

Mit signed char-, unsigned char-, short- und unsigned short-Werten werden in C keine Verknüpfungen zu Ausdrücken durchgeführt. Sie werden vor der Verknüpfung konvertiert und zwar:

- signed char, unsigned char und short in int,
- unsigned short in unsigned int, falls die Datentypen short und int äquivalent sind,
- unsigned short in int, falls die Datentypen short und int nicht äquivalent sind.

Dieser Vorgang wird als **Integer-Erweiterung** bezeichnet. Dies bedeutet, dass in C immer mit Integer-Werten gerechnet wird, die mindestens den Datentyp int haben.

246

Typumwandlung

Implizite Typumwandlung

Implizite Typumwandlung bei Zuweisung und Funktionsaufrufen

Stimmt der Typ der Variablen links des Zuweisungsoperators = nicht mit dem Typ des Ausdrucks auf der rechten Seite des Zuweisungsoperators überein, so findet eine implizite Konvertierung statt, wenn die Typen links und rechts „verträglich“ sind. Ansonsten wird eine Fehlermeldung generiert.

Verträgliche Typen sind arithmetische Typen bzw. Zeiger auf void.

Bei der Zuweisung wird der rechte Operand in den Typ des linken Operanden umgewandelt, d.h. der Resultattyp einer Zuweisung ist der des linken Operanden. Die Umwandlung kann dabei zwischen beliebigen verträglichen Typen erfolgen, d.h. auch von einem „größerem“ Typ zu einem „kleinerem“.

Aktuelle Parameter werden in den Typ des formalen Parameters wie im Falle einer Zuweisung gewandelt. Ebenso wird der return-Wert in den Rückgabetyp wie bei einer Zuweisung gewandelt.

Typumwandlung

Explizite Typumwandlung

Implizite Typumwandlungen sind sehr gefährlich, da man sie oft nicht richtig einschätzt. Sorgen Sie am besten selbst dafür, dass die Typen rechts und links des Zuweisungsoperators übereinstimmen.

Eine explizite Typumwandlung eines beliebigen Ausdrucks kann man mit dem **cast-Operator** (Typkonvertierungs-Operator) durchführen. Das englische Wort cast heißt u.a. „in eine Form gießen“. Durch

(Typname) Ausdruck

wird der Wert des Ausdrucks in den Typ gewandelt, der in der Klammer angegeben ist.

Typumwandlung

Explizite Typumwandlung

Es kann nicht jeder Typ in einen beliebigen anderen Typ gewandelt werden. Möglich sind Wandlungen zwischen skalaren Typen und von einem skalaren Typ in den Typ void:

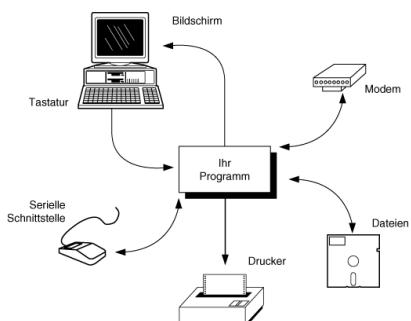
- Wandlungen zwischen Integer-Typen
- Wandlungen zwischen Gleitpunkt-Typen
- Wandlungen zwischen Integer- und Gleitpunkt-Typen
- Wandlungen zwischen Pointern auf Variablen
- Wandlungen zwischen Pointern und Integer-Typen
- die Wandlung eines Pointers auf einen Typ von Funktionen in einen Pointer auf einen anderen Typ von Funktionen
- Wandlungen zwischen Pointern und dem Typ void *
- die Wandlung eines Ausdrucks in den Typ void, aber nicht umgekehrt.

Ein-/Ausgabe

Übersicht

Zur Bildschirmausgabe und Einlesen von der Tastatur kennen Sie bereits die Funktionen `printf` und `scanf` in ihrer Grundform. Ein-/Ausgabe (E/A, I/O) in C kann in drei Gruppen eingeteilt werden

- Standardein- bzw ausgabe (`stdin`, `stdout`, `stderr`)
- String Ein- und Ausgabe
- Datei (File) Ein- und Ausgabe



Ein-/Ausgabe

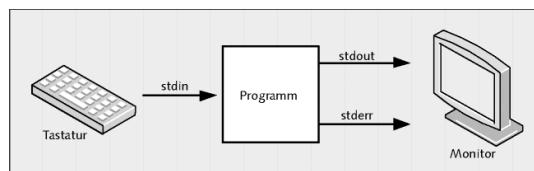
Übersicht

- Ein- und Ausgabefunktionen werden von der stdlib bereitgestellt und die Funktionsdeklarationen sind in stdio.h verfügbar.
- Die Ein-/Ausgabe in C ist Stream-basiert. Ein Stream (zu Deutsch Strom) ist eine Folge von Zeichen oder genauer gesagt eine Folge von Daten-Bytes.
- Eine Folge von Bytes, die in ein Programm fließt, ist ein Eingabe-Stream, und eine Folge von Bytes, die aus einem Programm herausfließt, ist ein Ausgabe-Stream.
- Die Ein- und Ausgabeprogrammierung ist damit geräteunabhängig. Programmierer müssen keine speziellen Ein- und Ausgabefunktionen für jedes Gerät (Tastatur, Festplatte und so weiter) schreiben.
- Das Programm behandelt die Ein- und Ausgabe einfach als einen kontinuierlichen Strom von Byte, egal woher die Eingabe kommt oder wohin die Ausgabe geht.

Ein-/Ausgabe

Standard Streams

- In ANSI C gibt es drei vordefinierte Streams, die auch als Standard-E/A-Dateien bezeichnet werden und alle drei unter Linux verfügbar sind.
 - stdin: Eingabestrom, der üblicherweise mit der Tastatur verbunden ist.
 - stdout: Ausgabestrom, der üblicherweise mit dem Bildschirm (Terminal) verbunden ist.
 - stderr: Ausgabestrom für Fehlermeldungen, der üblicherweise auch mit dem Bildschirm (Terminal) verbunden ist.
- Diese Streams werden automatisch geöffnet, wenn die Ausführung eines C-Programms beginnt, und geschlossen, wenn das Programm zu Ende ist.



Ein-/Ausgabe

Funktionen

In der Standardbibliothek von C gibt es eine Vielzahl von Funktionen, die für die Streameingabe und -ausgabe zuständig sind. Die meisten dieser Funktionen gibt es in zwei Ausprägungen: eine, die immer die Standard-Streams verwendet, und eine, für die der Programmierer den Stream angeben muss.

Verwendet einen der Standard-Streams	Erfordert einen Stream-Namen	Beschreibung
printf()	fprintf()	Formatierte Ausgabe
puts()	fputs()	String-Ausgabe
putchar()	putc(), fputc()	Zeichenausgabe
scanf()	fscanf()	Formatierte Eingabe
gets()	fgets()	String-Eingabe
getchar()	getc(), fgetc()	Zeicheneingabe
perror()		String-Ausgabe nur an stderr

Ein-/Ausgabe

Formatierte Ausgabe mit printf

- Die Funktion printf kann für formatierte Ausgabe verwendet werden. Sie hat folgende Signatur:

```
int printf(char format[], arg1, arg2, ...)
```

- printf akzeptiert eine veränderliche Anzahl von Argumenten und liefert die Anzahl der erfolgreich ausgegebenen Argumente zurück.
- Der format String kann sowohl feste Strings („Hello World“) als auch Formatierungsanweisungen die mit % anfangen beinhalten
- Formatierungsanweisungen haben folgendes Format:

```
%[flags][width][.precision ][length]<specifier>
```

Input: `printf("Color %s, Number %d, Float %.2f", "red", 123456, 3.14);`

Output: Color red, Number 123456, Float 3.14

Ein-/Ausgabe

Formatierte Ausgabe mit printf

specifier	Output	Example
d or i	Signed decimal integer	392
u	Unsigned decimal integer	7235
o	Unsigned octal	610
x	Unsigned hexadecimal integer	7fa
X	Unsigned hexadecimal integer (uppercase)	7FA
f	Decimal floating point, lowercase	392.65
F	Decimal floating point, uppercase	392.65
e	Scientific notation (mantissa/exponent), lowercase	3.9265e+2
E	Scientific notation (mantissa/exponent), uppercase	3.9265E+2
g	Use the shortest representation: %e or %f	392.65
G	Use the shortest representation: %E or %F	392.65
a	Hexadecimal floating point, lowercase	-0xc.90fep-2
A	Hexadecimal floating point, uppercase	-0XC.90FEP-2
c	Character	a
s	String of characters	sample
p	Pointer address	b8000000
n	Nothing printed. The corresponding argument must be a pointer to a signed int. The number of characters written so far is stored in the pointed location.	
%	A % followed by another % character will write a single % to the stream.	%

255

Ein-/Ausgabe

Formatierte Ausgabe mit printf

specifiers								
length	d i	u o x	f F e E g G A A	c	s	p	n	
(none)	int	unsigned int	double	int	char*	void*	int*	
hh	signed char	unsigned char					signed char*	
h	short int	unsigned short int					short int*	
l	long int	unsigned long int		wint_t	wchar_t*		long int*	
ll	long long int	unsigned long long int					long long int*	
j	intmax_t	uintmax_t					intmax_t*	
z	size_t	size_t					size_t*	
t	ptrdiff_t	ptrdiff_t					ptrdiff_t*	
L			long double					

flags	description
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).
+	Forces to precede the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.
(space)	If no sign is going to be written, a blank space is inserted before the value.
#	Used with o, x or X specifiers the value is preceded with 0, 0x or 0X respectively for values different than zero. Used with a, A, e, E, f, F, g or G it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see width sub-specifier).

256

Ein-/Ausgabe

Formatierte Ausgabe mit printf

width	description
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.
*	The width is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

.precision	description
.number	For integer specifiers (d, i, o, u, x, x): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. A precision of 0 means that no character is written for the value 0. For a, A, e, E, f and F specifiers: this is the number of digits to be printed after the decimal point (by default, this is 6). For g and G specifiers: This is the maximum number of significant digits to be printed. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for precision, 0 is assumed.
.*	The precision is not specified in the format string, but as an additional integer value argument preceding the argument that has to be formatted.

257

Ein-/Ausgabe

Formatierte Ausgabe mit printf

```
#include <stdio.h>

int main()
{
    printf ("Characters: %c %c \n", 'a', 65);
    printf ("Decimals: %d %ld\n", 1977, 650000L);
    printf ("Preceding with blanks: %10d \n", 1977);
    printf ("Preceding with zeros: %010d \n", 1977);
    printf ("Some different radices: %d %x %o %#x %#o \n", 100, 100, 100, 100, 100);
    printf ("floats: %4.2f %+0.e %E \n", 3.1416, 3.1416, 3.1416);
    printf ("Width trick: %*d \n", 5, 10);
    printf ("%s \n", "A string");
    return 0;
}
```

```
Characters: a A
Decimals: 1977 650000
Preceding with blanks:      1977
Preceding with zeros: 0000001977
Some different radices: 100 64 144 0x64 0144
floats: 3.14 +3e+000 3.141600E+000
Width trick:    10
A string
```

258

Ein-/Ausgabe

Formatierte Eingabe mit scanf

- Die Funktion `scanf` kann für formatierte Eingabe verwendet werden:
`int scanf(char format[], arg1, arg2, ...)`
- Auch `scanf` akzeptiert eine veränderliche Anzahl von Argumenten und liefert die Anzahl der erfolgreich eingegebene Argumente zurück (oder EOF).
- Die Formatanweisungen sind identisch mit `printf`.
- Wenn mehrere Werte eingelesen werden sollen müssen diese in der Eingabe durch Leerzeichen (oder tabs) getrennt sein.
- Leerzeichen und Tabulatoren im Formatstring werden ignoriert werden (sie können dazu verwendet werden, den Formatstring lesbarer zu machen).
- Wichtig: die Argumente müssen Adressen von Variablen sein.
- Wichtig: stimmt die Eingabe nicht mit dem Formatzeichen überein, dann wird die Eingabe abgebrochen (der Rest wird nicht gelesen).

Ein-/Ausgabe

Formatierte Eingabe mit scanf

```
#include <stdio.h>

int main ()
{
    char given [80];
    char family [80];
    int i;

    printf ("Enter your family name: ");
    scanf ("%79s %79s",given,family);
    printf ("Enter your age: ");
    scanf ("%d",&i);
    printf ("Mr. %s %s is %d years old.\n",given,family,i);
    printf ("Enter a hexadecimal number: ");
    scanf ("%x",&i);
    printf ("You have entered %#x (%d).\n",i,i);

    return 0;
}
```

Ein-/Ausgabe

Lesen und Schreiben von Dateien

- Oft ist es auch nützlich Dateien direkt lesen und schreiben zu können.
- Dazu gibt es in der Standardbibliothek verschiedene Funktionen.
- Genereller Ablauf:
 1. Datei öffnen
 2. In die Dateien schreiben bzw. aus ihr lesen
 3. Datei schließen
- Die Funktion **fopen** dient dazu, einen Datenstrom (Stream) zu öffnen. Datenströme sind Verallgemeinerungen von Dateien. Die Syntax dieser Funktion lautet:

```
FILE *fopen (const char *Pfad, const char *Modus);
```

Ein-/Ausgabe

Lesen und Schreiben von Dateien

Der Pfad ist der Dateiname, der Modus darf wie folgt gesetzt werden:

- r - Datei nur zum Lesen öffnen (READ)
- w - Datei nur zum Schreiben öffnen (WRITE), löscht den Inhalt der Datei, wenn sie bereits existiert
- a - Daten an das Ende der Datei anhängen (APPEND), die Datei wird nötigenfalls angelegt
- r+ - Datei zum Lesen und Schreiben öffnen, die Datei muss bereits existieren
- w+ - Datei zum Lesen und Schreiben öffnen, die Datei wird nötigenfalls angelegt
- a+ - Datei zum Lesen und Schreiben öffnen, um Daten an das Ende der Datei anzuhängen, die Datei wird nötigenfalls angelegt

Ein-/Ausgabe

Lesen und Schreiben von Dateien

- Als Ergebnis bekommt man einen Zeiger auf den Datentyp FILE, was einem Ein-/Ausgabestrom entspricht.
- Die Funktion fclose dient dazu, die mit der Funktion fopen geöffneten Datenströme wieder zu schließen. Die Syntax dieser Funktion lautet:

```
int fclose (FILE *datei);
```

- Alle nicht geschriebenen Daten des Stromes *datei werden gespeichert, alle ungelesenen Eingabepuffer geleert, der automatisch zugewiesene Puffer wird befreit und der Datenstrom *datei geschlossen. Der Rückgabewert der Funktion ist EOF, falls Fehler aufgetreten sind, ansonsten ist er 0.

Ein-/Ausgabe

Lesen und Schreiben von Dateien

```
#include <stdio.h>
int main (void)
{
    FILE *datei;
    datei = fopen ("testdatei.txt", "w");
    if (datei == NULL)
    {
        printf("Fehler beim Öffnen der Datei.");
        return 1;
    }
    fprintf (datei, "Hallo, Welt\n");
    fclose (datei);
    return 0;
}
```

Ein-/Ausgabe

Kommandozeile

- Viele Standardprogramme insbesondere unter U*ix (Linux, Xenix, etc.) akzeptieren Flags, z.B. gcc -Wall -o world world.c
- Sehr praktisch da so das Programm den Nutzer nicht erst im Dialog befragen muss, was es tun soll sondern durch die Flags gesteuert wird.
- Diese Parameter kann das main Programm lesen

```
int main(int argc, char* argv[])
```
- argc enthält die Anzahl der Kommandozeilenargumente einschließlich des Programmnamens
- der argv Vektor enthält die einzelnen Argumente (inkl. Programmname)
- also bei gcc -Wall -o world world.c:
- argc ist 5, argv[0] ist „gcc“, argv[1] ist „-Wall“, argv[2] ist „-o“, argv[3] ist „world“, argv[4] ist „world.c“,

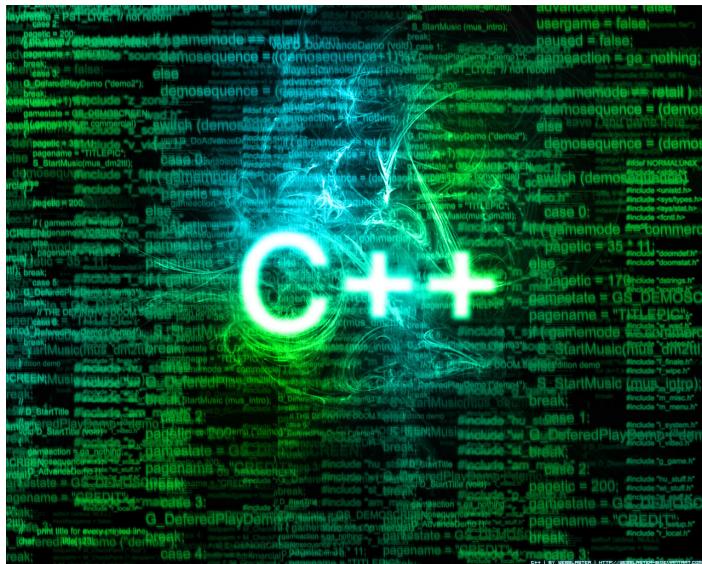
Ein-/Ausgabe

Kommandozeile

```
#include<stdio.h>

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

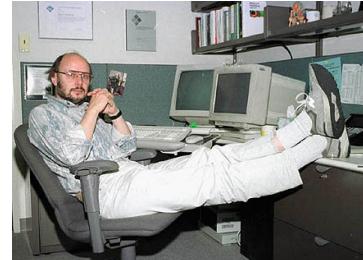
main.exe hier koennten flags stehen
Argument an Stelle 0 ist: main.exe
Argument an Stelle 1 ist: hier
Argument an Stelle 2 ist: koennten
Argument an Stelle 3 ist: flags
Argument an Stelle 4 ist: stehen
```



267

Grundlagen C++

- Wurde ab 1979 von Bjarne Stroustrup bei AT&T als Erweiterung der Programmiersprache C entwickelt.
 - C++ basiert auf der Programmiersprache C wie in ISO/IEC 9899:1990 beschrieben.
 - 1998 wurde die Sprache C++ genormt (ISO/IEC 14882:1998) . 2003 wurde ISO/IEC 14882:2003 verabschiedet, eine Nachbesserung der Norm von 1998.
 - C++ ist eine Obermenge von C, das heißt, es hat alle Syntaxeigenschaften von C, aber noch einige darüber hinaus.
 - Die Sprache ermöglicht eine Vielzahl von Programmierstilen, von rein prozedurelem Stil bis zu streng objektorientiertem Stil.



Grundlagen C++

Zusätzlich zu den in C vorhandenen Möglichkeiten bietet C++ folgende neuen Konzepte:

- Weitere Datentypen sowie neuartige Typumwandlungsmöglichkeiten
- Klassen mit Mehrfachvererbung und virtuellen Funktionen
- Ausnahmebehandlung (Exceptions)
- Schablonen (Templates)
- Namensräumen
- Inline-Funktionen
- Überladen von Operatoren und Funktionsnamen
- Referenzen
- Operatoren zur Freispeicherverwaltung und mit der C++ Standardbibliothek eine erweiterte Bibliothek.

Grundlagen C++

Aufbau eines C++ Programms

```
#include <iostream>
#include <string>
using namespace std;

int main() {
    cout << "Ich bin Ihr erstes C++ Programm. ";
    cout << "Wer sind Sie?" << endl;

    string name;
    cin >> name;

    cout << "Hallo, " << name << endl;
}
```

Auswahl des Namensbereiches std

Ausgabe auf Konsole cout durch Infixoperator „<<“

Neuer Datentyp string (ohne Längenangabe)

Variablendefinition nach Anweisung

Eingabe von der Tastatur cin durch Infixoperator „>>“

Zeilenumbruch

Namensräume (Name Spaces)

- Je größer ein Projekt wird, desto wahrscheinlicher werden Namenskollisionen, insbes. bei global gültigen Namen (globale Variablen, Typen, Funktionen).
- Idee: jedem verwendeten Namen einen Namensbereich zuordnen → Namenskollisionen nur noch innerhalb eines Namensbereiches möglich.
- Beispiel: alle Namen der Standardbibliothek (auch die Namen der alten ANSI-C-Funktionen und -Typen) sind im Namensbereich `std` deklariert
- In einem anderen Namensbereich können die selben Namen nochmals vergeben werden.
- Dem Namen muss dann der Namensraumes getrennt durch `::` vorangestellt werden.

Namensräume (Name Spaces)

Beispiel:

```
int main() {  
    std::cout << "Jetzt geht es los: ";  
    int cout = 3;  
    std::cout << cout << std::endl;  
    std::cout << std::endl;  
    return 0;  
}
```

Keine Verschattung von cout aus
dem Namensraum std

Namensräume (Name Spaces)

- Angabe des Namensraumes oft sehr umständlich ist → man kann zu jedem Zeitpunkt einen oder mehrere voreingestellte Namensbereiche definieren.
- Die darin deklarierten Namen können direkt verwendet werden solange sie nicht kollidieren.
- Ein Namensbereich kann vorab ausgewählt werden durch:

```
using namespace Namensbereich;
```
- Eine using-Anweisung gilt bis zum Ende des umgebenden Blocks, beziehungsweise bis zum Ende des Quelltextes, wenn sie außerhalb einer Funktion steht.
- Beispiel:

```
using namespace std;
int main() {
    cout << "Jetzt geht es los: " << endl;
    return 0;
}
```

Namensräume (Name Spaces)

- Man kann auch gezielt für einzelne Bezeichner den Namensraum vordefinieren:

```
using Name_des_Namensbereichs::Bezeichner;
```
- Beispiel:

```
using std::cout;
using std::endl;
int main() {
    cout << "Jetzt geht es los:" << endl;
}
```
- Eigene Namensräume definieren:

```
namespace test {
    int my_var_1;
    double my_var_2;
}
```
- Zugriff: `test::my_var_1` bzw. `test::my_var_2`

Namensräume (Name Spaces)

- Durch wiederholte namespace-Blöcke kann man zu einem bereits existierenden Namensbereich weitere Namen hinzufügen.
- Ein Entfernen bereits enthaltener Namen ist nicht möglich.

```
namespace MeinNamensraum {      // erste Deklaration
    int i;
}
namespace MeinNamensraum {      // erweiternde Deklaration
    int j;
}

int main() {
    MeinNamensraum::i = 1;
    MeinNamensraum::j = 2;
    cout << MeinNamensraum::i << " " <<
        MeinNamensraum::j << endl;
}
```

275

Namensräume (Name Spaces)

- Hat man einen Namensbereich bereits definiert, so kann man mit einer Art Zuweisung dafür noch einen anderen Namen vergeben.

```
namespace EinFurchtbarLangerName {
    int i;
}
namespace EFLN = EinFurchtbarLangerName;

int main() {
    EinFurchtbarLangerName::i = 25;
    EFLN::i = 26;    // greift auf das gleiche i zu,
                     // wie vorhergehende Zeile
    cout << EinFurchtbarLangerName::i << endl;
}
```

alternativer
Name

276

Scope Resolution Operator

- Mit dem Scope Resolution Operator :: kann zwischen einer globalen und lokalen Verwendung ein- und desselben Namens unterschieden werden.
- Normalerweise bezieht sich ein Name im Zweifelsfall auf die innerste Vereinbarung (Verschattungsprinzip).
- Mit dem Operator :: wird dagegen das globale Objekt angesprochen.

```
int variable = 10; // globale Variable

int main() {
    double variable = 3.1415; // lokale Variable gleichen Namens
    // Zugriff auf lokale u. globale Variable:
    cout << variable << " " << ::variable << endl;
    if( true ) {
        char variable = 'A'; // noch eine lokale Variable gleichen Namens
        // Äußere lokale Variable ist verschattet
        // Zugriff auf innere lokale Variable und globale Variable
        cout << variable << " " << ::variable << endl;
    }
    return 0;
}
```

277

Definition von Variablen

- In C++ müssen Variablendefinitionen in Funktionen nicht mehr am Anfang eines Blocks vor den ausführbaren Anweisungen stehen, sondern nur vor ihrer ersten Verwendung.
- Vorteil: ist übersichtlicher, wenn bspw. Schleifenzähler oder andere nur sehr lokal verwendete Variablen direkt bei ihrer Verwendung definiert werden.

```
#include <iostream>
using namespace std;
int main() {
    cout << "Start:" << endl;
    for ( int i=0; i<10; i++ ) {
        cout << "i ist " << i << endl;
    }
    return 0;
}
```

ist in C90 nicht
möglich
(jedoch in C99)

278

Vereinfachte Typdefinitionen

- Alle Typnamen, einschließlich der **struct**-, **union**- und **enum**-Bezeichner, werden vom Compiler in einem „Namensraum“ untergebracht und müssen deshalb auch untereinander eindeutig sein.
- Dafür kann man bei weiteren Vereinbarungen nach der ersten Deklaration auf das Schlüsselwort **struct**, **union** bzw. **enum** weglassen.

```
#include <iostream>
using namespace std;

struct koordinaten {
    float x;
    float y;
};

int main() {
    koordinaten k = {1.0,2.0};

    cout << "(" << k.x << "," << k.y << ")" << endl;

    return 0;
}
```

279

Inline Funktionen

- Wird eine Funktion als inline deklariert, so muss der Compiler keine Funktion dafür erzeugen, sondern kann den entsprechenden Code direkt an der Aufrufstelle einfügen.
- Programm ist dadurch schneller (kein Funktionsaufruf) – jedoch unter Umständen auch größer.
- Compiler kompiliert deshalb nicht beliebig komplexe und lange Funktionen tatsächlich inline, sondern nur kurze Funktionen, d. h. der Compiler bestimmt, welche Funktion als inline realisiert wird und welche nicht.
- Wirkungsweise ähnlich einem Makro, wobei aber bei Inline Funktionen die Typsicherheit gewährleistet ist.

280

Inline Funktionen

```
#include <iostream>
using namespace std;

inline int min(int a, int b) {
    return a<b ? a : b;
}

int main() {
    int minimum;

    minimum = min(3,4);
    cout << minimum << endl;
}

return 0;
}
```

```
#include <iostream>
using namespace std;

int main() {
    int minimum;

    minimum = 3<4 ? 3 : 4;
    cout << minimum << endl;
}

return 0;
}
```

281

Funktionsüberladung

- In C++ können mehrere Funktionen mit ein und demselben Namen versehen, d. h. **überladen** werden.
- Beim Aufruf der Funktion wird die Parameterliste analysiert (Anzahl und Typ der Parameter)
- Der Compiler wählt unter den überladenen Funktionen diejenige aus, deren Parameterliste "am besten" zu den Parametern eines Aufrufs passt.
- Achtung: Der Typ des Rückgabewerts wird nicht zur Unterscheidung herangezogen, auch wenn die überladenen Funktionen unterschiedliche Typen zurückgeben!
- Hinter einem bestimmten Funktionsnamen können sich also mehrere verschiedene Funktionen verstecken → Polymorphie
- Genauer: Ad-hoc-Polymorphie, da die Bindung zwischen Aufrufstelle und tatsächlich aufgerufener Funktion zur Übersetzungszeit hergestellt wird.

282

Funktionsüberladung

```
#include <iostream>
using namespace std;

void meineFunktion()
{
    cout << "Funktion ohne Parameter" << endl;
}

void meineFunktion(int x)
{
    cout << "Funktion mit Parameter: " << x << endl;
}

int main() {
    meineFunktion();
    meineFunktion(8);

    return 0;
}
```

283

Default-Wert bei Aufrufparametern

- Wenn eine C++-Funktion für einzelne Parameter oft den gleichen Wert bekommt, und nur manchmal ein anderer Wert angegeben wird, dann kann man dafür auch einen Default-Wert setzen.
- Die Parameter mit einem solchen vordefinierten Wert müssen die letzten sein, und dürfen keine Lücke aufweisen.

284

Default-Wert bei Aufrufparametern

```
#include <iostream>
using namespace std;

int f(string s, int i = 0) {
    return i;
}
int g(string s, int i = 0, int j = 12) {
    return i+j;
}
int h(string s, int i = 0, int j) {      // Fehler da Lücke!
    return i+j;
}

int main() {
    cout << f( "abc" ) << endl;           // i ist 0
    cout << f( "abc", 17 ) << endl;        // i ist 17
    cout << g( "abc" ) << endl;           // i ist 0, j ist 12
    cout << g( "abc", 2 ) << endl;         // i ist 2, j ist 12
    return 0;
}
```

285

Referenzvariable

- In C++ gibt es neben den Wert- und Zeigervariablen noch sog. **Referenzvariablen**.
- Eine Referenz auf einen Speicher kennzeichnet man mit einem vorangestellten &.
- Mit Referenzvariablen können für eine Variable mehrere Namen vergeben werden (Alias-Namen)
- Definition von Referenzvariablen: `typ& zweitname = erstname;`
- Durch Verwendung von Referenzvariablen erspart man sich die lästigen Zeigeroperationen.
- Referenzvariablen können im Gegensatz zu Zeigern nicht ungültig werden (solange die referenzierte Variable existiert)
- Die Dereferenzierung erfolgt automatisch.
- In der Regel werden Referenzen nur als Parameter in Funktionen verwendet

286

Referenzvariable

Erzeugen der Referenzvariable

```
#include <iostream>
using namespace std;

int main() {
    int a;
    int &b = a; //int a;
               //int b = &a;

    a = 10; // a = 10;
    b = 12; // *b = 12;
    cout << a << endl; // cout << a << endl;
    return 0;
}
```

Die Initialisierung einer Referenzvariablen muss immer in der Definition erfolgen u. die Referenz kann später nicht mehr geändert werden (im Gegensatz zu Zeigern).

Automatische Dereferenzierung

287

Parameterübergabe

- Aktuelle Parameter können an eine Funktion entweder als Wert (call-by-value) oder als Referenz übergeben werden.
- In C kann eine Referenzübergabe nur über Zeiger realisiert werden.
- In C++ kann eine Referenzübergabe auch ohne (sichtbare) Zeiger realisiert werden:

```
typ function (typ & parameter)
```

- Referenzvariablen müssen nicht explizit referenziert bzw. dereferenziert werden.

288

Parameterübergabe

```
struct koordinaten {  
    float x;  
    float y;  
};  
void print_koordinaten (koordinaten k) {  
    cout << "(" << k.x << "," << k.y << ")" << endl;  
}  
void print_koordinaten (const koordinaten* k) {  
    cout << "(" << k->x << ";" << k->y << ")" << endl;  
}  
void print_koordinaten_ref (const koordinaten& k) {  
    cout << "(" << k.x << ":" << k.y << ")" << endl;  
}  
int main () {  
    koordinaten meine_koordinate = {1.0, 2.5};  
  
    print_koordinaten(meine_koordinate);  
    print_koordinaten(&meine_koordinate);  
    print_koordinaten_ref(meine_koordinate);  
    return 0;  
}
```

Wertübergabe

Referenzübergabe
über ZeigerReferenzübergabe mit
Referenzparameter

289

Einführung in die Objektorientierung

- Objektorientiert ist ein **Paradigma**, mit der die Abbildung von „Dingen aus der realen Welt“ (Problemraum) in Einheiten eines laufenden Programms (Lösungsraum) geleistet werden soll.
- Definition von „objektorientiert“ nach ISO:

**Eine Technik oder Programmiersprache betreffend,
die Objekte, Klassen und Vererbung unterstützt.**

- Im Zentrum des Paradigmas steht das Objekt, d.h. wir bilden die Dinge der realen Welt ab in Objekte.
- Jedes Objekt hat eine Menge von Eigenschaften, die sogenannter **Attribute** (Daten) eines Objekts, und zeigt ein spezifisches Verhalten, welche durch die **Methoden** des Objekts modelliert werden.

290

Einführung in die Objektorientierung

Einfaches Beispiel:

Ein Tisch hat folgende Eigenschaften:

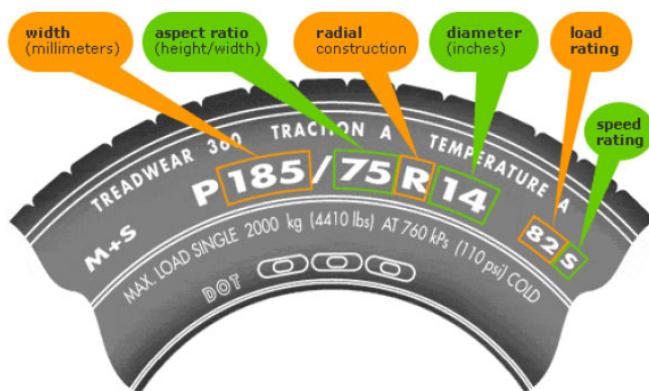
- Länge
- Breite
- Höhe
- Farbe
- Gewicht
- ...



291

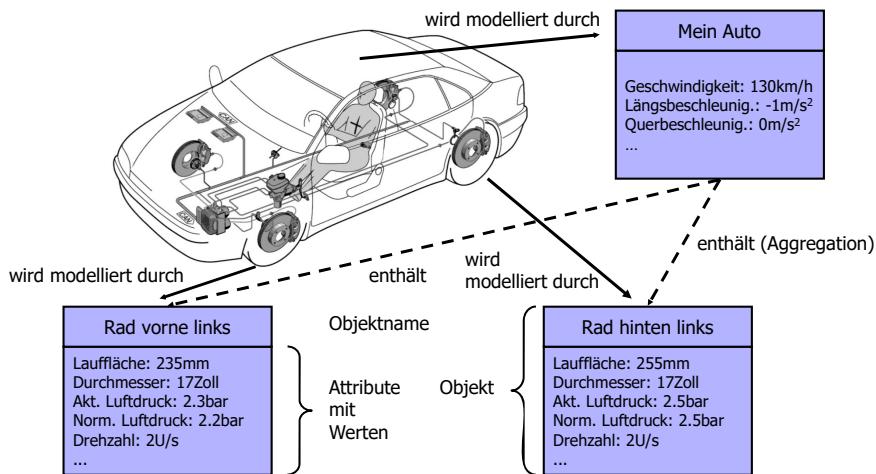
Einführung in die Objektorientierung

Ein weiteres Beispiel:



292

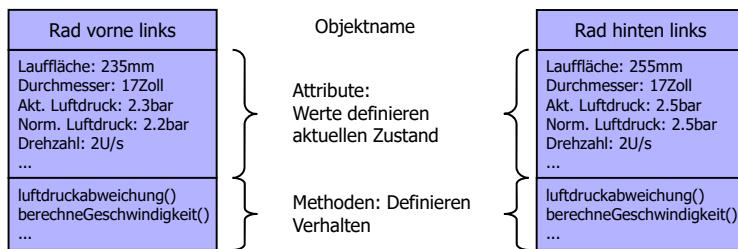
Einführung in die Objektorientierung



293

Einführung in die Objektorientierung Objekt

- Ein Objekt hat einen **Zustand**, der durch die Menge seiner aktuellen Attributwerte beschrieben wird.
- Ein Objekt hat zusätzlich ein **Verhalten**, das durch die Menge seiner **Operationen (Methoden, Funktionen)** beschrieben wird.

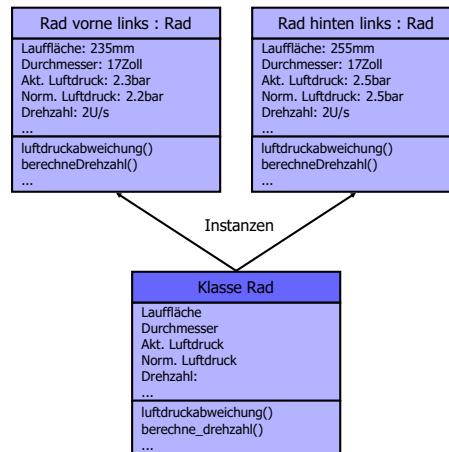


294

Einführung in die Objektorientierung

Klasse

- Besitzen mehrere Objekte dieselben Attribute und Eigenschaften, dann können diese zu einer **Klasse** gruppiert werden.
- Eine Klasse spezifiziert die gemeinsamen Attribute und Methoden.
- Eine Klasse ist ein Stempel, Bauplan, Schablone.
- Ein Objekt ist eine **Instanz** einer Klasse (Exemplar).



295

Einführung in die Objektorientierung

Klassenkonzept in C++

- Eine Klasse ist eine Erweiterung der aus C bekannten Struktur (**struct**), also ein Datentyp.
- Wesentlichen Erweiterungen:
 - in der Klasse können Funktionen (Methoden, *member functions*) untergebracht werden
 - eine Klasse kann von einer bereits vorhandenen Klasse abgeleitet werden (Vererbung)
 - es können Zugriffrechte auf die Elemente der Klasse vergeben werden:
 - private: kein Zugriff von außen
 - public: Zugriff von außen ist möglich

296

Einführung in die Objektorientierung

Klassenkonzept in C++

- Definition einer Klasse mit Schlüsselwort `class`:

```
class name {  
    ... // Datenelemente und Funktionen  
};
```

- Beispiel:

```
class Rad {  
    float umdrehung;  
    float durchmesser;  
    float druck;  
    float solldruck;  
  
    float berechne_geschwindigkeit(){  
        return durchmesser * 3.14 * umdrehung;  
    }  
};
```

Members

Attribute

Methode

Strichpunkt ist notwendig!

297

Einführung in die Objektorientierung

Zugriffsrechte

- Werden keine Zugriffsrechte angegeben, dann werden standardmäßig alle Elemente einer Klasse als `private` deklariert.
→ kein Zugriff von außen (im Gegensatz zu `struct`).
- Mit `public` kann Zugriff von außen gewährt werden:

```
class Rad {  
    private:  
        float umdrehung;  
        float solldruck;  
    public:  
        float durchmesser;  
        float druck;  
        float berechne_geschwindigkeit(){  
            return durchmesser * 3.14f * umdrehung;  
        }  
};
```

private Elemente

öffentliche Elemente

298

Einführung in die Objektorientierung

Objekte erzeugen

- Eine Klasse ist wie ein Datentyp verwendbar → es können "Variablen" dieser Klasse erzeugt werden.
- Die "Variablen" von Klassen heißen **Objekte**.
- Der Zugriff auf die Elemente (Attribute und Methoden) eines Objekts geschieht wie bei einer Struktur mit dem .-Operator.

```
int main(void) {  
analog C float f=11.56;  
    Rad rvl;  
    rvl.umdrehung = 10;  
    rvl.durchmesser = 0.5;  
    cout << rvl.berechne_geschwindigkeit() << endl;  
}
```

Objekt **rvl** der Klasse **Rad** erzeugen

Zugriff mit .-Operator

299

Einführung in die Objektorientierung

Kapselung der Attribute

- Die Kapselung von Daten ist ein wichtiges Konzept in der Objektorientierung.
- Ein direkter Zugriff auf die Attribute einer Klasse sollte nicht möglich sein.
- Ein Zugriff sollte nur über sogenannte Setter- und Getter-Methoden der Klasse erfolgen:

```
class Rad {  
private:  
    float umdrehung;  
    ...  
public:  
    // Setter und Getter ...  
    void set_umdrehung(float u){umdrehung = u;}  
    float get_umdrehung(void){return umdrehung;}  
    ...  
}
```

300

Einführung in die Objektorientierung

Dynamische Objekte

- Dynamische Objekte entsprechen den dynamischen Variablen in C, die dort mit Hilfe der Funktion malloc() angelegt werden.
- Dynamische Objekte werden bei Bedarf mit dem **new**-Operator zur Laufzeit auf dem Heap angelegt.
- Der new-Operator gibt (wie die Funktion malloc()) einen Zeiger auf das Objekt zurück.

```
int main(int argc, char *argv[]) {  
    Rad *rwl;  
    rvl = new Rad;  
    rvl->set_umdrehung(10.56);  
    ...  
}
```

Einführung in die Objektorientierung

Dynamische Objekte

- Die Gültigkeit und Lebensdauer eines Objekts wird **nicht** durch Blockgrenzen, d.h. **nicht** durch die statische Struktur des Programms bestimmt.
- Objekte werden jedoch nicht automatisch gelöscht.
- Objekte müssen explizit mit dem **delete**-Operator vom Heap wieder entfernt werden.
- Anschließend darf die Zeigervariable nicht mehr verwendet werden. Sie sollte auf NULL gesetzt werden.
- Werden Objekte nicht wieder korrekt freigegeben, so kann (insbes. nach längerer Programmalaufzeit) der verfügbare Speicher erschöpft.
- Objekte werden automatisch bei Programmende gelöscht.

Einführung in die Objektorientierung

Dynamische Objekte

```
class Rad {  
    float umdrehung; float durchmesser; float druck; float solldruck;  
public:  
    void set_umdrehung (float u){  
        umdrehung = u; }  
    void set_durchmesser (float d){  
        durchmesser = d; }  
    float berechne_geschwindigkeit(){  
        return durchmesser * 3.14 * umdrehung; }  
};  
  
int main(int argc, char *argv[]){  
    Rad *rwl;  
    rwl = new Rad;  
    rwl->set_umdrehung(10.56); rwl->set_durchmesser (0.7);  
    cout << rwl->berechne_geschwindigkeit() << endl;  
    delete rwl;  
    rwl = NULL;  
    return 0;  
}
```

303

Einführung in die Objektorientierung

Trennung von Deklaration und Definition

- In der Programmiersprache Java besteht der Zwang, für jede Klasse eine Quelltextdatei zu erzeugen. In C++ kann man die Unterteilung nach eigenen Vorstellungen vornehmen.
- **Empfehlung:** Jede Klasse in eigene Header- und Quellcode-Dateien mit dem Namen der Klasse:
 - meinklassenname.h → für die Deklaration
 - meinklassenname.cpp → für die Definition der Methoden
- Gründe für eine Aufteilung des Quelltextes auf mehrere Dateien:
 - bessere Übersicht und leichteres Auffinden von Klassen/Methoden
 - bei der Entwicklung im Team wird es vermeidbar, dass zwei Programmierer an der gleichen Datei arbeiten müssen
 - schnelleres Kompilieren: Dateien müssen nur übersetzt werden, wenn sie selbst oder durch sie inkludierte Header-Dateien geändert wurden.

304

Einführung in die Objektorientierung

Trennung von Deklaration und Definition

- Methoden von Klassen müssen (und sollen) nicht innerhalb der Klassedeclaration definiert werden, sondern außerhalb.
- Deklaration und Definition (Implementierung) einer Klasse sollten immer getrennt werden.
- Deklaration einer Klasse gehört in eine Header-Datei.

```
class Rad {  
    private:  
        // Attribute  
        float umdrehung;  
        float durchmesser;  
        float solldruck;  
        float druck;  
    public:  
        // Konstruktoren  
        Rad();  
        Rad(float u, float du, float sd, float dr);  
  
        // Kopierkonstruktoren  
        Rad(const Rad& r);  
  
        // Methoden  
        void set_umdrehung(float u);  
        float get_umdrehung(void);  
        void set_durchmesser(float d);  
        float get_durchmesser(void);  
        float berechne_geschwindigkeit();  
};
```

305

Einführung in die Objektorientierung

Trennung von Deklaration und Definition

- Die Definition der Methoden erfolgt in einer separaten Quellcode-Datei.
- In dieser Quelldatei muss die zugehörige Header-Datei mit dem #include-Präprozessorbefehl eingelesen (inkludiert) werden.
- Der Bezug zur Klasse muss mit dem Scope Resolution Operator :: hergestellt werden.

```
// Methoden  
void Rad::set_umdrehung(float u) {  
    umdrehung = u;  
}  
float Rad::get_umdrehung(void) {  
    return umdrehung;  
}  
void Rad::set_durchmesser(float d) {  
    durchmesser = d;  
}  
float Rad::get_durchmesser(void) {  
    return durchmesser;  
}  
float Rad::berechne_geschwindigkeit(){  
    return durchmesser * 3.14f * umdrehung;  
}
```

306

Einführung in die Objektorientierung

Trennung von Deklaration und Definition

- Die Header-Datei (nicht die cpp-Datei!) muss in jeder Datei, in der die Klasse verwendet wird mit #include inkludiert werden.
- Die eigentliche Definition der Klasse (cpp-Datei) wird getrennt übersetzt und mit dem Linker dazu gebunden.

```
#include <iostream>
#include "rad.h"
#include "auto.h"

using namespace std;

int main(int argc, char *argv[]) {

    Rad *rwl = new Rad(10,0.7,0,0);
    Rad *rvr = new Rad(*rwl);
    Rad *rhl = new Rad(*rwl);
    Rad *rhr = new Rad(*rwl);

    cout << rwl->berechne_geschwindigkeit() << endl;
    cout << rvr->berechne_geschwindigkeit() << endl;

    Auto *opel = new Auto;
    Auto *bmw = new Auto(rwl,rvr,rhl,rhr);
    Auto *audi = new Auto(*bmw);
```

Einführung in die Objektorientierung

Konstruktor

- Ein Konstruktor dient dazu, ein Objekt in einen definierten Anfangszustand zu versetzen, das heißt Speicherplatz für die Attribute bereitzustellen und gegebenenfalls die Attribute mit sinnvollen Anfangswerten zu initialisieren.
- Konuktoren haben einige besondere Eigenschaften, die sie von allen anderen Methoden unterscheiden:
 - Aufrufe von Konuktoren werden automatisch in das Programm eingefügt. Jedes Mal, wenn ein Objekt erzeugt wird und die Klasse über einen entsprechenden Konstruktor verfügt, wird dieser aufgerufen. Explizite Konuktoraufrufe gibt es also nicht.
 - Konuktoren tragen denselben Namen wie die Klasse.
 - Konuktoren haben keine Rückgabewerte (auch nicht void).

Einführung in die Objektorientierung

Konstruktor

Man unterscheidet mehrere Arten von Konstruktoren:

- **Standardkonstruktor** (default constructor):
Dieser wird bei jeder Erzeugung eines Objekts der Klasse verwendet, wenn kein anderer Konstruktor definiert worden ist.
- **Allgemeiner Konstruktor** (vom Programmierer geschrieben):
Dieser kann beliebige Argumente haben und wie eine Methode überladen werden. Es können also mehrere Konstruktoren mit verschiedenen Argumentlisten existieren.
- **Kopierkonstruktor** (copy constructor): Dieser dient dazu, ein Objekt dieser Klasse mit einem anderen derselben Klasse zu initialisieren.
- **Typumwandlungskonstruktor**:
Dieser hat nur ein Argument und dient dazu, einen anderen Datentyp in die Klasse (als Typ gesehen) umzuwandeln.

Einführung in die Objektorientierung

Standardkonstruktor

- Der sogenannte **Standardkonstruktor** oder **Defaultkonstruktor** einer Klasse hat keine Parameter.
- Er wird automatisch definiert, wenn überhaupt kein Konstruktor explizit definiert wird.
- Der Standardkonstruktor reserviert Speicher, macht aber **keine** Initialisierung.
- Der Standardkonstruktor wird im allgemeinen durch einen selbst geschriebenen Konstruktor ersetzt.

Rad () {
}

Einführung in die Objektorientierung

Allgemeiner Konstruktor

```
class Rad {  
    float umdrehung; float durchmesser; float druck; float solldruck;  
public:  
    void set_umdrehung (float u){  
        umdrehung = u; }  
    void set_durchmesser (float d){  
        durchmesser = d; }  
    float berechne_geschwindigkeit(){  
        return durchmesser * 3.14 * umdrehung; }  
    Rad () {  
        umdrehung = 0;  
        durchmesser = 0;  
        druck = 0;  
        solldruck = 0;  
    }  
};
```

311

Einführung in die Objektorientierung

Allgemeiner Konstruktor

- Es können für eine Klasse mehrere Konuktoren mit unterschiedlicher Anzahl und Typ von Parametern definiert werden (→ Überladen der Konuktoren). Achtung: es gibt dann keinen Standardkonuktur!
- Die Parameter können dann beim Anlegen des Objektes hinter dem Objektnamen angegeben werden.

```
Rad () {  
    umdrehung = 0;  
    durchmesser = 0;  
    druck = 0;  
    solldruck = 0;  
}  
Rad ( float um, float du, float dr, float so){  
    umdrehung = um;  
    durchmesser = du;  
    druck = dr;  
    solldruck = so;  
}
```

312

Einführung in die Objektorientierung

Allgemeiner Konstruktor

```
class Rad {
    float umdrehung; float durchmesser; float druck; float solldruck;
public:
    void set_umdrehung (float u){
        umdrehung = u; }
    void set_durchmesser (float d){
        durchmesser = d; }
    float berechne_geschwindigkeit(){
        return durchmesser * 3.14 * umdrehung; }
Rad () {
    umdrehung = 0; durchmesser = 0; druck = 0; solldruck = 0;
}
Rad ( float um, float du, float dr, float so){
    umdrehung = um; durchmesser = du; druck = dr; solldruck = so;
}
};

int main(int argc, char *argv[]){
    Rad *rvl;
    rvl = new Rad(10.5,0.7,2.0,2.3);
    cout << rvl->berechne_geschwindigkeit() << endl;
    delete rvl;
    rvl = NULL;
    return 0;
}
```

313

Einführung in die Objektorientierung

Allgemeiner Konstruktor

- **Initialisierungsliste** dient zur Initialisierung der Klassenattribute direkt bei ihrer Erzeugung
- Sie ist eine Komma-separierte Liste von Initialisierern und befindet sich zwischen dem Kopf und dem Rumpf der Konstruktordefinition und ist durch einen Doppelpunkt vom Kopf abgesetzt
- Initialisierer: attributname(initialwert)

```
Rad ( float um, float du, float dr, float so):
    umdrehung(um), durchmesser(du), druck(dr), solldruck(so) {
```

314

Einführung in die Objektorientierung

Destruktor

- Der Destruktor ist das Gegenstück zum Konstruktor.
- Der Destruktor einer Klasse wird automatisch beim Löschen eines Objekts aufgerufen, d.h. am Ende des Gültigkeitsbereiches oder beim delete-Operator.
- Ein Destruktor hat keine Argumente und keinen Rückgabewert.
- Er ermöglicht "Aufräumungsoperationen", bevor das Objekt wirklich gelöscht wird (z. B. Speicherplatzfreigabe). Dies ist dann wichtig, wenn das Objekt andere Objekte enthält, die dann auch gelöscht werden sollen.
- Destruktoren haben einen Namen, der identisch mit dem Klassennamen ist, jedoch wird ein Tilde \sim vorangestellt.

Einführung in die Objektorientierung

Destruktor

```
class Auto {
    Rad *rwl; Rad *rvr; Rad *rhl; Rad *rhr;
public:
    float berechne_geschwindigkeit() {
        return ( rvl->berechne_geschwindigkeit() +
                 rvr->berechne_geschwindigkeit() +
                 rhl->berechne_geschwindigkeit() +
                 rhr->berechne_geschwindigkeit() )/4;
    }
    Auto() {
        rvl = new Rad; rvr = new Rad; rhl = new Rad; rhr = new Rad;
    }
    ~Auto() {
        delete rvl; delete rvr; delete rhl; delete rhr;
    }
};
```

Einführung in die Objektorientierung

Kopierkonstruktor

- Der **Kopierkonstruktor** erzeugt für eine neues Objekt eine Kopie eines bereits existierenden Objektes.
- Dies ist z.B. dann nötig, wenn ein neues Objekt mit einem anderen Objekt der gleichen Klasse initialisiert werden soll.

```
Rad rvl(10,0.8,0,0);  
Rad rvr(rvl);  
Rad rhl = rvl;
```

Objekt **rvl** mit Konstruktor erzeugen

Objekt **rvr** mit Objekt **rvl** initialisieren.

Objekt **rhl** mit Objekt **rvl** initialisieren.
Achtung: keine Zuweisung!

- Die Aufgabe des Kopierkonstruktors ist es, die Attributwerte umzukopieren.

Einführung in die Objektorientierung

Kopierkonstruktor

- Wird kein expliziter Kopierkonstruktor definiert, dann wird ein Standard Kopierkonstruktor zur Verfügung gestellt. Dieser kopiert einfach die Attributwerte um und macht damit eine sog. **flache Kopie**.
- Der implizite Kopierkonstruktor sieht so aus:

```
Rad(const Rad& r) : umdrehung(r.umdrehung), durchmesser(r.durchmesser),  
solldruck(r.solldruck), druck(r.druck) {}
```

- Dieses einfache Umkopieren reicht nicht, wenn ein Objekt Zeiger enthält, d.h. z.B. auf andere Objekte auf dem Heap verweist (siehe Klasse Auto).

Einführung in die Objektorientierung

Kopierkonstruktor

- Wie beim Destruktor muss dann ein expliziter Kopierkonstruktor geschrieben werden, um eine **tiefe Kopie** zu erzeugen:

```
Rad(const Rad& r):  
    umdrehung(r.udrehung),  
    durchmesser(r.durchmesser),  
    solldruck(r.solldruck),  
    druck(r.druck) {}  
  
Auto(const Auto& a){  
    rvl = new Rad(*a.rvl);  
    rvr = new Rad(*a.rvr);  
    rhl = new Rad(*a.rhl);  
    rhr = new Rad(*a.rhr);  
}
```

Einführung in die Objektorientierung

Kopierkonstruktor

- Der explizite Copy Konstruktor ist auch immer dann notwendig, wenn in einer Klasse C Strings verwendet werden.
- Der Kopierkontruktor wird auch bei einem Funktionsaufruf mit Wertübergabe (call-by-value) und bei einer Funktionsrückgabe (return) automatisch aufgerufen.
- Achtung:** Er wird nicht bei einer Zuweisung aufgerufen! Dazu muss man einen überladenen Zuweisungsoperator definieren.
- Faustregel:** benötigt man einen Destruktor, dann benötigt man auch einen Kopierkonstruktor und einen überladenen Zuweisungsoperator.

Einführung in die Objektorientierung

this-Zeiger

- Jede Methode eines Objekts kann auf alle Teile (Attribute und Methoden) des Objekts zugreifen.
- Das Objekt selbst ist **implizites Argument** jeder Methode, d. h. es ist ein Parameter, obwohl es nicht ausdrücklich in der Parameterliste steht.

```
float Rad::berechne_geschwindigkeit() {
    return durchmesser * 3.14f * umdrehung;
}

Rad *rwl = new Rad(10,0.7,0,0);
cout << rvl->berechne_geschwindigkeit() << endl;
```

- Die Adresse des aktuellen Objekts (des impliziten Arguments) erhält man innerhalb der Methoden mit Hilfe des vordefinierten Zeigers this.

Einführung in die Objektorientierung

this-Zeiger

- Damit könnte man die Methode berechne_geschwindigkeit auch folgendermaßen definieren:

```
float Rad::berechne_geschwindigkeit() {
    return this->durchmesser * 3.14f * this->umdrehung;
}
```

- Das bietet sich auch bei den Setter-Methoden an, weil man dann das Attribut und den formalen Parameter gleich nennen kann:

```
void Rad::set_umdrehung(float umdrehung) {
    this->umdrehung = umdrehung;
}
```

Einführung in die Objektorientierung this-Zeiger

- Der this-Zeiger wird immer dann benötigt, wenn man das Objekt als Ganzes an eine Methode weitergeben möchte oder es als return-Wert zurückgeben möchte.
- Beispiel:

```
class Koordinaten {  
    public:  
        // Attribute  
        int x;  
        int y;  
        // Konstruktor  
        Koordinaten(int x, int y);  
        // Methoden  
        Koordinaten* add (Koordinaten* k);  
        void print (void);  
};
```

323

Einführung in die Objektorientierung this-Zeiger

```
Koordinaten* Koordinaten::add (Koordinaten *k) {  
    x = x+k->x;  
    y = y+k->y;  
    return this;  
}
```

- Dadurch kann man verkettete Methodenaufrufe in einem Term schreiben:

```
Koordinaten *k1 = new Koordinaten(1,2);  
Koordinaten *k2 = new Koordinaten(3,4);  
Koordinaten *k3 = new Koordinaten(3,4);  
  
k1->add(k2)->add(k3)->print();
```

324

Einführung in die Objektorientierung

Statische Attribute und Methoden

- Statische Attribute in Klassen sind Attribute die zur Klasse gehören und nicht zum Objekt.
- Sie existieren pro Klasse exakt einmal, egal wie viele Objekte dieser Klasse erzeugt werden.
- Sie haben für alle Objekte den gleichen Wert.
- Sie werden angelegt, bevor das erste Objekt der Klasse erzeugt wird. Deshalb kann ein Konstruktor der Klasse nicht zur Initialisierung verwendet werden.
- Statische Attributen werden in der Klasse durch ein vorangestelltes Schlüsselwort static deklariert:

```
class Rad {  
public:  
    static int anzahl;
```

325

Einführung in die Objektorientierung

Statische Attribute und Methoden

- Durch die Deklaration wird noch **keine Instanz** der statischen Attributs angelegt.
- Statische Attribute müssen deshalb wie globale Variablen außerhalb der Klasse in einer Quellcodedatei definiert werden.
- Der Zugriff auf das statische Attribut erfolgt analog zur externen Definition einer Methode mithilfe des Scope Resolution Operators.
- Statischer Attribute können weder im Konstruktor noch in der Klasse initialisiert werden. Sie sollten bei der Definition initialisiert werden.

```
#include <iostream>  
#include "rad.h"  
  
using namespace std;  
  
int Rad::anzahl=0;  
  
// Konstruktoren
```

326

Einführung in die Objektorientierung

Statische Attribute und Methoden

- Der Zugriff auf ein statisches Attribut sollte immer über den Klassennamen erfolgen, und nicht über einen Objektnamen (es ist aber beides möglich).

```
Rad::Rad(float u, float du, float sd, float dr) {
    cout << "Rad Konstruktor" << endl;
    anzahl++;
    umdrehung = u;
    durchmesser = du;
    soldruck = sd;
    druck = dr;
}

cout << "Anzahl der Raeder: " << Rad::anzahl << endl;
cout << "Anzahl der Raeder: " << rvl->anzahl << endl;
```

Einführung in die Objektorientierung

Statische Attribute und Methoden

- Auch statische Attribute sollten als private deklariert werden.
- Öffentliche Getter-Methoden für private, statische Attribute sollten als statisch Methoden deklariert werden.
- Statische Methoden sind die Schnittstelle zu den statischen Attributen der Klasse.
- Sie können auch ohne Objekt aufgerufen werden → können nur auf statische Attribute zugreifen und nicht auf nichtstatische Attribute (von Objekten).
- Sie gehören (wie auch die statischen Attribute) nicht zu bestimmten Objekten sondern zur Klasse.
- Merke:** Statische Elemente gehören also zur Klasse, nichtstatische Elemente gehören zu einem Objekt.

Einführung in die Objektorientierung

Statische Attribute und Methoden

```
class Rad {  
    private:  
        // Attribute  
        static int anzahl;  
        float umdrehung;  
        float durchmesser;  
        float solldruck;  
        float druck;  
    public:  
        // Konstruktoren  
        Rad();  
        Rad(float u, float du, float sd, float dr);  
        // Kopierkonstruktoren  
        Rad(const Rad& r);  
        // Destructor  
        ~Rad();  
        // Methoden  
        static int get_anzahl();  
        void set_umdrehung(float u);  
        ...  
  
        cout << "Anzahl der Raeder: " << Rad::get_anzahl() << endl;  
        cout << "Anzahl der Raeder: " << rvl->get_anzahl() << endl;
```

329

Einführung in die Objektorientierung

Konstante Methoden

- Funktionsparameter werden gern als konstant deklariert, um dem Aufrufer zu signalisieren, dass sie innerhalb der Funktion nicht verändert werden können.
- Ist der Parameter ein Zeiger auf ein konstantes Objekt, dann darf in der Funktion kein verändernder Zugriff auf das Objekt erfolgen, weder direkt noch mittelbar.

```
bool Rad::schneller(const Rad *r){  
    return r->berechne_geschwindigkeit() > this->berechne_geschwindigkeit();  
}
```

- Für den Compiler ist es so einfach nicht festzustellen, ob die Methode berechne_geschwindigkeit() das Objekt r ändert. → Übersetzungsfehler

330

Einführung in die Objektorientierung

Konstante Methoden

- Damit der Compiler sicher sein kann, dass eine Methode ein Datenelement nicht ändert, kann man sie als konstant deklarieren, indem man das Schlüsselwort const hinter die Parameterklammern der Methode setzt:

```
float Rad::berechne_geschwindigkeit() const {
    return durchmesser * 3.14f * umdrehung;
}
```

- **Merke:** Wenn eine Methode auf die Elemente der Klasse nur lesend zugreift, dann sollte sie immer als const deklariert werden.
- Dann kann mit dieser Methode auch auf konstante Objekte zugegriffen werden.

Einführung in die Objektorientierung

Friend Funktionen

- Man kann in einer Klasse eine Funktion als friend deklarieren.
- Dadurch wird dieser Funktion der Zugriff auf private Members gewährt, d.h. die Funktion kann z.B. direkt auf die privaten Attribute zugreifen.
- Die Funktion muss außerhalb der Klasse definiert werden. Sie ist kein Member der Klasse.

```
class Rad {
    friend void print_umdrehung(Rad *);    void print_umdrehung(Rad *r) {
    private:                                cout << r->umdrehung << endl;
        // Attribute
        static int anzahl;
        float umdrehung;
    }
```

- Mit dieser Deklaration sollte man sehr vorsichtig umgehen, weil es das Kapselungsprinzip untergräbt.

Einführung in die Objektorientierung

Friend Klassen

- Man kann in einer Klasse auch eine andere Klasse als friend deklarieren.
- Damit erhalten alle Methoden dieser Klasse den Zugriff auf alle Attribute der Klasse (auch private).

```
class Rad {           Auto::Auto(Rad *vl, Rad *vr, Rad *hl, Rad *hr) {  
    friend class Auto;  
    private:  
        // Attribute  
        static int anzahl;  
        float umdrehung;  
        float durchmesser;  
        float solldruck;  
        float druck;  
    public:  
        // Konstruktoren  
        ...  
        cout << "Auto Konstruktor" << endl;  
        rvl = new Rad(*vl);  
        rvl->umdrehung = rvl->umdrehung * 10;  
        rvr = new Rad(*vr);  
        rhl = new Rad(*hl);  
        rhr = new Rad(*hr);  
}
```

333

Einführung in die Objektorientierung

Vererbung

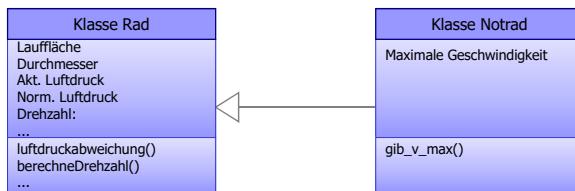
- Klassen können aus anderen Klassen abgeleitet werden.
- Abgeleitete Klassen
 - haben alle Methoden und Daten der Basisklassen
 - können weitere Daten und Methoden erhalten
 - können Methoden der Basisklasse überschreiben (verschatten).
- Typische Fälle von Vererbung
 - Eine Klasse ist Spezialisierung oder Erweiterung einer vorhandenen Klasse.
 - Eine Klasse vereint die Eigenschaften mehrerer vorhandener Klassen (Generalisierung).
 - Häufiger Fall: Aus mehreren parallel entwickelten Klassen kristallisiert sich ein gemeinsamer Kern hinsichtlich Funktion und Benutzung heraus.

334

Einführung in die Objektorientierung

Vererbung

- Manchmal möchte eine Klasse spezialisieren.
- Beispiel: Ein Notrad ist ein spezielles Rad. Es hat alle Eigenschaften eines Rades, aber noch zusätzlich speziellere, z.B. eine Maximalgeschwindigkeit.
- Es gilt: Jedes Notrad **ist ein** Rad aber nicht jedes Rad ist ein Notrad.
- Speziellere Eigenschaften heißt, dass ein Notrad mehr Attribute und Methoden hat. Es besitzt aber auch alle Attribute und Methoden eines Rades, d.h. die Attribute und Methoden werden **vererbt**.



335

Einführung in die Objektorientierung

Vererbung

- Das Rad wird jetzt als **Oberklasse** bezeichnet und das Notrad als **Unterklasse**.
- Das Erzeugen einer Unterklasse aus einer existierenden Klasse wird als **Ableitung** bezeichnet. Man spezialisiert eine Klasse. Die spezialisierte Klasse wird deshalb auch als **abgeleitete Klasse** bezeichnet, die ursprüngliche Klasse als **Basisklasse**.
- Das Erzeugen einer Oberklasse aus einer existierenden Klasse wird als **Generalisierung** bezeichnet.
- Ein Objekt einer Unterklasse kann immer dort verwendet werden, wo ein Objekt einer Oberklasse erwartet wird, aber nicht umgekehrt. D.h. ein Objekt der Klasse Notrad kann immer dort verwendet werden, wo ein Objekt der Klasse Rad erwartet wird, weil jedes Notrad ein Rad ist.

336

Einführung in die Objektorientierung

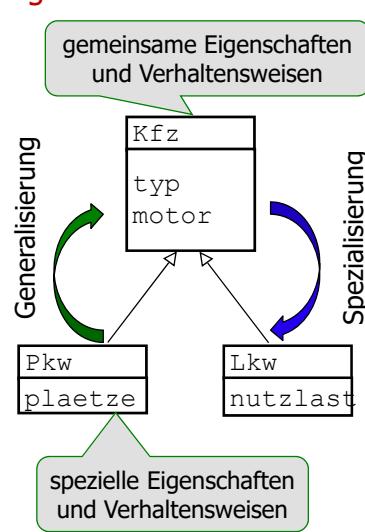
Vererbung

- Abgeleitete Klassen:
 - haben alle Attribute und Methoden der Basisklassen
 - können weitere Attribute und Methoden erhalten
 - können Methoden der Basisklasse überschreiben.
- Die Basisklasse wird oft auch **Ober-, Super- oder Elternklasse** genannt.
- Eine abgeleitete Klasse wird auch **Sub-, Unter- oder Kindklasse** genannt.
- Abgeleitete Klasse und Basisklasse stehen typischerweise in einer „**ist-ein**“-Beziehung zueinander.

Einführung in die Objektorientierung

Vererbung

- Ein Kfz hat
 - typ
 - motor
- PKW ist ein Kfz, hat zusätzlich Sitzplätze
- LKW ist ein Kfz, hat zusätzlich Nutzlast



Einführung in die Objektorientierung Vererbung

Abgeleitete Klasse kann auf Mitglieder der Basisklasse zugreifen

neue Klasse

```
#include "Notrad.h"
void Notrad::set_v_max(int v) {
    v_max = v;
}
int Notrad::get_v_max() {
    return v_max;
}
bool Notrad::check_v() {
    return this->berechne_geschwindigkeit() <= v_max;
}
```

```
#include "Rad.h"
class Notrad:public Rad {
private:
    int v_max;
public:
    void set_v_max(int v);
    int get_v_max();
    bool check_v();
};
```

Basisklasse

339

Einführung in die Objektorientierung Vererbung

```
#include <iostream>
#include "Notrad.h"

using namespace std;

int main(int argc, char *argv[]) {
    Notrad nr;
    Rad r;

    nr.set_v_max(50);
    cout << nr.get_v_max() << endl;
    cout << nr.berechne_geschwindigkeit() << endl;
    cout << nr.check_v() << endl;

    r = nr;      // Das geht
    nr = r;      // Das geht nicht

    return 0;
}
```

340

Einführung in die Objektorientierung

Vererbung

- Da öffentlich (public) abgeleitete Klassen die komplette Schnittstelle ihrer Basisklasse enthalten, sind sie zur Basisklasse zuweisungskompatibel.
- Ein Objekt der Klasse Notrad kann also einem Objekt der Klasse Rad zugewiesen werden.
- Nach der Zuweisung ist aber die Information über die v_max verloren.
- Das kopierte Objekt besitzt also die Erweiterungen der abgeleiteten Klasse nicht mehr.
- Einem Zeiger auf die Basisklasse kann die Adresse des Objekts einer abgeleiteten Klasse zugewiesen werden. In diesem Fall gehen dem Objekt keine Informationen verloren, weil es ja nicht verändert wird.
- Umgekehrt funktioniert das nicht. Einem Objekt einer abgeleiteten Klasse kann nicht das Objekt einer Basisklasse zugewiesen werden.

Einführung in die Objektorientierung

Vererbung und Zugriffsrechte

- Durch die public-Ableitung gelten die in der Basisklasse definierten Zugriffsrechte auch für die abgeleitete Klasse.
- Die abgeleitete Klasse kann nur auf die public-Elemente zugreifen. Ein Zugriff auf die private-Elemente der Basisklasse ist nur innerhalb der Methoden der Basisklasse möglich, jedoch nicht innerhalb der neuen Methoden der abgeleiteten Klasse.

```
void Notrad::print() {
    cout << "V-max: " << v_max << endl;
    cout << "Druck: " << druck << endl;
}
```

- Manchmal wäre es aber wünschenswert, dass dieser Zugriff möglich ist, ohne jedoch außerhalb der abgeleiteten Klasse zugreifen zu können.

Einführung in die Objektorientierung

Vererbung und Zugriffsrechte

- Zu diesem Zweck gibt es einen weiteren Schutzmechanismus **protected**.
- Dieser ist ein "Zwischending" zwischen public und private.
- protected bedeutet
 - private für den Zugriff von außen, d.h. durch den Anwender einer Klasse in der main-Funktion.
 - public für den Zugriff von innen, d. h. innerhalb einer Methode der abgeleiteten Klasse.

Zugriff	„von außen“	„von innen“
private	nicht möglich	nicht möglich
protected	nicht möglich	möglich
public	möglich	möglich

Einführung in die Objektorientierung

Vererbung und Zugriffsrechte

- Neben der public-Ableitung gibt es auch eine **private-Ableitung**.
- Öffentliche Elemente der Basisklasse werden durch privates Ableiten in der abgeleiteten Klasse privat, d.h. von außen kann bei abgeleiteten Objekten nicht mehr auf die Methoden der Basisklasse zugegriffen werden.
- Diese Art der Ableitung ist kaum nützlich.
- Wird aber keine Ableitungsart angegeben, so erfolgt eine private Ableitung.
- Bei einer privaten Ableitung können Objekte der abgeleiteten Klasse nicht mehr Objekten der Basisklasse zugewiesen werden. Sonst könnten die Schutzrechte ausgehoben werden.

Einführung in die Objektorientierung

Vererbung und Konstruktoren

- Definition eines Objektes heißt immer Konstruktorauftrag (ggfs. des automatischen Standardkonstruktors).
- Haben Objekte Basisklassen, so wird mit einem abgeleiteten Objekt gleichzeitig ein Objekt der Basisklasse erzeugt. Es müssen also für diese Basisklassen auch Konstruktoren aufgerufen werden.
- Bevor der Konstruktor einer abgeleiteten Klasse ausgeführt wird, wird immer zuerst der Standardkonstruktor der Basisklasse gestartet.
- Umgekehrt ist es beim Destruktor: Hier wird der Standarddestruktur der Basisklasse zuletzt aufgerufen.
- Dieses Verhalten ist logisch, da abgeleitete Klassen auf den Attributen und Eigenschaften der Basisklassen aufbauen.

Einführung in die Objektorientierung

Vererbung und Konstruktoren

- Hat die Basisklasse keinen Standardkonstruktor oder möchte man einen anderen Konstruktor verwenden, so muss der Konstruktorauftrag immer explizit in der **Initialisierungsliste** des Konstruktors der abgeleiteten Klasse erfolgen.
- Initialisierung von Basisklassenelementen ist nicht durch direkte Angabe dieser Elemente in der Initialisierungsliste möglich.

```
Notrad::Notrad() {
    std::cout << "Notrad: Defaultkonstruktor" << std::endl;
    v_max = 0;
}

Notrad::Notrad(float u, float du, float sd, float dr, int v) : Rad(u,du,sd,dr) {
    std::cout << "Notrad: Konstruktor" << std::endl;
    v_max = v;
}
```

Einführung in die Objektorientierung

Überschreiben von Methoden

- In C++ können in einer abgeleiteten Klasse die Methoden der Basisklasse **überschrieben** werden, d.h. der Name der Methode bleibt gleich, aber die Implementierung ändert sich.
- Jedes Objekt einer abgeleiteten Klasse auch aber auch ein Objekt der Basisklasse. Dies wird als **Inklusionspolymorphie** bezeichnet.
- Es kann also verschiedene „Gestalten“ annehmen: Abhängig von der Gestalt ändert sich aber durch das Überschreiben das Verhalten des Objekts.
- Die Entscheidung darüber welche Gestalt es annimmt, d.h. die Bindung eines Objekts, wird entweder **statisch** zur Übersetzung oder **dynamisch** zur Laufzeit entschieden.
- Der Standard in C++ ist die statische Bindung, d.h. es wird zur Übersetzungszeit festgelegt, welche Methode aufgerufen wird.
- Es wird immer die speziellste Klasse verwendet.

Einführung in die Objektorientierung

Überschreiben von Methoden

```
void Rad::print() {
    cout << "Durchmesser: " << durchmesser << endl;
    cout << "Umdrehung: " << umdrehung << endl;
    cout << "Solldruck: " << solldruck << endl;
    cout << "Druck: " << druck << endl;
}

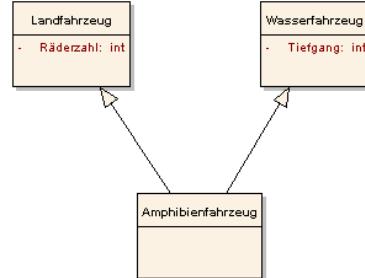
void Notrad::print() {
    Rad::print();
    std::cout << "V-max: " << v_max << std::endl;
}

int main(int argc, char *argv[]) {
    Notrad nr(1,2,3,4,5);
    Rad r;
    r = nr;
    nr.print();
    r.print();
```

Einführung in die Objektorientierung

Mehrfachvererbung

- Um Mehrfachvererbung handelt es sich, wenn eine abgeleitete Klasse direkt von **mehr** als einer Basisklasse erbt.
- Eine abgeleitete Klasse besitzt alle Komponenten seiner Basisklassen und kann weitere eigene Komponenten besitzen.
- die "ist-ein"-Beziehung muss für alle Basisklassen gelten!
- absolute Notwendigkeit ist umstritten
 - nicht alle objektorientierten Sprachen unterstützen Mehrfachvererbung
 - Java z. B. kennt keine Mehrfachvererbung

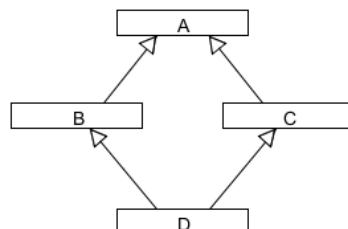


349

Einführung in die Objektorientierung

Mehrfachvererbung und das Diamond-Problem

- Das **Diamond-Problem** entsteht durch Mehrfachvererbung in der Objektorientierten Programmierung.
- Es kann auftreten, wenn eine Klasse D auf zwei verschiedenen Vererbungspfaden (B und C) von ein und derselben Basisklasse A abstammt.
- Zeichnet man die Vererbungsbeziehungen zwischen den Klassen als Diagramm, so ergibt sich die Form einer Raute (englisch rhombus oder diamond), nach der das Diamond-Problem benannt ist.



350

Einführung in die Objektorientierung Mehrfachvererbung und das Diamond-Problem

Beim Diamond-Problem wirft folgende Fragen auf:

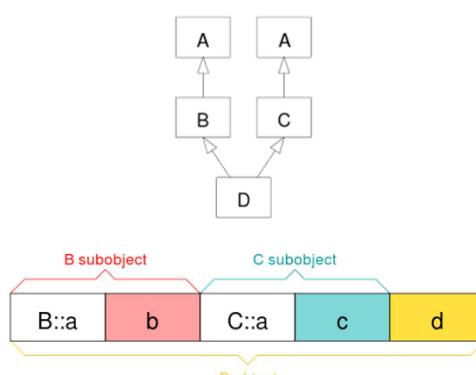
- Haben die Objekte der Klasse **D** die Attribute von **A** zweimal?
- Wenn die Klassen **B** und **C** eine Methode von **A** überladen, welche wird dann bei Objekten der Klasse **D** aufgerufen?
- Es gibt bei der Modellierung oft Fälle, wo es Sinn macht, dass sich die Objekte von **D** zwar die Attribute von **A** teilen, die Objekte von **D** jedoch verschiedene Überladungen der Methoden von **A** besitzen.

In C++ kann bei der Definition der Klassen **B** und **C** gewählt werden, ob sie sich eine gemeinsame Instanz der Klasse **A** teilen sollen (Diamond), oder ob sie jeweils ihre eigene Instanz besitzen sollen (normale Mehrfachvererbung).

Einführung in die Objektorientierung Mehrfachvererbung und das Diamond-Problem

Normale Mehrfachvererbung:

```
class A {  
    int a;  
};  
  
class B: A {  
    int b;  
};  
  
class C: A {  
    int c;  
};  
  
class D: B, C {  
    int d;  
};
```

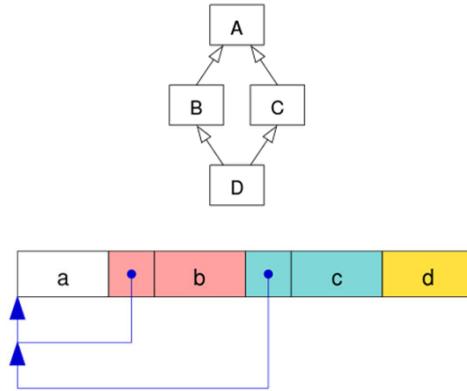


Die Klassen **B** und **C** haben jeweils ihre eigenen Kopien der Member der Oberklasse **A** und haben Zugriff auf zwei unterschiedliche Variablen **a** der Oberklassen **B** und **C**.

Einführung in die Objektorientierung Mehrfachvererbung und das Diamond-Problem

Diamond Mehrfachvererbung:

```
class A {  
    int a;  
};  
  
class B: virtual A {  
    int b;  
};  
  
class C: virtual A {  
    int c;  
};  
  
class D: B, C {  
    int d;  
};
```



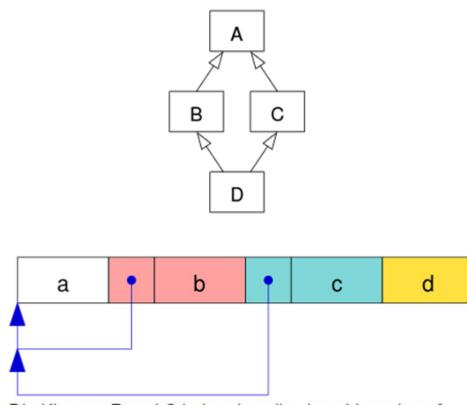
Die Klassen **B** und **C** haben jeweils einen Verweis auf Member der Oberklasse **A** und haben Zugriff auf ein und dieselbe Variable **a** der Oberklasse **A**.

353

Einführung in die Objektorientierung Mehrfachvererbung und das Diamond-Problem

Diamond Mehrfachvererbung:

```
class A {  
    int a;  
};  
  
class B: virtual A {  
    int b;  
};  
  
class C: virtual A {  
    int c;  
};  
  
class D: B, C {  
    int d;  
};
```



Die Klassen **B** und **C** haben jeweils einen Verweis auf Member der Oberklasse **A** und haben Zugriff auf ein und dieselbe Variable **a** der Oberklasse **A**.

354

Einführung in die Objektorientierung Mehrfachvererbung und das Diamond-Problem

Die bei der normalen Mehrfachvererbung auftretende Mehrdeutigkeit muss aufgelöst werden, bei der Diamond-Vererbung gibt es diese nicht:

```
int main() {
    D d;
    // d hat 2 Variablen namens a
    d.B::a = 1;
    d.C::a = 2;
    cout << d.B::a << " " << d.C::a << endl;
}

int main() {
    D d;
    // d hat nur 1 Variable namens a
    d.a = 3;
    cout << d.a << endl;
    // folgende Anweisungen greifen auf das gleiche a zu:
    d.B::a = 1;
    d.C::a = 2;
    cout << d.B::a << " " << d.C::a << endl;
}
```

355

C Funktionen Funktionen mit nicht-definiertem Ergebnis

Definition: partiell

- Eine Funktion heißt **partiell**, wenn sie nicht auf allen Eingabeelementen definiert ist.
- Undefiniertes Ergebnis wird semantisch ausgedrückt durch Zeichen \perp (bottom).

Beispiel

- Ganzzahlige Division div: unsigned x unsigned → unsigned
- Undefiniert bei 0 als Divisor: $\text{div}(x, 0) = \perp$

356

C Funktionen

Funktionen mit nicht-definiertem Ergebnis

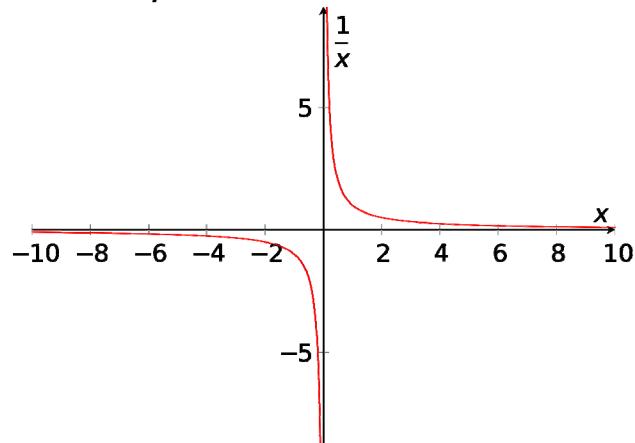
```
unsigned div (unsigned x, unsigned y) {  
    if (x < y) {  
        return 0;  
    }  
    else {  
        return 1+div(x-y,y);  
    }  
}
```

357

C Funktionen

Funktionen mit nicht-definiertem Ergebnis

Graph der Funktion $1/x$:



358

C Funktionen

Funktionen mit nicht-definiertem Ergebnis

Definition: strikt

- Eine Funktion heißt **strikt**, wenn sie immer dann ein undefiniertes Ergebnis liefert, wenn irgendein Argument undefiniert ist.
- Eine Funktion, die trotz eines undefinierten Arguments ein definiertes Ergebnis liefern kann, heißt **nicht strikt**.

Bemerkung

- Striktheit garantiert, dass jeder Fehler bei der Anwendung der Funktion durchschlägt.
- Nichtstrikte Funktionen haben in gewisser Weise heilende Eigenschaften (Fehler können abgefangen werden).
- In C sind alle selbst definierten Funktionen strikt.

C Funktionen

Funktionen mit nicht-definiertem Ergebnis

Unterschiedliche Aufrufstrategien:

- **Call by value, eager evaluation**
 - Beim Funktionsaufruf werden zunächst **alle** Argumente ausgewertet.
 - Semantik des Funktionsaufrufes ist somit strikt!
 - Die meisten gängigen Programmiersprachen (auch C/C++/Java) haben ein „Call by value“ Semantik.
- **Call by need, lazy evaluation**
 - Nur die im Rumpf tatsächlich benötigten Argumente werden ausgewertet.
 - Semantik des Funktionsaufrufs ist i. A. nicht strikt!
 - Beispiele: LazyML, Haskell

C Funktionen

Funktionen mit nicht-definiertem Ergebnis

```
unsigned const (unsigned x) {  
    return 0;  
}
```

Semantische Varianten bei der Auswertung

- Call by value, strikt: $\text{const}(\text{div}(1,0)) = \perp$
- Call by need, nicht strikt: $\text{const}(\text{div}(1,0)) = 0$

Grund: Parameter wird im Rumpf gar nicht verwendet!

C Funktionen

Funktionen mit nicht-definiertem Ergebnis

Nicht-strikte Funktionen in C

- Der bedingte Ausdruck `_?:_` ist nur strikt im ersten Argument
- `x==0?0:1/x` liefert das Ergebnis 0!
- Die Booleschen Funktionen `&&` und `||` sind nur strikt im ersten Argument
- `0 == 1 && 1/0 == 0` liefert meist den wert 0 (false). Der rechte Ausdruck von `&&` wird nicht mehr ausgewertet, weil die Aussage schon durch den linken Ausdruck falsch ist.
- `0 == 0 || 1/0 == 0` liefert meist den wert 1 (true). Der rechte Ausdruck von `||` wird nicht mehr ausgewertet, weil die Aussage schon durch den linken Ausdruck wahr ist.
- **Achtung:** Das Verhalten ist abhängig vom Compiler und den Einstellungen

C Funktionen

Funktionen mit nicht-definiertem Ergebnis

Kurzschlussauswertung

- Kurzschlussauswertung (auch bedingte Auswertung, englisch short-circuit evaluation) bezeichnet eine Strategie der Auswertung von Booleschen Ausdrücken.
- Im Allgemeinen steht das Ergebnis eines Booleschen Ausdrucks ohne die Verwendung von Kurzschlussauswertung erst nach der Auswertung aller Teilausdrücke fest.
- Kurzschlussauswertung ermöglicht das vorzeitige Abbrechen einer Auswertung eines Booleschen Ausdrucks, sobald das Auswertungsergebnis durch einen Teilausdruck eindeutig bestimmt ist.