Die Programmiersprache C

Historie

Entwicklung der Programmiersprache C:

- ist eng mit der Entwicklung des Betriebssystems UNIX verbunden
- UNIX und die meisten Programme, die damit eingesetzt werden, sind in C geschrieben
- C ist nicht von einem bestimmten Betriebssystem oder einer bestimmten Maschine abhängig

wesentliche Entwicklungsarbeiten zu UNIX und C: Anfang der 70er Jahre von Ken Thompson und Dennis Ritchie an den Bell Laboratorien

C basiert auf den Sprachen BCPL und B (beide typenlos)

Erste Schritte

Datentypen in C:

- Zeichen: `char`
- ganze Zahlen: `int`, `short int`, `long int`
- Gleitpunktzahlen: `float`, `double`

abgeleitete Datentypen können erzeugt werden mit:

- Zeigern
- Vektoren (auch: Array oder Felder)
- Strukturen
- Vereinigungen

Zeiger erlauben maschinenunabhängige Adress-Arithmetik.

Erste Schritte (2)

Das folgende Programm enthält Beispiele für die meisten Grundelemente, aus denen die Sprache C aufgebaut ist.

„Hello, world!“ auf dem Bildschirm ausgeben:

```
/* Einfügen von Standard-Bibliotheken */
#include <stdio.h>
#include <stdlib.h>

/* Durch main wird das Hauptprogramm eingeleitet */
main() {
    char *str = "world";
    printf("Hello, %s!\n", str);
    exit(0);
}
```

Erste Schritte (3)

- Durch `/*` und `*/` werden Kommentare eingeschlossen.
- Mittels `#include <stdio.h>` wird eine Bibliothek bereitgestellt, die Funktionen zur Ein- und Ausgabe enthält.
- Der Start eines Programms besteht im Ausführen der Funktion `main`.
- `char *str = "world"` definiert eine Variable `str`. Alle in einem C-Programm benutzten Variablen müssen explizit deklariert werden, wobei der Typ und der Name der Variablen festgelegt werden.
- Die Funktion `printf()` gibt eine Zeichenkette auf dem Bildschirm aus. Solche Standardfunktionen sind übersetzte Funktionen, die zur C-Implementierung gehören.

Erste Schritte (4)

- Mit `exit(0)` wird das Programm verlassen und der Wert 0 an den Kommandointerpreter zurückgegeben.
- Alle Anweisungen werden mit einem Semikolon beendet.
- Anweisungsfolgen werden mit geschweiften Klammern zusammengefasst, der geklammerte Block gilt als eine Anweisung.

Kriterien bei der Programmentwicklung

- Korrektheit bzgl. der Aufgabenstellung
- Handhabung, Benutzerführung, Online-Hilfe
- Technische Qualität
  * Laufzeit
  * Verbrauch von Ressourcen (Speicherplatz)
- Wirtschaftliche Qualität
  * Wartbarkeit
  * Erweiterbarkeit
  * Entwicklungskosten

Software-Qualität in ISO 9126 beschrieben

Programmieren

- **Editieren:**
  * C-Programm mit beliebigem Text-Editor eingeben
  * Programm kann aus mehreren Quelltext-Dateien bestehen (siehe „Modulare Programmierung“)
- **Übersetzen:**
  * Bei jeder Übersetzung wird zunächst automatisch ein **Präprozessor** aufgerufen, der eine Vorverarbeitung des Textes vornimmt → `#include <stdio.h>` ersetzen
  * Enthält der Quelltext keine Syntaxfehler, erzeugt der C-Compiler eine **Objektdatei**, auch Modul genannt.
  * Objektdatei: enthält Maschinencode und zusätzliche Informationen für das Binden.

Programmieren (2)

- **Binden:**
  * Die vom Compiler erzeugten Objektdateien werden durch den **Linker** zu einem lauffähigen Programm gebunden.
  * Beispiel: Verwendete Standardbibliotheken und das Programm werden zu einem ausführbaren Programm zusammengefügt.
- **Ausführen:** Das übersetzte und vollständig gebundene Programm kann ausgeführt und getestet werden.

Wie diese Schritte auszuführen sind, hängt sowohl vom Betriebssystem als auch von der C-Implementierung ab.

Vom Programm zur Maschine

Programme, in einer höheren Programmiersprache, müssen in eine Folge von Maschinenbefehlen übersetzt werden.

Die Übersetzung eines Programmtextes in eine Folge von Maschinenbefehlen wird vom **Compiler** durchgeführt.

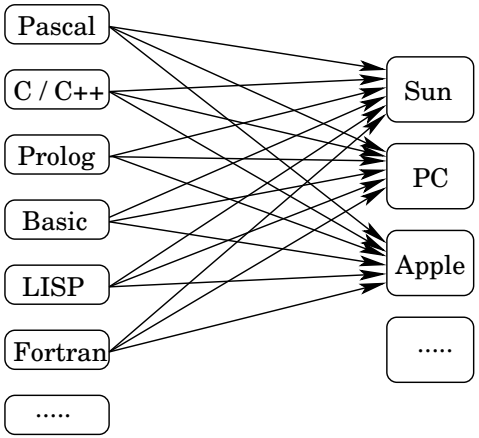
Das Ergebnis ist ein **Maschinenprogramm**, das in einer als **ausführbar** (engl. **executable**) gekennzeichneten Datei gespeichert ist.

Eine ausführbare Datei muss von einem **Ladeprogramm** in den Speicher geladen werden, um ausgeführt zu werden.

Ladeprogramme sind Teil des Betriebssystems, der Benutzer weiß in der Regel gar nichts von deren Existenz.

Vom Programm zur Maschine (2)

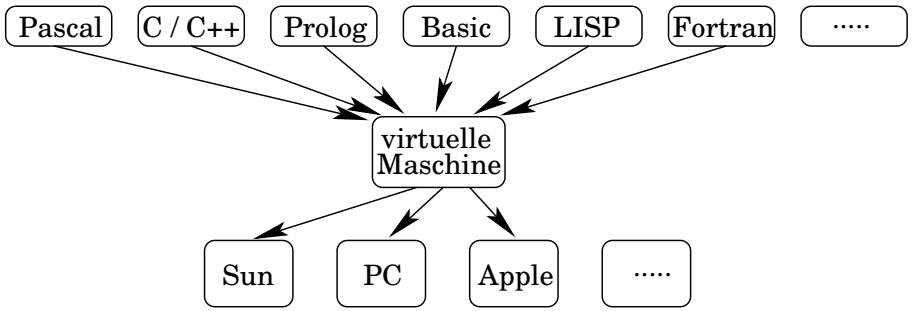
n Sprachen und m Rechnertypen $\Rightarrow$ $n \cdot m$ Compiler



Virtuelle Maschinen

Es würden $n + m$ Compiler ausreichen, wenn

- Code für eine virtuelle Maschine erzeugt wird, und
- die virtuelle Maschine auf allen konkreten Maschinen emuliert (in Software nachgebildet) wird.



Virtuelle Maschinen (2)

Probleme:

- Einige Sprachen oder Maschinentypen könnten bevorzugt werden.
- Geschwindigkeit bei der Programmausführung wird beeinträchtigt.

Für Java und C# sind virtuelle Maschinen auf verschiedenen Plattformen realisiert.

Compiler-Entwickler arbeiten mit **Zwischensprachen**, um unabhängig von der Quellsprache zu sein. Problem: Die Zwischensprache muss hinreichend allgemein sein (näheres dazu in der Vorlesung *Compilerbau* bei Prof. Dr. Becker).

Programmieren und Testen

Fehler, die während des Übersetzungsvorganges erkannt werden:

- **Syntaxfehler:** ist Rechtschreib- oder Grammatikfehlern vergleichbar (vertippt, falscher Satzbau)
- **Typfehler:** wenn nicht zueinander passende Dinge verknüpft werden (addieren des Straßennamens auf die Hausnummer)

Wurde das Programm fehlerlos übersetzt, kann es ausgeführt und getestet werden.

Programmieren und Testen (2)

Fehler, die nicht durch den Compiler erkannt werden:

- **Laufzeitfehler**
  - zulässige Wertebereiche werden überschritten
  - es wird durch 0 dividiert
  - es wird die Wurzel aus einer negativen Zahl gezogen
- **Denkfehler** werden sichtbar, wenn das Programm nicht das tut, was es tun soll.

Testen zeigt nur die Anwesenheit von Fehlern, nicht deren Abwesenheit!

Funktionalität hat höchste Priorität
⇒ **Tests sind immer durchzuführen**

Vom Problem zum Algorithmus

Ein Algorithmus ist eine präzise Vorschrift, um

- aus vorgegebenen Eingaben
- in endlich vielen Schritten
- eine bestimmte Ausgabe zu ermitteln.

(Abu Jáfar Hohammed ibu Musa al-Chowarizmi)

Programmieren ist das Umsetzen eines Algorithmus in eine für den Computer verständliche und ausführbare Form.

Problem ⟶ Algorithmus ⟶ Programm

Erst denken, dann programmieren!

Top-down Entwurf
Sortieren durch Einfügen (Insertion Sort)

Annahme:

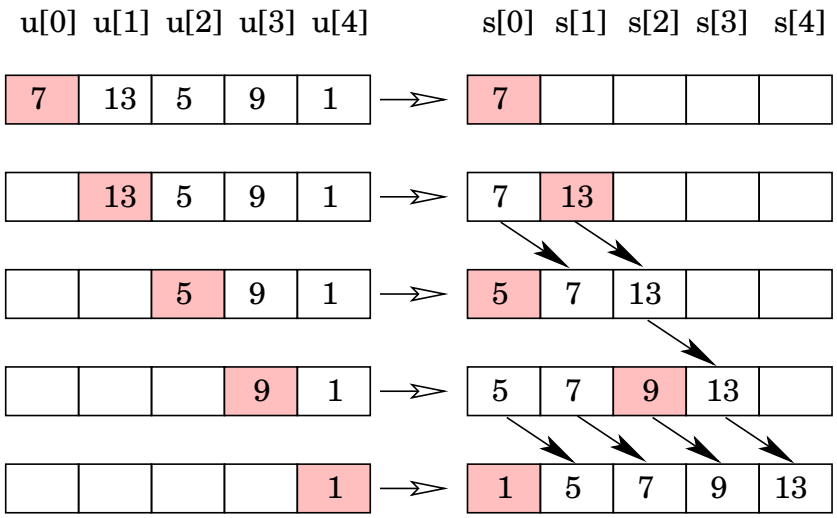
- Objekte sind im Array $u[0] \ldots u[n-1]$ gespeichert.
- Schlüssel stehen in $u[i].key$, Informationen in $u[i].data$.
- sortierte Objekte stehen hinterher in $s[0] \ldots s[n-1]$.

Algorithmus:

for $i := 0$ to $n-1$ step 1 do
    füge $u[i]$ am richtigen Platz in $s[0] \ldots s[i]$ ein

Top-down Entwurf (2)

Insertion Sort: Prinzip



Top-down Entwurf (3)

Das Einfügen von $u[i]$ in die Folge $s[0] \ldots s[i]$ muss genauer spezifiziert werden → **schrittweise verfeinern**

- suche die Stelle p, an der eingefügt werden soll
- verschiebe Zahlen ab Position p eine Stelle nach rechts
- füge $u[i]$ an der Position p ins Array ein

Top-down Entwurf (4)

for $i := 0$ to $n-1$ step 1 do

    ** suche die Stelle p, an der eingefügt werden soll **
    $p := 0$
    while $(u[i].key > s[p].key)$ and $(p < i)$ do
        $p := p + 1$

    ** verschiebe Zahlen ab Position p nach rechts **
    for $j := i - 1$ down to p step 1 do
        $s[j+1] := s[j]$

    ** füge Objekt an Position p ins Array ein **
    $s[p] := u[i]$

Top-down Entwurf (5)

Anmerkungen:

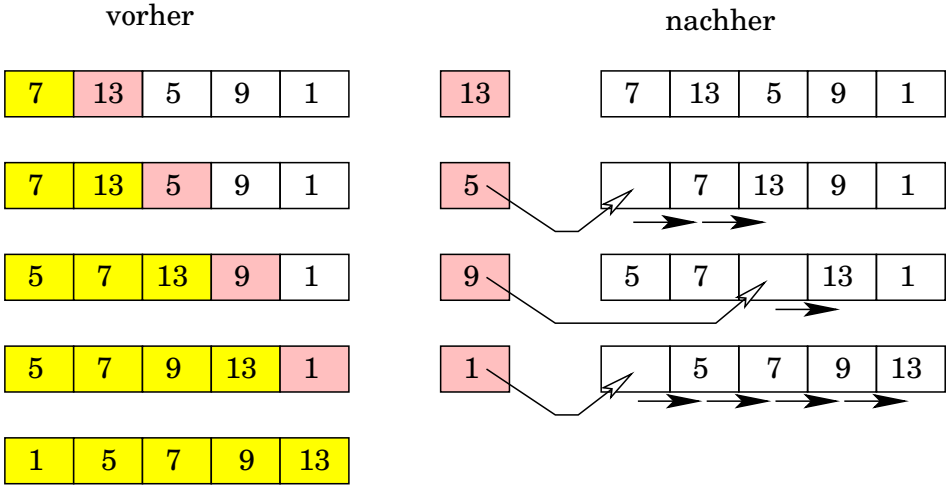
- Elemente in u und $s \rightarrow$ Speicherplatzverschwendung
- Verschieben der Zahlen kostet viel Zeit
- zusammenfassen: Position suchen/Zahlen verschieben

Insertion Sort Variante: am Ort sortieren

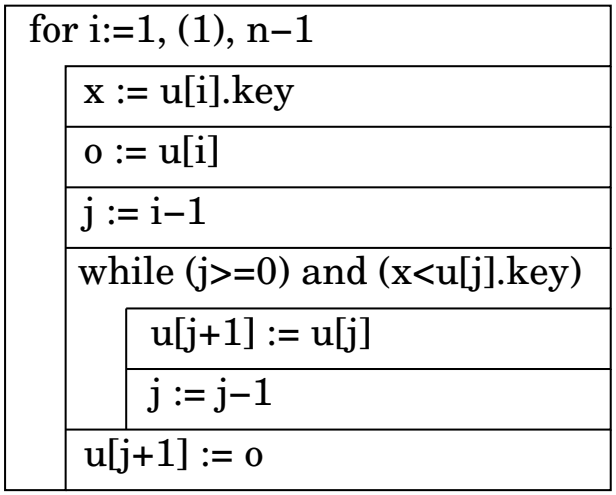
for $i := 1$ to $n-1$ step 1 do
  $x := u[i].key$
  $o := u[i]$
  $j := i-1$
  while $(j \geq 0)$ and $(x < u[j].key)$ do
    $u[j+1] := u[j]$
    $j := j-1$
  $u[j+1] := o$

Top-down Entwurf (6)

Insertion Sort Variante: Prinzip



Struktogramm: Insertion Sort



Sortieren: Hauptprogramm

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

int liste[50];

main() {
    int numberOfElements;

    numberOfElements = getData();
    insertionSort(numberOfElements);
    dataToScreen(numberOfElements);

    exit(0);
}
```

Sortieren: Daten einlesen

```
int getData() {
    int i, n;

    printf("Insertion Sort\n");
    printf("Wieviele Zahlen? (maximal 50)\n");
    scanf("%d", &n);

    for (i = 0; i < n; i++) {
        printf("%2d. Zahl eingeben: ", i + 1);
        scanf("%d", &liste[i]);
    }
    return n;
}
```


Sortieren: Sortierprozedur

```
void insertionSort(int n) {
    int i, j, x;
    for (i = 1; i < n; i++) {
        x = liste[i];
        j = i - 1;
        while ((j >= 0) && (x < liste[j])) {
            liste[j + 1] = liste[j];
            j = j - 1;
        }
        liste[j + 1] = x;
    }
    return;
}
```


Sortieren: Daten ausgeben

```
void dataToScreen(int n) {
    int i;

    printf("\nSortierte Zahlen\n");
    for (i = 0; i < n; i++) {
        printf("%2d. Zahl: %d\n", i + 1, liste[i]);
    }
}
```


Top-down Entwurf
Berechnung von Pi

$$\frac{\text{Viertelkreisfläche}}{\text{Quadratfläche}} = \frac{\text{Treffer im Viertelkreis}}{\text{Treffer im Quadrat}}$$

mit $r = 1$ und $A_{Kreis} = r^2 \cdot \pi$ ergibt sich:

$$\pi = 4 \cdot \frac{\text{Treffer im Viertelkreis}}{\text{Treffer im Quadrat}}$$

Algorithmus

treffer := 0
wiederhole n mal
    erzeuge Zufallszahlen x, y zwischen 0 und 1
    falls Punkt (x,y) innerhalb vom Viertelkreis
    dann treffer := treffer + 1
pi = 4 * treffer / n

Top-down Entwurf (2)

schrittweise verfeinern

- wiederhole n mal
  `for (i = 0; i < n; i++)`
- erzeuge Zufallszahlen x, y zwischen 0 und 1

```
x = 1.0 * rand() / RAND_MAX;
y = 1.0 * rand() / RAND_MAX;
```

- falls Punkt (x,y) innerhalb vom Viertelkreis
  `if (sqrt(x*x + y*y) < 1)`

Top-down Entwurf (3)

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

int main(void) {
    int i;
    int n = 100000;
    int treffer = 0;
    double x, y;
    double pi;

    srand(0);
```


Top-down Entwurf (4)

```
    for (i = 0; i < n; i++) {
        x = 1.0 * rand() / RAND_MAX;
        y = 1.0 * rand() / RAND_MAX;
        if (sqrt(x*x + y*y) <= 1)
            treffer += 1;
    }
    pi = 4.0 * treffer / n;
    printf("pi = %8.6f\n", pi);

    return 0;
}
```


Variablen

Um sinnvolle C-Programme schreiben zu können, ist es notwendig,

- Zwischenwerte zu speichern und
- diese Werte in weiteren Berechnungen zu verwenden.

Für die Speicherung der Werte steht der Hauptspeicher zur Verfügung. Ohne höhere Programmiersprachen:

- An welcher Stelle im Speicher steht der Wert?
- Wie viele Bytes gehören zu dem Wert?
- Welche Speicherplätze sind noch frei? ...

Variablen sind Behälter für Werte eines bestimmten Datentyps. Der Compiler setzt jeden Bezug auf eine Variable in die entsprechende Hauptspeicheradresse um.

Grundelemente der Sprache

Variablen und **Konstanten**: grundsätzliche Datenobjekte, die ein Programm manipuliert

Vereinbarungen:

- führen Variablen ein, die benutzt werden dürfen
- legen den Typ der Variablen fest und ggf. den Anfangswert

Operatoren kontrollieren, was mit den Werten geschieht.

In **Ausdrücken** werden Variablen und Konstanten mit Operatoren verknüpft, um neue Werte zu produzieren.

Der **Datentyp** eines Objekts legt seine Wertemenge und die Operatoren fest, die darauf anwendbar sind.

Zeichensatz

Folgende Zeichen sind in C-Programmen zulässig:

- Alphanumerische Zeichen:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
```

- Leerzeichen und Zeilenendezeichen
- Sonderzeichen:

```
( ) [ ] { } < >  + - * / % ^ ~ & | _
= ! ? # \ , . ; : ’ "
```


Zeichensatz (2)

- Steuerzeichen:
  - `\a` Gong-Zeichen (Bell)
  - `\b` ein Zeichen zurück (Backspace)
  - `\t` horizontaler Tabulator (Tab)
  - `\f` Seitenvorschub (Formfeed)
  - `\r` Wagenrücklauf (Carriage Return)
  - `\n` Zeilentrenner (New Line)
  - `\"` doppelte Anführungsstriche
  - `\’` einfacher Anführungsstrich
  - `\\` Backslash

Schlüsselwörter

Die folgenden Wörter haben eine vordefinierte Bedeutung:

`auto`	`double`	`int`	`struct`
`break`	`else`	`long`	`switch`
`case`	`enum`	`register`	`typedef`
`char`	`extern`	`return`	`union`
`const`	`float`	`short`	`unsigned`
`continue`	`for`	`signed`	`void`
`default`	`goto`	`sizeof`	`volatile`
`do`	`if`	`static`	`while`

Bezeichner und Namen

Bezeichner dienen zur **eindeutigen Identifizierung** von Objekten innerhalb eines Programms.

Ein Objekt in C ist ein Speicherbereich, der aus einer zusammenhängenden Folge von einem oder mehreren Bytes bestehen muss. (In OOP ist der Begriff eines Objekts anders definiert!)

Funktionsnamen, Variablennamen und andere Namen (z.B. Sprungmarken) sind Folgen von Zeichen, bestehend aus

- Buchstaben,
- Ziffern und
- dem Unterstrich.

Groß-/Kleinschreibung wird unterschieden.

Bezeichner und Namen (2)

Eigenschaften:

- Ein Bezeichner beginnt immer mit einem Buchstaben oder einem Unterstrich. Namen vermeiden, die mit zwei Unterstrichen beginnen → evtl. systemintern benutzt!
- Bezeichner dürfen nicht mit Schlüsselwörtern wie `int`, `while` oder `if` übereinstimmen.
- Die Anzahl der Zeichen in Bezeichnern ist in einigen Implementierungen begrenzt. Der Standard definiert für
  * interne Namen: die ersten 31 Zeichen sind signifikant
  * externe Namen: mind. 6 Zeichen werden unterschieden, jedoch nicht notwendig Groß-/Kleinschreibung

Bezeichner und Namen (3)

Anmerkung: Benennen Sie Variablen so,

- dass der Name den Zweck der Variablen andeutet, und
- eine Verwechselung durch Tippfehler unwahrscheinlich ist.

Datentypen

C ist eine **typisierte** Programmiersprache: Alle in einem C-Programm verwendeten Größen (Konstanten, Variablen, Funktionen) haben einen Typ (im Gegensatz zu Skript-Sprachen wie Perl).

Datentypen (2)

Der Typ von Konstanten (`#define ...`) ergibt sich in der Regel aus deren Wert.

Die Typen von Variablen und Funktionen werden in deren Deklaration bzw. Definition festgelegt.

Vorteil: Da der Compiler Unverträglichkeiten von Typen erkennen kann, werden durch die Typisierung Fehlerquellen verringert.

Die Größen der primitiven Datentypen sind nicht festgelegt, sie sind abhängig von der C-Implementierung und von der Wortlänge des Rechners
→ führt zu Problemen bei Portierungen, anders in Java

Elementare Datentypen

Der **Integer-Typ** `int` stellt ganzzahlige Werte mit/ohne Vorzeichen dar, die üblichen arithmetischen Operationen sind definiert.

- Typen: `short`, `int` und `long`.
- Versionen: `signed` und `unsigned`.
- Größen: `short` $\geq$ 16 Bit, `int` $\geq$ 16 Bit, `long` $\geq$ 32 Bit.

Der **Character-Typ** `char` wird zur Darstellung von einzelnen Zeichen bzw. Zeichenketten (Strings) verwendet.

Der **Fließkommatyp** `float` stellt reelle Zahlen dar.

- `float`: einfach-genaue Fließkommawerte
- `double`: mindestens einfach-genaue Fließkommawerte
- `long double`: mindestens so groß wie `double`

Elementare Datentypen (2)

Der **leere Typ** `void` ist ein besonderer Typ, der keinen Wert hat und auf dem keine Operationen definiert sind. Er steht für die leere Menge.

Bei Funktionen wird mittels `void` definiert, dass kein Rückgabewert geliefert wird (Prozedur) bzw. keine Parameter übergeben werden.

Durch Gruppierungen wie Felder (array), Verbund (union) und Strukturierung (struct) können neue Typen gebildet werden.

Character-Typ

Der Datentyp `char` hat die Größe ein Byte und kann ein Zeichen aus dem Zeichensatz der Maschine speichern.

Der Wert einer Zeichenkonstanten ist der numerische Wert des Zeichens im Zeichensatz der Maschine (bspw. ASCII).

Zeichen sind ganzzahlig, arithmetische Operationen sind definiert. **Beispiel:** `'a' + 'b' = Ã` ($\rightarrow 97 + 98 = 195$)

Schreibweise:

- ein Zeichen innerhalb von einfachen Anführungszeichen
- Ersatzdarstellungen für Steuerzeichen (`\n`, `\t`, ...)
- oktale Darstellung `'\ooo'`: ein bis drei oktale Ziffern
- hexadezimal `'\xnn'`: ein oder zwei Hex-Ziffern

Character-Typ (2)

Beispiele:

```
/* 'X' in ASCII  */
char c1 = 'X', c2 = '\130', c3 = '\x58';
/* Sonderzeichen */
char c4 = '\"', c5 = '\\', c6 = '\'';
/* Steuerzeichen */
char c7 = '\n', c8 = '\t', c9 = '\r';
```

konstante Zeichenkette: Eine Folge von beliebig vielen Zeichen, die von doppelten Anführungszeichen umgeben ist.

```
"Eine konstante Zeichenkette"
```

Character-Typ (3)

Konstante Zeichenketten können aneinandergehängt werden, um sie im Programm auf mehrere Zeilen zu verteilen:

```
"Eine " "konstante" " Zeichenkette"
```

Eine Zeichenkette ist ein Vektor (oder Array) von Zeichen. Intern hat jede Zeichenkette am Ende ein Null-Zeichen (`'\0'`), wodurch die Länge prinzipiell nicht begrenzt ist.

Integer-Typ

Schreibweise:

- Buchstabe l oder L am Ende bedeutet `long`
- Buchstabe u oder U am Ende bedeutet `unsigned`
- Ziffer 0 am Anfang bedeutet oktal
- Zeichen 0x oder 0X am Anfang bedeuten hexadezimal

Beispiele:

- `123456789UL`
- `022` bedeutet $(22)_8 = (18)_{10}$
- `0x1F` bedeutet $(1F)_{16} = (31)_{10}$

Fließkomma-Typ

Schreibweise:

- Suffix f oder F vereinbart `float`
- Suffix l oder L vereinbart `long double`
- Dezimalpunkt und/oder Exponent vorhanden

Beispiele:

- 1e-3 $= 1 \cdot 10^{-3} = 0,001$
- 234E4 $= 234 \cdot 10^4 = 2.340.000$
- -12.34e-67F
- 234.456
- .123e-42

Vereinbarungen

Alle Variablen müssen vor Gebrauch vereinbart (deklariert) werden. Eine Vereinbarung gibt einen Typ an und enthält eine Liste von einer oder mehreren Variablen dieses Typs:

```
int lower, upper, step;
char c, line[256];
```

Variablen können beliebig auf mehrere Vereinbarungen verteilt werden → Vereinbarung kann kommentiert werden

```
int lower;
int upper;
int step;
char c;
char line[256];
```

Wenn eine Vereinbarung kommentiert werden muss, dann ist der Name der Variablen schlecht gewählt!

Vereinbarungen (2)

Eine Variable kann bei ihrer Vereinbarung auch initialisiert werden.

Beispiel: `double epsilon = 1.0e-5;`

Mit dem Attribut `const` kann bei der Vereinbarung einer Variablen angegeben werden, dass sich ihr Wert nicht ändert.

Beispiel: `const double e = 2.71828182845904523536;`

Bei einem Vektor bedeutet `const`, dass die Elemente nicht verändert werden.

Beispiel: `const int arr[] = {1, 2, 3, 4};`
Dann ist zwar eine Anweisung wie `arr[0] = 0;` verboten, aber Elemente durch `int *z = arr;` über Zeiger `z` änderbar

Vereinbarungen (3)

Ändern einer `const`-Variablen: Resultat implementierungsabhängig!

- gcc → Warning: assignment of read-only variable
  aber: Laufzeitfehler!
- Borland C++→ Error: Cannot modify a const object.

Es ist möglich, einen Zeiger als `const` zu definieren → keine Adressarithmetik möglich, aber der Inhalt kann geändert werden:

```
int i = 10;
int* const zi = &i;

*zi += 1;   /* i = 11   */
zi += 1;    /* verboten */
```


Arithmetische Operatoren

Operator	Beispiel	Bedeutung
`+`	`+i`	positives Vorzeichen
`-`	`-i`	negatives Vorzeichen
`+`	`i+5`	Addition
`-`	`i-j`	Subtraktion
`*`	`i*8`	Multiplikation
`/`	`i/5`	Division
`%`	`i%6`	Modulo
`=`	`i = 5+j`	Zuweisung
`+=`	`i += 5`	`i = i+5`
`-=`	`i -= 6`	`i = i-6`
`*=`	`i *= 5`	`i = i*5`
`/=`	`i /= 7`	`i = i/7`

Arithmetische Operatoren (2)

Anmerkungen:

- der Operator `%` kann **nicht** auf `float`- oder `double`-Werte angewendet werden.
- negative Operanden → maschinenabhängiges Verhalten
  * In welcher Richtung wird bei `/` abgeschnitten?
    Beispiel: `-15 / 2 = -7` oder `-15 / 2 = -8` ???
  * Welches Vorzeichen hat das Resultat von `%`?
    Beispiel: `-15 % 12 = -3` oder `-15 % 12 = 9` ???

  → Probleme bei Portierungen!

Arithmetische Operatoren (3)

Anmerkungen: (Fortsetzung)

- Vorrang der Operatoren in abnehmender Reihenfolge:
  1. unäre Operatoren + und - (Vorzeichen)
  2. *, / und %
  3. binäre Operatoren + und -

  Beispiel: `5 * -7 - 3` → `(5 * (-7)) - 3`
- Arithmetische Operationen werden von links her zusammengefasst.

  Beispiel: `1 + 3 + 5 + 7` → `((1 + 3) + 5) + 7`

  zur Erinnerung: die Addition auf Gleitkommazahlen ist nicht assoziativ aufgrund von Rundungsfehlern bei der Denormalisierung

Inkrement- und Dekrementoperatoren

Ausdruck	Bedeutung
`++i`	`i` wird um den Wert 1 erhöht, **bevor** `i` im Ausdruck weiterverwendet wird (Präfix-Notation)
`--i`	`i` wird um den Wert 1 vermindert, **bevor** `i` im Ausdruck weiterverwendet wird (Präfix-Notation)
`i++`	`i` wird um den Wert 1 erhöht, **nachdem** `i` im Ausdruck weiterverwendet wird (Postfix-Notation)
`i--`	`i` wird um den Wert 1 vermindert, **nachdem** `i` im Ausdruck weiterverwendet wird (Postfix-Notation)

Inkrement- und Dekrementoperatoren (2)

Diese Operatoren können nur auf Variablen angewendet werden, nicht auf Ausdrücke. **Verboten:** `(i+j)++`

Ausdrücke werden unter Umständen schwer einsichtig:

```
int x, y;   /* Variablendeklaration */

x = 1;
y = ++x + 1;
            /* hier: x = 2, y = 3 */
x = 1;
y = x++ + 1;
            /* hier: x = 2, y = 2 */
x = 1;
x = ++x + 1;
            /* hier: x = 3 */
```


Seiteneffekte

Der Wert eines Ausdrucks ist oft von Variableninhalten abhängig. Die Auswertung des Ausdrucks ändert aber nicht den Inhalt der Variablen:

- `(x+1) * (x+1)` kann durch `sqr(x+1)` ersetzt werden
- `(x+1) - (x+1)` kann durch 0 ersetzt werden

Schreibt man `++x` anstelle von `x+1`, so ergeben sich unter Umständen seltsame Gleichungen: `(++x) - (++x) = -1`

`++x` hat hier den Seiteneffekt, dass der Inhalt von x um 1 erhöht wird.

Anmerkung: Die Auswertung eines Ausdrucks soll einen Wert liefern, aber keinen Seiteneffekt haben.

Vergleichsoperatoren

Op	Beispiel	Bedeutung
<	`i < 7`	kleiner als
<=	`i <= 7`	kleiner gleich
==	`i == 7`	gleich

Op	Beispiel	Bedeutung
>	`i > j`	größer als
>=	`i >= j`	größer gleich
!=	`i != j`	ungleich

Prioritäten:

- Vergleichsoperatoren <, >, <=, >= haben gleiche Priorität
- Äquivalenzoperatoren ==, != haben geringere Priorität
- Vergleiche: geringerer Vorrang als arithm. Operatoren

Beispiele:

- `i < l - 1` wird bewertet wie `i < (l - 1)`
- `2+2 < 3 != 5 > 7` entspricht `(4 < 3) != (5 > 7)`

Boolesche Werte in C

In C wird

- **false** durch den Wert 0 und
- **true** durch einen Wert ungleich 0 dargestellt.

$\Rightarrow$ **bizarre Ausdrücke möglich:** `3 < 2 < 1` ist **true**, denn `(3 < 2)` = **false** = 0 und `0 < 1`.

Oft findet man in C-Programmen verkürzte Anweisungen.

Beispiel:

- `while (x)` $\Longleftrightarrow$ `while (x != 0)`
- `while (strlen(s))` $\Longleftrightarrow$ `while (strlen(s) > 0)`

Logische Verknüpfungen

Op	Beisp.	Ergebnis (Bedeutung)
&&	`a && b`	a und b wahr, dann 1, sonst 0 (log. UND)
||	`a || b`	a oder b wahr, dann 1, sonst 0 (log. ODER)
!	`!a`	liefert 1, falls a falsch ist, sonst 0

Prioritäten:

- && hat Vorrang vor ||
- Vergleichs- und Äquivalenzoperatoren: höherer Vorrang

Beispiele:

- `i < l-1 && c != EOF` $\rightarrow$ keine Klammern notwendig
- `!valid` $\Longleftrightarrow$ `valid == 0`

Verkürzte Auswertung

Ausdrücke werden nur solange bewertet, bis das Ergebnis feststeht!

Es gilt:

- `X && Y == 0`, falls `X == 0`
- `X || Y == 1`, falls `X == 1`

Verkürzte Auswertung ist notwendig, um Laufzeitfehler zu vermeiden.

Beispiele:

- `(x != 1) && (1/(x-1) > 0)`
- `scanf("%d", &n) > 0 || exit(errno)`

Sonstige Operatoren in C

was es nicht gibt:

- keine Operationen, mit denen zusammengesetzte Objekte wie Zeichenketten, Mengen, Listen oder Vektoren direkt bearbeitet werden können
- keine Operationen, die einen ganzen Vektor oder eine Zeichenkette manipulieren, jedoch Struktur als Einheit kopierbar
- keine Ein- und Ausgabe, keine eingebauten Techniken für Dateizugriff

$\Rightarrow$ abstrakte Mechanismen müssen als explizit aufgerufene Funktionen zur Verfügung gestellt werden

C-Implementierungen enthalten eine relativ standardisierte Sammlung solcher Funktionen (ANSI-Standard).

Typumwandlungen

Operator mit Operanden unterschiedlichen Typs: Werte werden in gemeinsamen Datentyp umgewandelt!

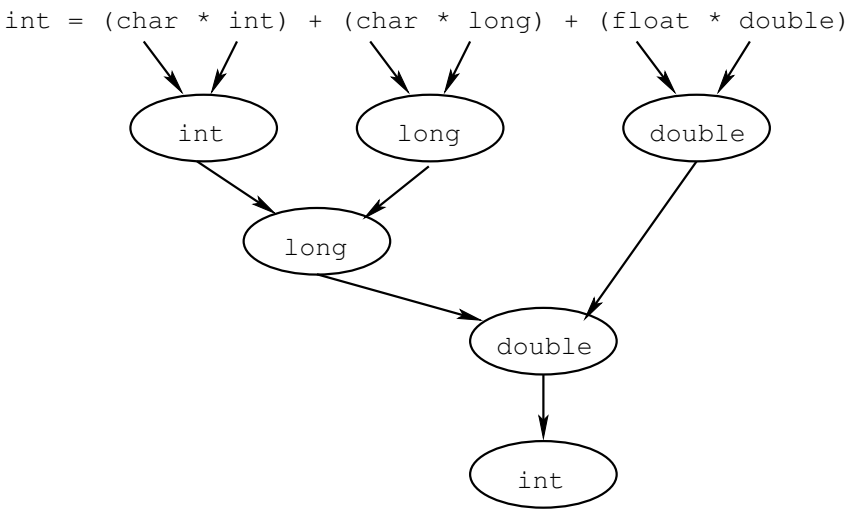
Implizite Typumwandlungen nur, wenn „kleiner“ Operand in „großen“ umgewandelt wird, ohne dabei Informationen zu verlieren (z.B. `int` nach `double`).

Ausdrücke, die zu Informationsverlust führen könnten, sind nicht verboten, können aber eine Warnung hervorrufen:

- Zuweisung eines langen Integer-Typs an einen kurzen
- Zuweisung eines Gleitpunkt-Typs an einen Integer-Typ

Sinnlose Ausdrücke, wie ein `float`-Wert als Vektorindex, sind verboten.

implizite Typumwandlungen



explizite Typumwandlungen

Typumwandlung mit unärer Umwandlungsoperation (**cast**) explizit erzwingen:

```
int a = 1, b = 2;
float x;

x = a / b;          /* x == 0   */
x = (float)a / b;   /* x == 0.5 */
```

Operatoren zur Bitmanipulation

Ganze Zahlen können als Bitvektoren aufgefasst werden:

```
    ...
-3  =  1111 1111 1111 1101
-2  =  1111 1111 1111 1110
-1  =  1111 1111 1111 1111
 0  =  0000 0000 0000 0000
 1  =  0000 0000 0000 0001
 2  =  0000 0000 0000 0010
 3  =  0000 0000 0000 0011
    ...
```

Manipulation einzelner Bits in C: Shift-Operatoren >> und << sowie logische Operatoren `&`, `|`, `^` und `~`

Operatoren zur Bitmanipulation (2)

nur auf Integer-Typen anwendbar:

Op	Beispiel	Bedeutung
<<	`i << j`	Links-Shift von `i` um `j` Stellen
>>	`i >> j`	Rechts-Shift von `i` um `j` Stellen
&	`i & j`	Bitweises UND von `i` und `j`
|	`i | j`	Bitweises ODER von `i` und `j`
^	`i ^ j`	Bitweises Exklusiv-ODER von `i` und `j`
~	`~i`	Einerkomplement von `i`

Prioritäten:

- Shift-Operatoren vor Äquivalenzoperatoren `==` und `!=`
- Äquivalenzoperatoren Vorrang vor `&`, `|` und `^`
- **Beispiel:** `(x & MASK) == 0` statt `x & MASK == 0`

Operatoren zur Bitmanipulation (3)

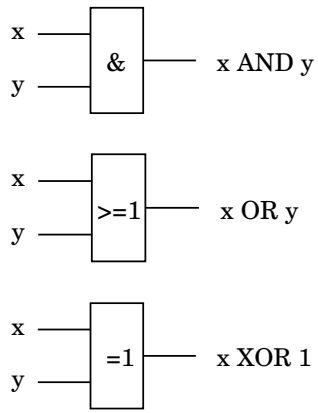
Beispiele:

```
short i = 1;          /* i = 0000 0000 0000 0001 */
i = i << 3;           /* i = 0000 0000 0000 1000 */
i = i >> 2;           /* i = 0000 0000 0000 0010 */
i = i | 5;            /* i = 0000 0000 0000 0111 */
i = i & 3;            /* i = 0000 0000 0000 0011 */
i = i ^ 5;            /* i = 0000 0000 0000 0110 */
i = ~i;               /* i = 1111 1111 1111 1001 */
```


Operatoren zur Bitmanipulation (4)

Beispiel: $x = 39 = (100111)_2$ und $y = 45 = (101101)_2$

```
  x   100111
  y = 101101
x&y = 100101

  x   100111
  y = 101101
x|y = 101111

  x   100111
  y = 101101
x^y = 001010
```



Shift-Operatoren

- schiebe einen Wert vom Typ `unsigned` nach rechts
  → es wird immer Null nachgeschoben

- schiebe vorzeichenbehafteten Wert nach rechts
  * **arithmetic shift:** Vorzeichenbit wird nachgeschoben
  * **logical shift:** Null-Bits werden nachgeschoben

Shift-Operatoren (2)

Mathematische Deutung für positive, ganze Zahlen:

- Links-Shift um m Stellen: Multiplikation mit 2^m
- Rechts-Shift, m Stellen: ganzzahlige Division durch 2^m

$z = (x_n x_{n-1} \ldots x_0)_2 \;\Rightarrow\; z = x_n \cdot 2^n + x_{n-1} \cdot 2^{n-1} + \cdots + x_0 \cdot 2^0$

Links-Shift oder Multiplikation mit 2:

$$\begin{aligned} z \cdot 2 &= x_n \cdot 2^{n+1} + x_{n-1} \cdot 2^n + \cdots + x_0 \cdot 2^1 + 0 \cdot 2^0 \\ &= (x_n\ x_{n-1}\ x_{n-2}\ \ldots\ x_0\ 0)_2 \end{aligned}$$

Rechts-Shift oder ganzzahlige Division durch 2:

$$\begin{aligned} z/2 &= x_n \cdot 2^{n-1} + x_{n-1} \cdot 2^{n-2} + \cdots + x_1 \cdot 2^0 \\ &= (x_n\ x_{n-1}\ x_{n-2}\ \ldots\ x_1)_2 \end{aligned}$$

Shift-Operatoren (3)

Beispiel: $17 = (00010001)_2$

$17 \cdot 2 = 34 = (00100010)_2$ $\quad \lfloor 17/2 \rfloor = 8 = (00001000)_2$
$17 \cdot 4 = 68 = (01000100)_2$ $\quad \lfloor 17/4 \rfloor = 4 = (00000100)_2$

Einerkomplement:

$$\begin{aligned} -17 &= (11101110)_2 \\ -17 \cdot 2 = -34 &= (11011101)_2 \rightarrow \textbf{kein Links-Shift!} \\ \lfloor -17/2 \rfloor = -8 &= (11110111)_2 \end{aligned}$$

Zweierkomplement:

$$\begin{aligned} -17 &= (11101111)_2 \\ -17 \cdot 2 = -34 &= (11011110)_2 \\ \lfloor -17/2 \rfloor = -8 &= (11111000)_2 \rightarrow \textbf{kein Rechts-Shift!} \end{aligned}$$

Shift-Operatoren (4)

Beispiel: $29 = (00011101)_2$

$29 \cdot 2 = 58 = (00111010)_2$ $\quad \lfloor 29/2 \rfloor = 14 = (00001110)_2$
$29 \cdot 4 = 116 = (01110100)_2$ $\quad \lfloor 29/4 \rfloor = 7 = (00000111)_2$

Einerkomplement:

$$\begin{aligned} -29 &= (11100010)_2 \\ -29 \cdot 2 = -58 &= (11000101)_2 \rightarrow \textbf{kein Links-Shift!} \\ \lfloor -29/2 \rfloor = -14 &= (11110001)_2 \end{aligned}$$

Zweierkomplement:

$$\begin{aligned} -29 &= (11100011)_2 \\ -29 \cdot 2 = -58 &= (11000110)_2 \\ \lfloor -29/2 \rfloor = -14 &= (11110010)_2 \rightarrow \textbf{kein Rechts-Shift!} \end{aligned}$$

Kontrollstrukturen

Kontrollstrukturen → zur Steuerung des Programmablaufs

In C: Kontrollstrukturen für **wohlstrukturierte** Programme

- Zusammenfassen von Anweisungen
- Entscheidungen (`if/else`)
- Auswahl aus einer Menge möglicher Fälle (`switch`)
- Schleifen mit Test des Abbruchkriteriums
  * am Anfang (`while`, `for`)
  * am Ende (`do`)
- vorzeitiges Verlassen einer Schleife (`break`, `continue`) ist nicht wohlstrukturiert!

Zusammenfassen von Anweisungen

Ausdrücke:

```
x = 0
i--
printf(...)
```

→

Anweisungen:

```
x = 0;
i--;
printf(...);
```

Anweisungsfolgen mit geschweiften Klammern zusammenfassen → der geklammerte Block gilt als eine Anweisung

```
{
    x = 0;
    i--;
    printf(...);
}
```

Struktogramm:

Anweisung 1
Anweisung 2
Anweisung 3

Konditional-Ausdrücke

→ werden mittels ternärem Operator `?:` gebildet

Syntax:

```
expr_0 ? expr_1 : expr_2
```

Erklärung: (Semantik)

- Ausdruck `expr_0` auswerten
- falls `expr_0` gilt, dann `expr_1` auswerten, sonst `expr_2`
- es wird entweder `expr_1` oder `expr_2` ausgewertet

Beispiele:

- `max = (i > j) ? i : j;`
- `x = (x < 20) ? 20 : x;`

abweisende/kopfgesteuerte Schleife

Syntax:

```
while (ausdruck)
    anweisung
```

Erklärung: (Semantik)

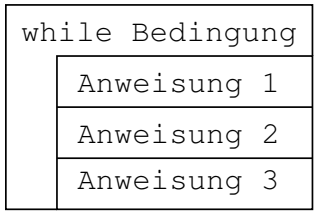
- `anweisung` wird ausgeführt, solange `ausdruck` wahr ist.
- `ausdruck` wird vor jedem Schleifendurchlauf bewertet.
- Kontrollausdruck vorm ersten Durchlaufen der Schleife nicht erfüllt → `anweisung` wird gar nicht durchlaufen

abweisende/kopfgesteuerte Schleife (2)

Beispiel:

```
int i = 0, sum = 0;
while (i < 10) {
    sum += i;
    i++;
}
```

Struktogramm:



fußgesteuerte Schleife

Syntax:

```
do
    anweisung
while (ausdruck);
```

Erklärung: (Semantik)

- Die Anweisung in der Schleife wird ausgeführt, bis der Kontrollausdruck `ausdruck` nicht mehr erfüllt ist.
- **Die Schleifenanweisung `anweisung` wird mindestens einmal ausgeführt!**

fußgesteuerte Schleife (2)

Beispiel:

```
int i = 0, sum = 0;
do {
    sum += i;
    i++;
} while (i < 10);
```

Struktogramm:

do/while vs. repeat/until

In C gibt es keine `repeat/until`-Schleifen. Diese können aber durch `do/while`-Schleifen nachgebildet werden:

```
do                          entspricht      repeat
    .....                                      .....
while B                                     until !B
```

Beispiel:

```
do {                                       repeat {
    sum += i;                                  sum += i;
    i++;                    entspricht         i++;
} while (i < 10);                          } until (i >= 10);
```

Zähl-Schleife

`for`-Schleife: bietet Möglichkeit, einfache Initialisierungen und Zählvorgänge übersichtlich zu formulieren

Syntax: `for (ausdruck_1; ausdruck_2; ausdruck_3)`
`anweisung`

Erklärung: (Semantik)

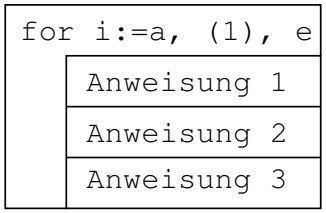
- `ausdruck_1` → Initialisierungsausdruck
- `ausdruck_2` → Abbruchkriterium der Schleife: die Schleife wird solange durchlaufen, wie `ausdruck_2` erfüllt ist
- `ausdruck_3` wird nach jedem Schleifendurchlauf bewertet → Schleifenvariablen ändern

Zähl-Schleife (2)

Beispiel:

```
int i, sum = 0;
for (i = 0; i < 10; i++) {
    sum += i;
}
```

Struktogramm:



Anmerkung: Bei keinem Schleifentyp sind die geschweiften Klammern { und } Bestandteil der Syntax! Gleichbedeutend zu oben:

```
int i, sum = 0;
for (i = 0; i < 10; i++)
    sum += i;
```


Zähl-Schleife (3)

Sollen mehrere Anweisungen bei jedem Schleifendurchlauf ausgeführt werden, dann müssen diese durch { und } zu einer logischen Anweisung (Block) zusammengefasst werden.

Beispiel: Das folgende Programm gibt die Werte n, n^2, n^3, 2^n und $n!$ in Form einer Tabelle aus.

Zähl-Schleife (4)

```
#include <stdio.h>

main() {
    int i, iHoch2, iHoch3;
    int pot2 = 1, fak = 1;

    for (i = 1; i <= 12; i++) {
        iHoch2 = i * i;
        iHoch3 = iHoch2 * i;
        pot2 *= 2;
        fak *= i;
        printf("%2d\t%4d\t%6d\t%8d\t%10d\n",
               i, iHoch2, iHoch3, pot2, fak);
    }
}
```

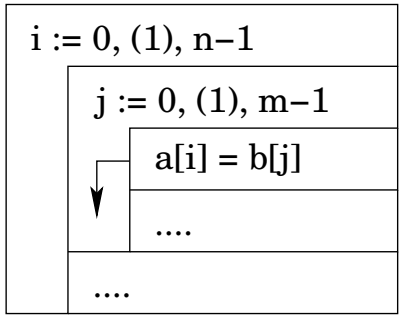
break

Mittels der `break`-Anweisung kann die innerste Schleife vorzeitig und unmittelbar verlassen werden.

Beispiel:

```
for (i = 0; i < n; i++) {
    for (j = 0; j < m; j++) {
        if (a[i] == b[j])
            break;
        .....
    }
    .....
}
```

Struktogramm:



keine strukturierte Programmierung

continue

`continue`-Anweisung: die nächste Wiederholung der umgebenden Schleife wird unmittelbar begonnen.

Beispiel:

```
for (i = 0; i < n; i++) {
    if (a[i] < 0)
        continue;
    .....
}
```

- keine strukturierte Programmierung
- in Struktogrammen nicht darstellbar

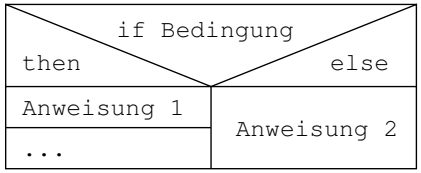
Auswahlanweisungen

Mittels Auswahlanweisungen kann der Ablauf eines Programms abhängig von Bedingungen geändert werden.

Syntax:

```
if (ausdruck)
    anweisung_1
else
    anweisung_2
```

Struktogramm:



Erklärung: (Semantik)

- `ausdruck` wird bewertet. Falls er wahr ist, wird die Anweisung `anweisung_1` ausgeführt, sonst `anweisung_2`.
- Der `else`-Zweig kann entfallen → `anweisung_1` wird übersprungen, falls der Ausdruck `ausdruck` nicht erfüllt ist.

Auswahlanweisungen (2)

In C ist der Wert einer Zuweisung der Wert, der an die linke Seite zugewiesen wird.

- `if (i = 1)` ist immer erfüllt → **Vorsicht!**
- `if (i == 1)` die wahrscheinlich gewünschte Anweisung
- `if (1 == i)` verhindert solche Fehler → **besser!**

Zuweisung kann als Teil eines Ausdrucks verwendet werden!

Beispiele:

- `while ((c = getchar()) != EOF)`
- `if ((l = strlen(s)) > 0)`

⇒ **Probleme beim Debuggen!**

Multiplikations-Algorithmus

```
#include <stdio.h>
main() {
    int a, b, i, n, mult;           /* a, b aus Nat */
    printf("Eingabe: a, b");
    scanf("%d, %d", &a, &b);

    n = 0;              /* Wie viele Stellen hat b? */
    while (b > (1 << n))
        n++;

    mult = 0;                  /* Multiplikation a*b */
    for (i = 0; i < n; i++)
        if (b & (1 << i))
            mult += a << i;
}
```


Multiplikations-Algorithmus (2)

Ablauf: Sei $a = 9 = (1001)_2$ und $b = 13 = (1101)_2$.

Wie viele Stellen hat b?

n	1 << n	b > (1 << n)
0	00001	ja
1	00010	ja
2	00100	ja
3	01000	ja
4	10000	nein

Multiplikation:

i	1 << i	b & (1 << i)	a << i
0	0001	0001	1001
1	0010	0000	----
2	0100	0100	100100
3	1000	1000	1001000

Multiplikations-Algorithmus (3)

C-Freaks schreiben wahrscheinlich:

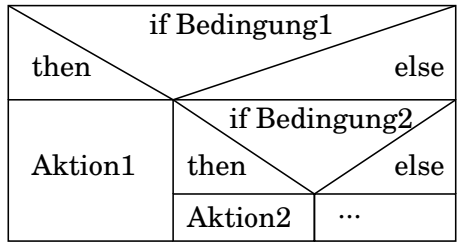
```
#include <stdio.h>
main() {
    int a, b, i, n, mult;
    printf("Eingabe: a, b");
    scanf("%d, %d", &a, &b);

    for (n = 0; b > (1 << n); n++)
        ;

    mult = 0;
    for (i = 0; i <= n; i++)
        (b & (1 << i)) && (mult += a << i);
}
```


Auswahlanweisungen (3)

Syntax:

```
if (ausdruck_1)
    anweisung_1
...
else if (ausdruck_n)
    anweisung_n
else anweisung
```

Struktogramm:



Erklärung: (Semantik)

- `ausdruck_1`, `ausdruck_2`, ... der Reihe nach bewerten
- `ausdruck_i` erfüllt → `anweisung_i` ausführen und die Abarbeitung der Kette abbrechen
- ist kein Ausdruck wahr → die Anweisung im `else`-Teil ausführen. Der `else`-Teil ist optional.

Auswahlanweisungen (4)

Beispiel:
Schaltjahr-Bestimmung mittels Auswahlanweisungen

```
if (jahr % 4 != 0)
    tage = 365;
else if (jahr % 100 != 0)  /* jahr % 4 == 0 */
    tage = 366;
else if (jahr % 400 != 0)  /* jahr % 100 == 0 */
    tage = 365;
else tage = 366;
```


Auswahlanweisungen (5)

Beispiel:
Schaltjahr-Bestimmung mittels Konditional-Ausdruck

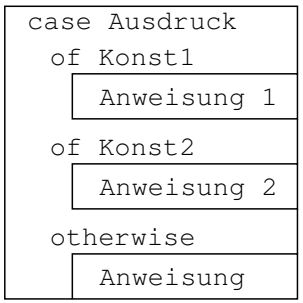
```
tage = (jahr % 4 != 0)
        ? 365
        : ((jahr % 100 != 0)
                ? 366
                : ((jahr % 400 != 0) ? 365 : 366));
```


Auswahl aus mehreren Alternativen

Syntax:

```
switch (ausdruck) {
   case konst_1: anweisung_1
   case konst_2: anweisung_2
   ...
   case konst_n: anweisung_n
   default: anweisung
}
```

Struktogramm:



Erklärung: (Semantik)

1. `ausdruck` wird ausgewertet → muss konstanten ganzzahligen Wert liefern (`switch`-Ausdruck)
2. `case`-Marken werden abgefragt (besteht aus `case`, ganzzahligem konstanten Ausdruck und Doppelpunkt.

`switch`-Anweisung (2)

3. `case`-Konstante stimmt mit `switch`-Ausdruck überein → Programmfluss wird hinter der `case`-Marke fortgesetzt, Anweisungen folgender `case`-Teile werden ausgeführt
4. soll Kette nach Abarbeitung des `case`-Teils beendet werden → explizit durch `break`-Anweisung erzwingen
5. `case`-Konstanten einer Kette müssen unterschiedliche Werte haben.
6. keine übereinstimmende `case`-Marke gefunden → Anweisung hinter der `default`-Marke wird ausgeführt (`default`-Marke und `default`-Anweisung sind optional!)
7. `case`-Konstante darf in einigen Implementierungen keine `const`-Variable sein. (erlaubt: mittels `#define` festgelegte Konstanten.)

`switch`-Anweisung: Beispiel

```
switch (c) {
   case 'm': klein = 1;
   case 'M': n = 1000;
             break;
   case 'd': klein = 1;
   case 'D': n = 500;
             break;
   case 'c': klein = 1;
   case 'C': n = 100;
             break;
   case 'l': klein = 1;
   case 'L': n = 50;
             break;
   case 'x': klein = 1;
   case 'X': n = 10;
             break;
   case 'v': klein = 1;
   case 'V': n = 5;
             break;
   case 'i': klein = 1;
   case 'I': n = 1;
             break;
   default: n = 0;
}
```


Komma-Ausdrücke

Das Komma wird außer als Operator auch als Trennzeichen verwendet, z.B. in Listen von Argumenten.

Die Bedeutung eines Kommas hängt vom Kontext ab, ggf. Komma-Ausdrücke in Klammern setzen, um gewünschtes Ergebnis zu erhalten.

Syntax: `expr_1, expr_2`

Erklärung: zuerst `expr_1` auswerten, dann `expr_2`

Komma-Operator wird dazu verwendet, zwei Ausdrücke an einer Stelle unterzubringen, wo nur ein Ausdruck erlaubt ist, z.B. in Schleifen- oder Funktionsaufrufen.

Komma-Ausdrücke (2)

Beispiel:

```
....
for (i = 0, j = N; i < j; i++, j--) {
    ....
}
a = fkt(x, (y=3, y+2), z);     /* a = fkt(x, 5, z) */
....
```

Typ und Wert des Resultats eines Komma-Ausdrucks sind Typ und Wert des rechten Operanden.

Alle Nebenwirkungen der Bewertung des linken Operanden werden abgeschlossen, bevor die Bewertung des rechten Operanden beginnt.

Format von C-Programmen

C-Programme können formatfrei geschrieben werden, d.h. sie müssen keine bestimmte Zeilenstruktur (im Gegensatz zu Cobol) haben.

Schlüsselwörter und Namen sind getrennt zu schreiben!

Trennzeichen sind:

- Leerzeichen
- horizontaler Tabulator
- vertikaler Tabulator
- Seitenvorschub
- Zeilenende
- Kommentare

In vielen Firmen gibt es Konventionen

- wie Quelltexte zu formatieren sind
- wie Variablen, Tabellen, Dateien usw. zu benennen sind

Format von C-Programmen (2)

guter Einstieg für übersichtliche Formatierung:

`http://java.sun.com/docs/codeconv/index.html`

Ziel unserer Konventionen: größtmögliche

- Deutlichkeit
- Übersichtlichkeit
- Testfreundlichkeit

Im folgenden einige Regeln (Auszüge aus den Java Code Conventions) die für alle C-Programme, die Sie im Rahmen der Veranstaltung *Informatik I* schreiben, verbindlich sind!

Format von C-Programmen (3)

Line length: Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

Wrapping lines: When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Format von C-Programmen (4)

Example: break after a comma

```
someMethod(longExpression1, longExpression2,
        longExpression3, longExpression4,
        longExpression5);
```

Example: prefer higher-level breaks to lower-level breaks

```
var = someMethod1(longExpression1,
                  someMethod2(longExpression2,
                          longExpression3));
```


Format von C-Programmen (5)

Example: breaking an arithmetic expression

```
/* AVOID! */
longName1 = longName2 * (longName3 + longName4
            - longName5) + 4 * longname6;

/* PREFER! */
longName1 = longName2
            * (longName3 + longName4 - longName5)
            + 4 * longname6;
```

Format von C-Programmen (6)

Example: indenting method declarations

```
/* CONVENTIONAL INDENTATION */
someMethod(int anArg, int anotherArg,
           char yetAnotherArg, char andStillAnother) {
    ...
}

/* INDENT 8 SPACES TO AVOID VERY DEEP INDENTS */
long int horkingVeryLongMethodName(int anArg,
        int anotherArg, char yetAnotherArg,
        char andStillAnother) {
    ...
}
```


Format von C-Programmen (7)

Simple Statements: Each line should contain at most one statement.

Blank Lines: Blank lines improve readability by setting off sections of code that are logically related.

- Two blank lines should always be used between sections of a source file.
- One blank line should always be used
  * between methods
  * between the local variables in a method and its first statement
  * before a block or single-line comment
  * between logical sections inside a method

Format von C-Programmen (8)

Blank spaces: should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space.
- A blank space should appear after commas in argument lists.
- All binary operators except . should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment („++“), and decrement („--“) from their operands.
- The expressions in a for statement should be separated by blank spaces.
- Casts should be followed by a blank space.

Format von C-Programmen (9)

Naming conventions: make programs more understandable by making them easier to read.

- Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.

  Example: `run()` oder `getBackgroundColor()`
- The names of variables declared constants should be all uppercase with words separated by underscores.

  Example:

```
#define PI_OVER_2 1.5707963
const int MIN_WIDTH = 4;
```

Format von C-Programmen (10)

Compound statements are statements that contain lists of statements enclosed in braces { statements }.

- The enclosed statements should be indented one more level than the compound statement.
- The opening brace should be at the end of the line that begins the compound statement; the closing brace should begin a line and be indented to the beginning of the compound statement.

Format von C-Programmen (11)

if statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult.

```
/* DON'T USE THIS INDENTATION */
if ((condition1 && condition2)
    || (condition3 && condition4)) {
    doSomethingAboutIt();
}

/* USE THIS INDENTATION INSTEAD */
if ((condition1 && condition2)
        || (condition3 && condition4)) {
    doSomethingAboutIt();
}
```


Kommentare

Jeder Dummkopf kann Code schreiben, den ein Computer versteht. Aber: Gute Programmierer schreiben Code, den Menschen verstehen!

Programme sind grundsätzlich zu kommentieren:

- immer anzugeben: Author, Version, Beschreibung, Datenstrukturen, Rahmenbedingungen, Änderungsliste.
- Code-Blöcke kommentieren, keine einzelnen Zeilen!
- Wenn eine Variable kommentiert werden muss, ist der Name der Variablen schlecht gewählt! → Bezeichner müssen aussagekräftig sein
- Jede Funktion wird beschrieben, dazu gehören u.a. die Parameter und der Rückgabewert.

Kommentare (2)

So nicht:

```
i += 1;                 /* i um eins erhöhen        */
for (j=0; j<=n; j++)    /* zähle j von 0 bis n hoch */
```

sondern so:

```
/* tausche x und y */
if (x > y) {
    x = x - y;
    y = x + y;
    x = y - x;
}
```

oder so:

```
/* berechne s = x*x */
s = 0;
i = 1;
for (j=1; j <= x; j++) {
    s += i;
    i += 2;
}
```

Kommentare (3)

Zur Zeit: Wenn man einen Kommentar benötigt, um zu erklären, was ein Code-Block tut, sollte der Code umgeschrieben werden. → Refactoring: Methode extrahieren

```
...
if (d1.jahr > d2.jahr)
   || (d1.jahr == d2.jahr
      && d1.monat > d2.monat)
   || (d1.jahr == d2.jahr
      && d1.monat == d2.monat
      && d1.tag > d2.tag) {
   ...
}
...
```

```
...
bool greater(date d1,
        date d2) {
    ...
}
...
if (greater(d1, d2)) {
    ...
}
...
```


Ohne Worte

```
#include<stdio.h>
#include<string.h>
main(){char*O,l[999]="’‘acgo\177~|xp .-"
"\OR^8)NJ6%K4O+A2M(*OID57$3G1FBL";while
(O=fgets(l+45,954,stdin)){*l=O[strlen(O)[O-1]=0,
strspn(O,l+11)];while(*O)switch((*l&&isalnum(*O))-!*l){
case-1:{char*I=(O+=strspn(O,l+12)+1)-2,O=34;while
(*I&3&&(O=(O-16<<1)+*I---’-’)<80);putchar(O&93?*I&8||
!(I=memchr(l,O,44))?’?’:I-l+47:32);break;case 1:;}*l=
(*O&31)[l-15+(*O>61)*32];while(putchar(45+*l%2),(*l=
*l+32>>1)>35);case 0:putchar((++O,32));}putchar(10);}}
```

gefunden unter `http://www.arkko.com/ioccc.html`