OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

INFORMATIK UND
MATHEMATIK

Programmieren 1 – Informatik

Kontrollstrukturen & Funktionen

Prof. Dr.-Ing. Maike Stern | 25.10.2023

Organisatorisches

- Die nächste Vorlesung fällt aus: Allerheiligen
- Die Übung nach Allerheiligen (6.11.):
  - Ich lade ein weiteres Übungsblatt zu den bisherigen Themen hoch, inklusive der Lösung
  - Auf ELO wird es am 6.11. einen Chat geben für Fragen
  - Sie können natürlich trotzdem den CIP-Pool nutzen

Austausch auf ELO

Im Kurs-ELO gibt es ein offenes Forum, auf dem diejenigen mit Erfahrung ihre Hilfe denjenigen anbieten können, die sich noch nicht so gut auskennen.

Organisatorisches

Vorlesungsfolien auf ELO:

Ich werde die Vorlesungsfolien (meist recht kurzfristig) vor der Vorlesung auf ELO hochladen – inklusive aller enthaltenen Fragen / Übungen und Antworten – würde ich die Antworten entfernen, bliebe teilweise nicht mehr viel übrig.

Die Übungen / Fragen sind als Lernhilfe für Sie gedacht. Spoilern Sie sich also im Idealfall nicht selbst, sondern nutzen Sie die Folien um sich während der Vorlesung Notizen zu machen.

Rückblick

- Hello World
- Ein- und Ausgabe
- Variablen
- Datentypen
- Texteditor, Konsole, IDE

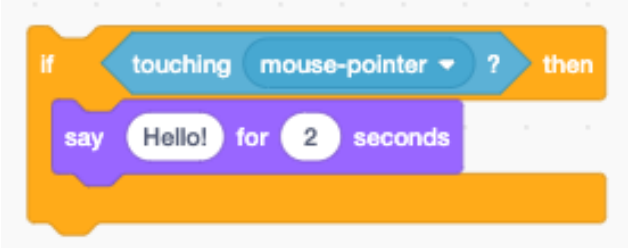
In dieser Vorlesung

- Kontrollstrukturen
  - If
  - Switch
  - While
  - For
- Funktionen
  - Funktionen mit Parametern und Rückgabewert
  - Geltungsbereich von Funktionen
  - Datentyp-Spezifizierer
  - Rekursive Funktionen

Kontrollstrukturen

Kontrollstrukturen

- In der ersten Vorlesung haben wir bereits verschiedene Kontrollstrukturen kennengelernt – in Verbindung mit der Programmiersprache Scratch
- Diesmal schauen wir uns die C-Syntax genauer an



ifExample.c — UNREGISTERED

```c
#include <stdio.h>

int main (){
    double number = 42.;

    if(number < 100){
        number += .1;
        printf("Number is now: %2.2f\n", number);
    }
}
```

Line 8, Column 39 — Tab Size: 4 — C

If-Anweisung

Kontrollanweisungen
If-Anweisung

Die If-Anweisung ermöglicht es Entscheidungen zu treffen, basierend auf einem Ereignis

Wenn dieses Ereignis wahr ist
dann führe diese Aktion aus

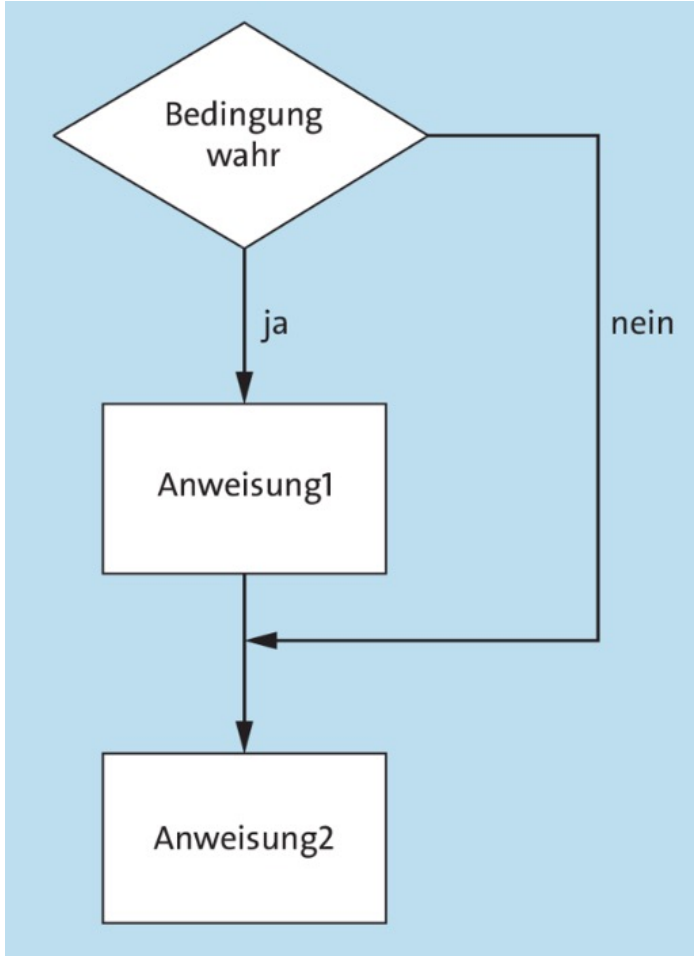


Bild: Wolf & Krooß, C von A bis Z, Rheinwerk Verlag

Kontrollanweisungen
If-Else-Anweisung

Die If-Anweisung ermöglicht es Entscheidungen zu treffen, basierend auf einem Ereignis

Wenn dieses Ereignis wahr ist
dann führe diese Aktion aus

Die If-Anweisung kann auch erweitert werden zu einer "Entweder-Oder-Entscheidung", nämlich if-else:

Entweder ist dieses Ereignis wahr
dann führe diese Aktion aus
Oder dieses andere Ereignis ist wahr
dann führe diese andere Aktion aus

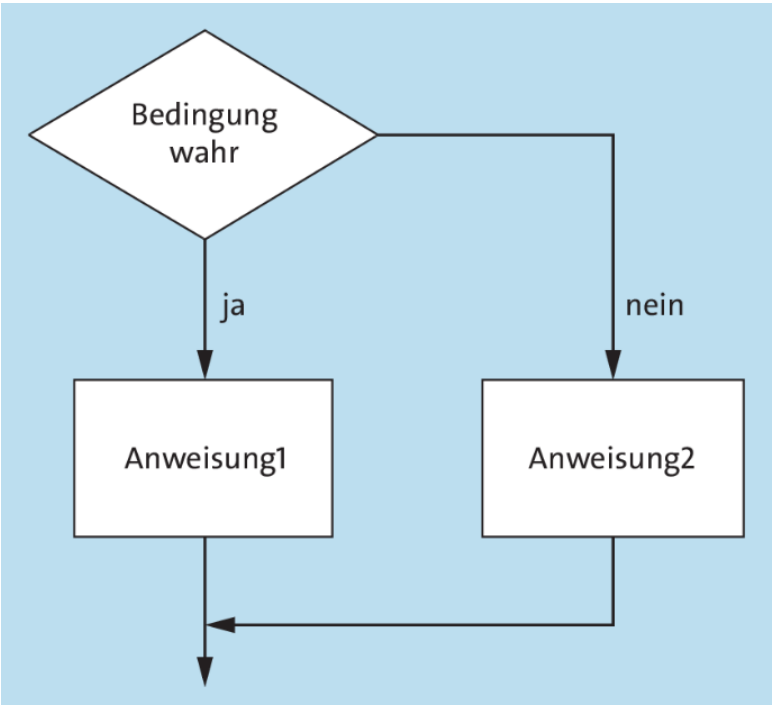


Bild: Wolf & Krooß, C von A bis Z, Rheinwerk Verlag

Kontrollanweisungen
If-Else-Anweisung

Syntax:

```
…
if(Bedingung){
    Anweisung 1;
}
else{
    Anweisung 2;
}
…
```

Achtung:
Einzeilige Kontrollsequenzen können auch ohne geschweifte Klammern geschrieben werden:

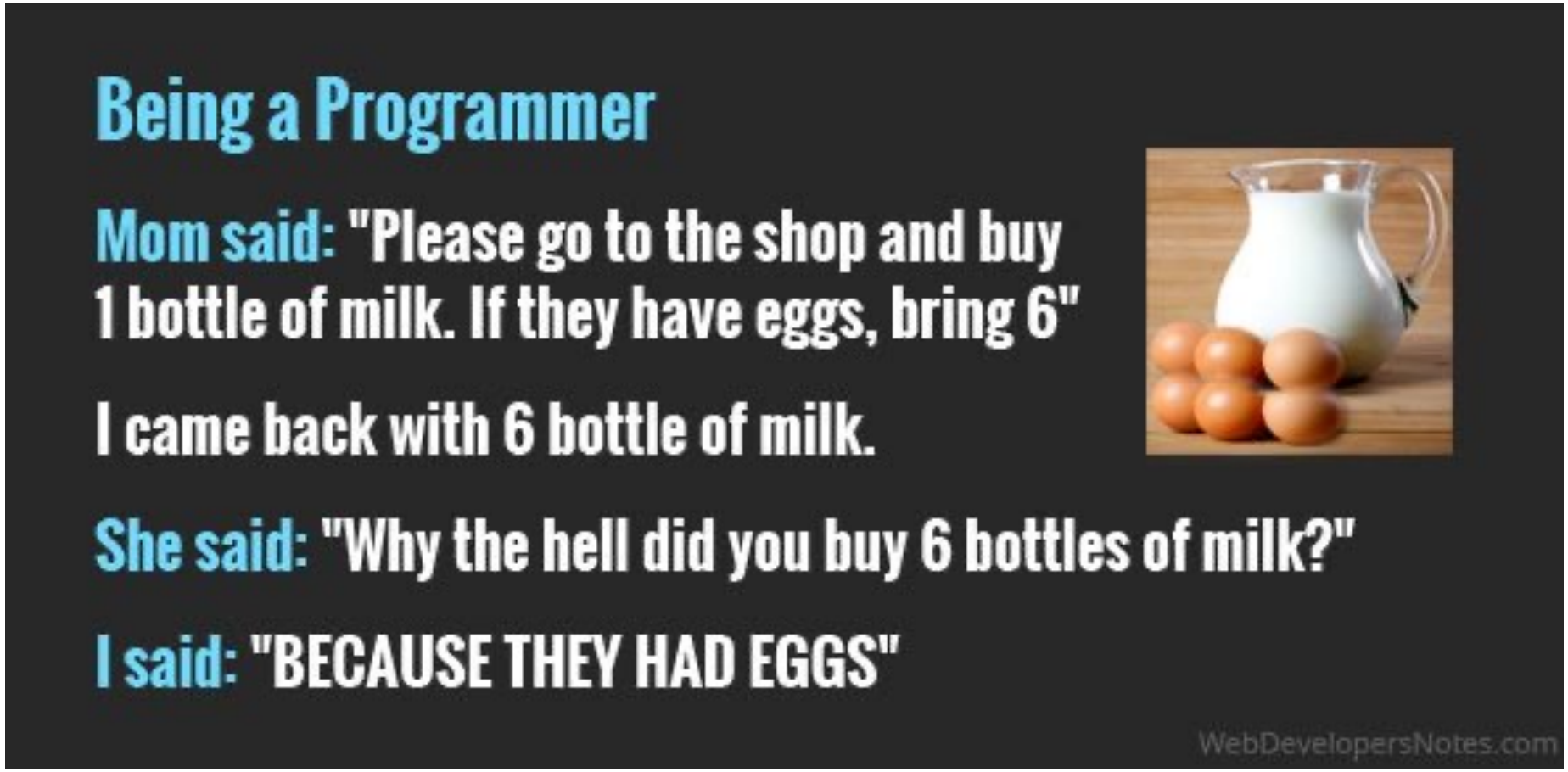
```
If(bar)
   foo = false;
```

ABER: ergänzt man später eine Zeile und vergisst die Klammer zu ergänzen, dann gehört die Anweisung in der zweiten Zeile nicht mehr zur Kontrollsequenz → fehleranfällig

Bild: Wolf & Krooß, C von A bis Z, Rheinwerk Verlag

Kontrollstrukturen
If-Else-Anweisung

```
…
if(Bedingung){
    Anweisung 1;
}
else{
    Anweisung 2;
}
…
```



eggsAndMilk.c

Kontrollstrukturen
If-Anweisung: Vergleiche

Die Bedingung in der If-Anweisung kann verschiedene Formen annehmen
→ Grundsätzlich geht es darum, dass die Bedingung im Ergebnis wahr oder falsch ist

- True / false
- Größer gleich / kleiner gleich / ungleich / gleich …
- Logische Operatoren
- Funktionsrückgabewert (kommt später)

```
…
if(Bedingung){
    Anweisung 1;
}
else{
    Anweisung 2;
}
…
```

Kontrollstrukturen
If-Anweisung: Operatoren für Vergleiche

Die Bedingung in der If-Anweisung kann verschiedene Formen annehmen
→ Grundsätzlich geht es darum, dass die Bedingung im Ergebnis wahr oder falsch ist

- True / false als boolesche Variable haben wir schon kennengelernt in den vorherigen Vorlesungen
  - Einbinden via #include <stdbool.h>
  - false entspricht 0
  - true entspricht ≥ 1, das heißt:
    int i = 4;
    if (i){…} // die Bedingung ist true

```
#include <stdbool.h>

bool decision = true;
…
if(decision){
    Anweisung 1;
}
else{
    Anweisung 2;
}
…
```

Kontrollstrukturen
If-Anweisung: Operatoren für Vergleiche

Die Bedingung in der If-Anweisung kann verschiedene Formen annehmen
→ Grundsätzlich geht es darum, dass die Bedingung im Ergebnis wahr oder falsch ist

- True / false als boolesche Variable haben wir schon kennengelernt in den vorherigen Vorlesungen
  - Einbinden via #include <stdbool.h>
  - false entspricht 0
  - true entspricht ≥ 1, das heißt:
    int i = 4;
    if (i){…} // die Bedingung ist true

```
#include <stdbool.h>

bool decision = true;
…
if(decision){
    Anweisung 1;
}
else{
    Anweisung 2;
}
…
```

Grundsätzlich gilt:
if(Ausdruck) ist dasselbe wie if(Ausdruck == true) bzw. if(Ausdruck != 0)
if(!Ausdruck) ist dasselbe wie if(Ausdruck == false) bzw. if(Ausdruck == 0)

Kontrollstrukturen
If-Anweisung: Operatoren für Vergleiche

Die Bedingung in der If-Anweisung kann verschiedene Formen annehmen
→ Grundsätzlich geht es darum, dass die Bedingung im Ergebnis wahr oder falsch ist

- Größer gleich / kleiner gleich / ungleich / gleich …

Operator	Bedeutung	Beispiel	Wahr, wenn:
>	größer (greater than)	a > b	a größer ist als b
<	kleiner (less than)	a < b	a ist kleiner als b
==	ist gleich	a == b	a ist gleich b
>=	größer gleich (greater than or equal to)	a >= b	a ist größer als b oder beide haben denselben Wert
<=	kleiner gleich (less than or equal to)	a <= b	a ist kleiner als b oder beide haben denselben Wert
!=	ungleich	a != b	a und b haben unterschiedliche Werte

```
int a = 3, int b = 5;
…
if(a == b){
    Anweisung 1;
}
else{
    Anweisung 2;
}
…
```

Kontrollstrukturen
If-Anweisung: Vergleiche

Die Bedingung in der If-Anweisung kann verschiedene Formen annehmen
→ Grundsätzlich geht es darum, dass die Bedingung im Ergebnis wahr oder falsch ist

- Größer gleich / kleiner gleich / ungleich / gleich …

if(a == b)

wird zu true oder false
ausgewertet: a==b → true/false

```
int a = 3, int b =5;
…
if(a == b){
    Anweisung 1;
}
else{
    Anweisung 2;
}
…
```

Nicht verwechseln:
a == b ist ein Vergleich,
a = b weist a den Wert von b zu

example_equalAndIs.c

Was ist an diesem Programm falsch?

```c
#include <stdio.h>

int main(){

    int x = 10, y = -20;

    if(x < y);
        printf("%d is less than %d", x, y);

}
```

example_semikolon.c

Kontrollstrukturen
If-Anweisung: Logische Operatoren

Die Bedingung in der If-Anweisung kann verschiedene Formen annehmen
→ Grundsätzlich geht es darum, dass die Bedingung im Ergebnis wahr oder falsch ist

- Logische Operatoren: Verknüpfen mehrerer Bedingungen

Operator	Name	Wahr, wenn	Beispiel
&&	and	beide Vergleiche wahr sind	(20 == 20 && 1 > -2)
||	or	Einer der Vergleiche ist wahr	(20 == 20 || 7 < 2)
!	not	Die Bedingung ist falsch	(20 == 20 && !(7 < 2))

Short circuiting:
Ist die linke Seite des UND-Vergleichs false, so wird die rechte Seite gar nicht mehr ausgewertet

example_logicalOperators.c

Kurzschluss und Faulheit

- short circuit behaviour: Nutzlose Berechnungen werden **nicht ausgeführt**, wie zum Beispiel bei logischen UND-Vergleichen, wenn die linke Seite bereits *false* ist
  → Insbesondere bei booleschen Operationen

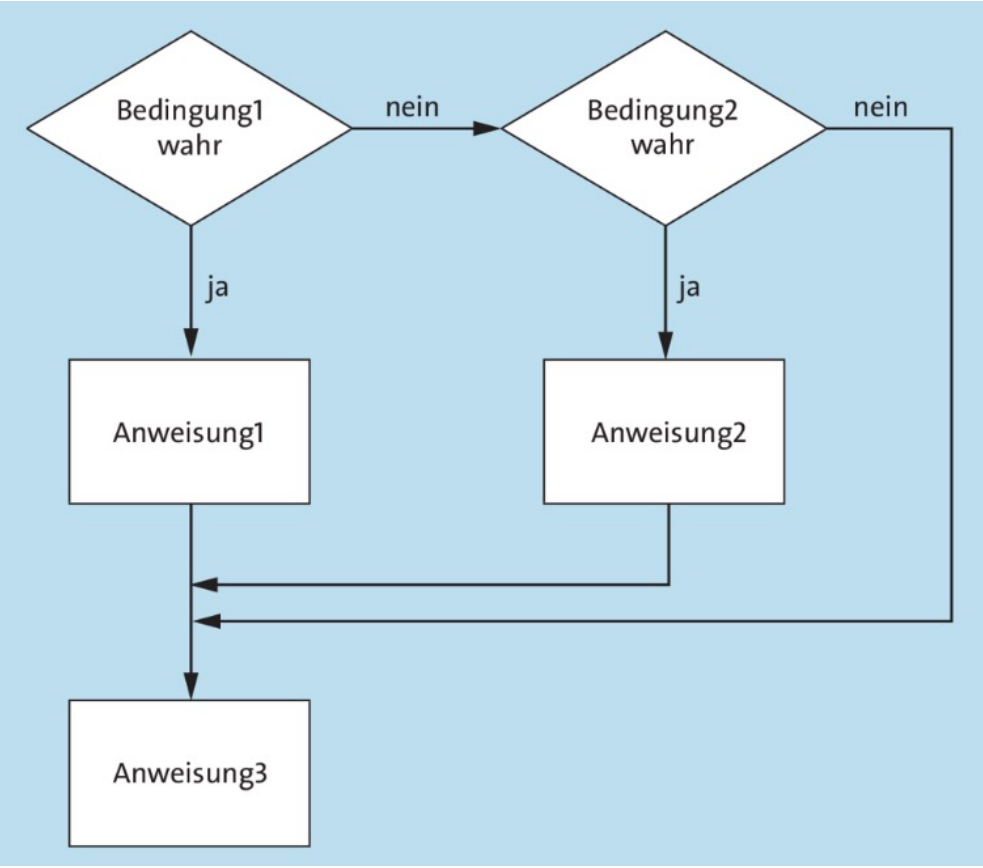
- lazy computation: **Verzögert** die Ausführung einer Berechnung bis zu dem Zeitpunkt, an dem sie tatsächlich notwendig ist. Wird die Berechnung gar nicht notwendig, wird sie auch nicht ausgeführt.
  → Gilt für alle Berechnungen

```
variable = bigAndSlowFunc() or evenSlowerFnc()
if (carry out heavy computations)
  print "Here it is: ", variable
else
  print "As you wish :-)"
```


https://stackoverflow.com/questions/14908548/any-difference-between-lazy-evaluation-and-short-circuit-evaluation

Kontrollstrukturen
If-Anweisung

Neben if-else-Anweisungsblöcken gibt es auch if-elseif-else-Anweisungen, die eine weitere Unterscheidung ermöglichen




```
int a = 3, int b =5;
…
if(a == b){
    Anweisung 1;
}
else if(a < b){
    Anweisung 2;
}
else{
    Anweisung 2;
}
…
```

If-Else und If-ElseIf-Else

```
int a = 3, int b =5;
…
if(a <= b){
    Anweisung 1;
}
if(a < b){
    Anweisung 2;
}
else{
    Anweisung 3;
}
…
```

```
int a = 3, int b =5;
…
if(a <= b){
    Anweisung 1;
}
else if(a < b){
    Anweisung 2;
}
else{
    Anweisung 3;
}
…
```

Frage: Wo liegt hier der Unterschied?

example_ifElseIf.c

Kontrollstrukturen
Fortgeschrittene Operatoren: Der Bedingungsoperator "?"

- In C gibt es auch eine Art Kurzform der If-Else-Anweisung, einen sogenanten ternären Operator: ?
- Syntax: Variable = Bedingung ? AnweisungWennWahr : AnweisungWennFalsch

```
if (a > b){
    maxValue = a;
}
else{
    maxValue = b;
}
```

Bedingung
?
Anweisung1
:
Anweisung2

Kontrollstrukturen
Fortgeschrittene Operatoren: Der Bedingungsoperator "?"

- In C gibt es auch eine Art Kurzform der If-Else-Anweisung, einen sogenanten ternären Operator: ?
- Syntax: Variable = Bedingung ? AnweisungWennWahr : AnweisungWennFalsch

```
if (a > b){
    maxValue = a;
}
else{
    maxValue = b;
}
```

```
int a = 0, b = 7;
maxValue = (a > b) ? a : b;  // maxValue = b = 7
```

Kontrollstrukturen
Fortgeschrittene Operatoren: Der Bedingungsoperator "?"

- Ternäre Ausdrücke sind hilfreich bei kleinen Problemen
- Es ist auch möglich ternäre Ausdrücke zu verschachteln, dann wird es aber schnell komplett unübersichtlich

```
int a = 1, b = 5, c = 3;
biggestNumber = (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);
```

Kontrollstrukturen
Fortgeschrittene Operatoren: Der Bedingungsoperator "?"

- Ternäre Ausdrücke sind hilfreich bei kleinen Problemen
- Es ist auch möglich ternäre Ausdrücke zu verschachteln, dann wird es aber schnell komplett unübersichtlich

```
int a = 1, b = 5, c = 3;
biggestNumber = (a > b) ? ((a > c) ? a : c) : ((b > c) ? b : c);
```

(a > c) ? a : c) → 3

((b > c) ? b : c) → 5

```
biggestNumber = (a > b) ? 3 : 5
```

(a > b) ? 3 : 5 → 5

example_ternaryExpression.c

Switch Case

Kontrollstrukturen
Switch Case

- Verschachtelte if-ElseIf-ElseIf-Else-Blöcke eignen sich zwar um Multiple-Choice-Szenarios zu programmieren, man verliert aber schnell den Überblick
- Für diesen Fall gibt es den sogenannten Switch Case, der eine Reihe von entweder-oder-oder-…-Abfragen umsetzt
- Syntax:

```
switch(Ausdruck){
        AUSDRUCK_1:     anweisung_1;
                        break;
        AUSDRUCK _2:    anweisung_2;
                        break;
        AUSDRUCK _3:    anweisung_3;
                        break;
        default:        anweisung;
}
```

```
int Ausdruck = 0;

printf("Input a number \
    between 1 and 3");
scanf("%d", &Ausdruck);

switch(Ausdruck){
    case 1: …
            break;
    case 2: …
            break;
    case 3: …
            break;
    default: …
}
```

example_switchCase.c

Kontrollstrukturen
Switch Case

- Verschachtelte if-ElseIf-ElseIf-Else-Blöcke eignen sich zwar um Multiple-Choice-Szenarios zu programmieren, man verliert aber schnell den Überblick
- Für diesen Fall gibt es den sogenannten Switch Case, der eine Reihe von entweder-oder-oder-...-Abfragen umsetzt
- Syntax:

die Ausdrücke können Integer-Zahlen aber auch Zeichen sein

```
switch(Ausdruck){
        AUSDRUCK_1:     anweisung_1;
                        break;
        AUSDRUCK _2:    anweisung_2;
                        break;
        AUSDRUCK _3:    anweisung_3;
                        break;
        default:        anweisung;
}
```

lässt man die break-Anweisung weg, so wird der nächste Fall auch ausgegeben. Solange, bis eine break-Anweisung die Switch-Verzweigung beendet.

wird ausgegeben, falls keiner der Ausdrücke gepasst hat

example_switchCalculator.c

While-Schleife
und
Do-While-Schleife

While-Schleife

- Auch die While-Schleife haben wir schon in der ersten Vorlesung kennengelernt. In Scratch ist sie als repeat-until-Schleife implementiert
- While-Schleifen ermöglichen es, Anweisungsblöcke wiederholt auszuführen
- Syntax:

```
while(Bedingung == wahr){
   // Anweisungen
}
```

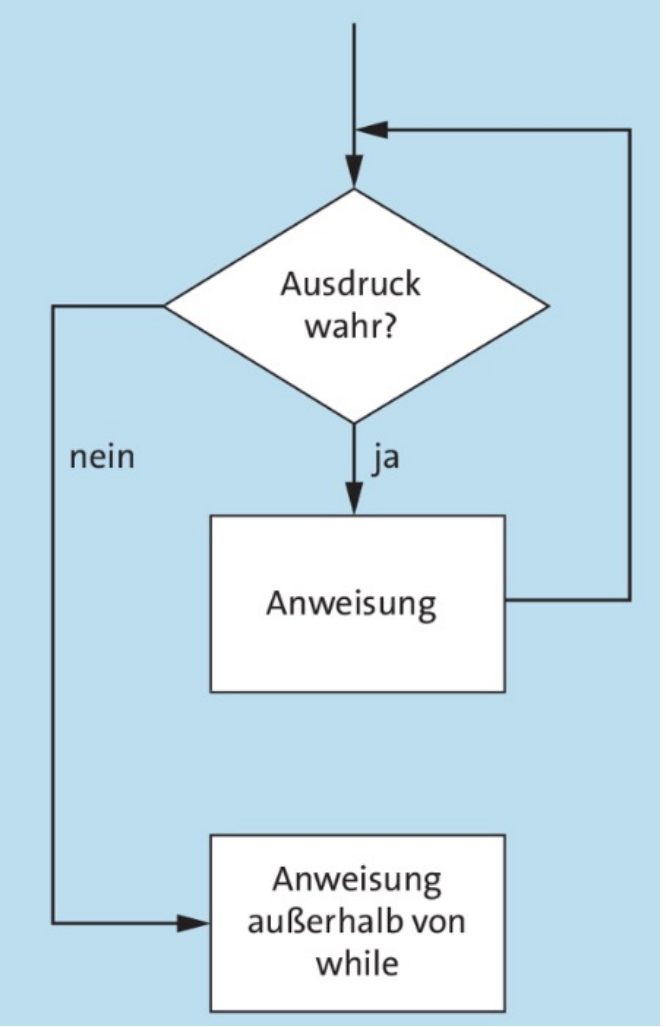


Bild: Wolf & Krooß, C von A bis Z, Rheinwerk Verlag

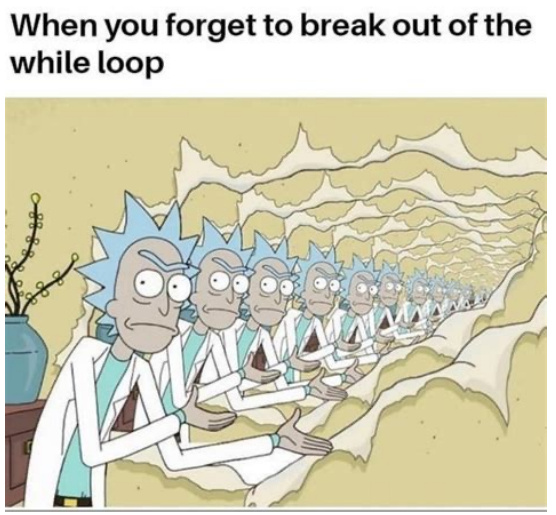
While-Schleife

- Auch die While-Schleife haben wir schon in der ersten Vorlesung kennengelernt. In Scratch ist sie als repeat-until-Schleife implementiert
- While-Schleifen ermöglichen es, Anweisungsblöcke wiederholt auszuführen
- Syntax:

```
while(Bedingung == wahr){
   // Anweisungen
}
```

```
int var = 0;

while(var < 10){
    // Anweisungen
    var++;
}
```

wichtig: Schleifenvariable reinitialisieren!



example_whileAddNumbers.c

While-Schleife

Bugs finden & fixen

example_whileBug_2.c
example_whileBug_3.c

While-Schleife Abbruchbedingungen

Neben dem Erfüllen der Abbruchbedingung gibt es noch weitere Möglichkeiten, Schleifen zu beenden

- continue: Beendet bei Schleifen (For-Schleife, While-Schleife) den aktuellen Schleifendurchlauf und startet den nächsten, falls vorhanden
- break: Beendet die Schleife oder eine Fallunterscheidung (if-Anweisung) vollständig. Bei verschachtelten Schleifen wird nur die innere Schleife beendet.

```
while(1){
    // Anweisungen
    if(Bedingung){
        break;
    }
}
```

example_whileBug.c

Do-While-Schleife

Die Do-While-Schleife ergänzt die While-Schleife:

- Der Anweisungsblock wird einmal ausgeführt
- Danach wird die Bedingung überprüft (jetzt also am Ende des Anweisungsblocks)
- Syntax:

```
do{
    // Anweisungen
} while(Bedingung == wahr);
```

Achtung, in diesem Fall wird die While-Bedingung mit einem Semikolon abgeschlossen

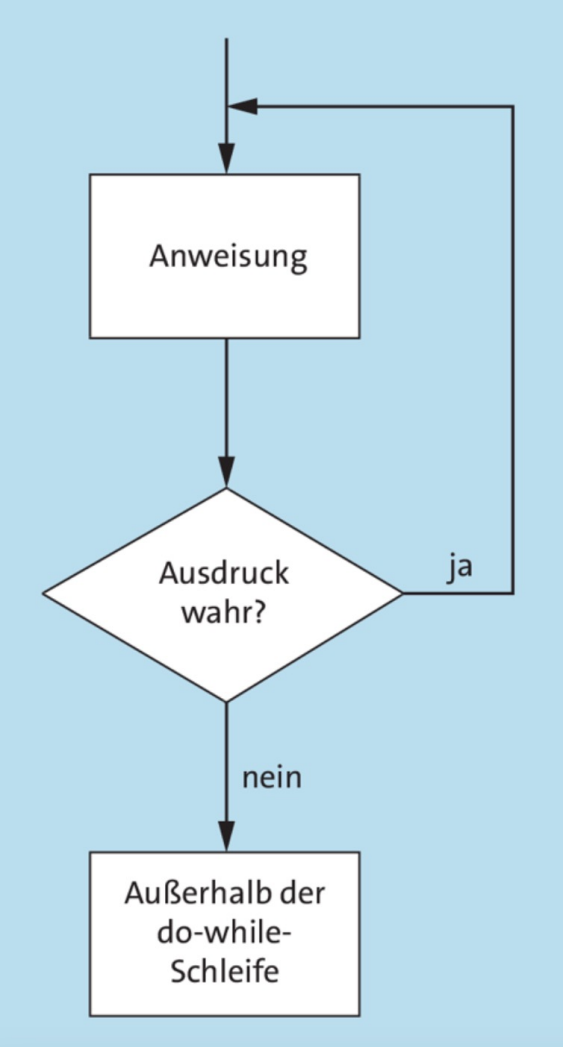


Bild: Wolf & Krooß, C von A bis Z, Rheinwerk Verlag

example_doWhile.c

Zwischenfazit
While vs. Do While

For-Schleife

Kontrollstrukturen
For-Schleife

Die For-Schleife haben wir auch schon in der ersten Vorlesung kennengelernt. Sie ermöglicht es, eine Aktion für eine geplante Anzahl zu wiederholen

Syntax:

```
for(Initialisierung; Bedingung; Reinitialisierung){
        // Anweisung
}
```

oder auch:

```
for([Variable] = [Start]; [Variable] <= [Ende]; [Variable] += [Schrittweite]){
        // Anweisung
}
```

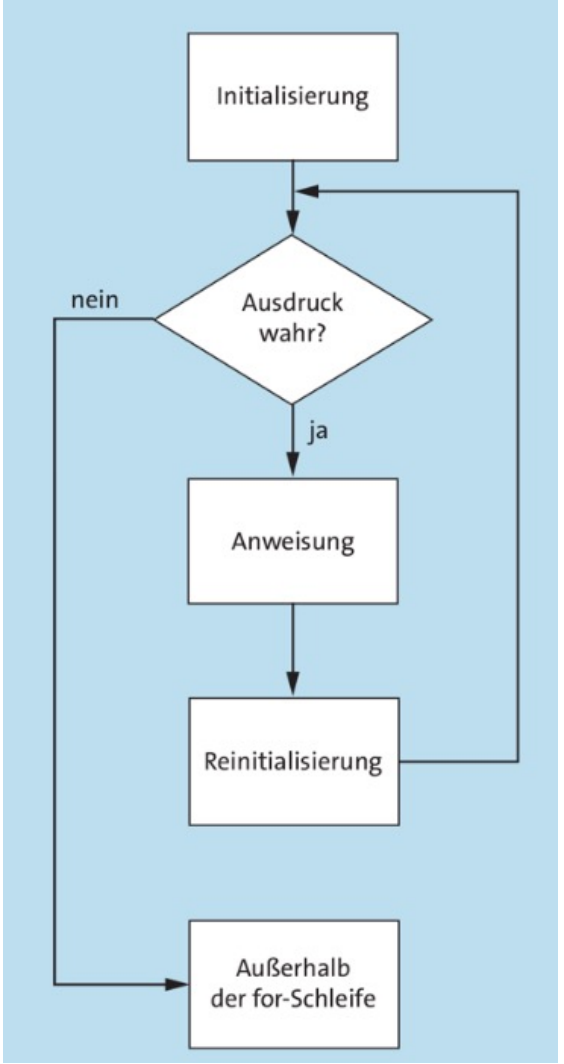


Bild: Wolf & Krooß, C von A bis Z, Rheinwerk Verlag

example_forPrint.c, example_forLargerThan1.c

Kontrollstrukturen
For-Schleife

Bugs finden & fixen

```c
#include <stdio.h>

int main(){

    for (char c = 'a', c <= 'z', c++){
        printf("%c\n", c);
    }
    return 0;
}
```

example_forBug.c

Kontrollstrukturen
For-Schleife

Bugs finden & fixen

```c
#include <stdio.h>

int main(){

    for (int i = 0; i = 10; i+= 1){
        printf("foo");
    }
    return 0;
}
```

example_forBug_2.c

Kontrollstrukturen
For-Schleife | Kuriose Verwendungen

```c
int main(){

    for (;;){
        printf("foo");
    }
    return 0;
}
```

example_forEndless.c

Kontrollstrukturen
For-Schleife | Kuriose Verwendungen

```c
int main(){

    for (;;){
        printf("Endlosschleife");
    }
    return 0;
}
```

for(;;) ist gleichwertig zu while(1),
also eine Endlosschleife

Kontrollstrukturen
For-Schleife | Kuriose Verwendungen

```c
int main(){

    for (;;){
        printf("Endlosschleife");
    }
    return 0;
}
```

for(;;) ist gleichwertig zu while(1), also eine Endlosschleife

```c
int main(){
    int n;

    for (printf("Bitte eine Zahl eingeben: \n"); n != 22;){
        scanf("%d", &n);
    }
    printf("Korrekte Zahl!\n");
    return 0;
}
```

example_forStrange.c

Kontrollstrukturen
For-Schleife | Kuriose Verwendungen

```c
int main(){

    for (;;){
        printf("Endlosschleife");
    }
    return 0;
}
```

for(;;) ist gleichwertig zu while(1), also eine Endlosschleife

```c
int main(){
    int n;

    for (printf("Bitte eine Zahl eingeben: \n"); n != 22;){
        scanf("%d", &n);
    }
    printf("Korrekte Zahl!\n");
    return 0;
}
```

Hier wird die Schleifenvariable nicht im Schleifenkopf reinitialisiert, sondern durch die Abfrage über die Konsole. Die Schleife läuft, bis die gesuchte Zahl (22) eingegeben wird.

example_forStrange.c

Kontrollstrukturen Zwischenfazit

Kontrollstrukturen ermöglichen es, aus dem sequentiellen Programmablauf auszubrechen

- **If-Anweisung:** Implementierung von Entweder-Oder-Entscheidungen, beruhend auf einem Ereignis
- **Switch Case:** Implementierung verschiedener Fälle
- **While-/Do-While-/For-Schleife:** Wiederholen des Anweisungsblocks

Funktionen

Funktionen

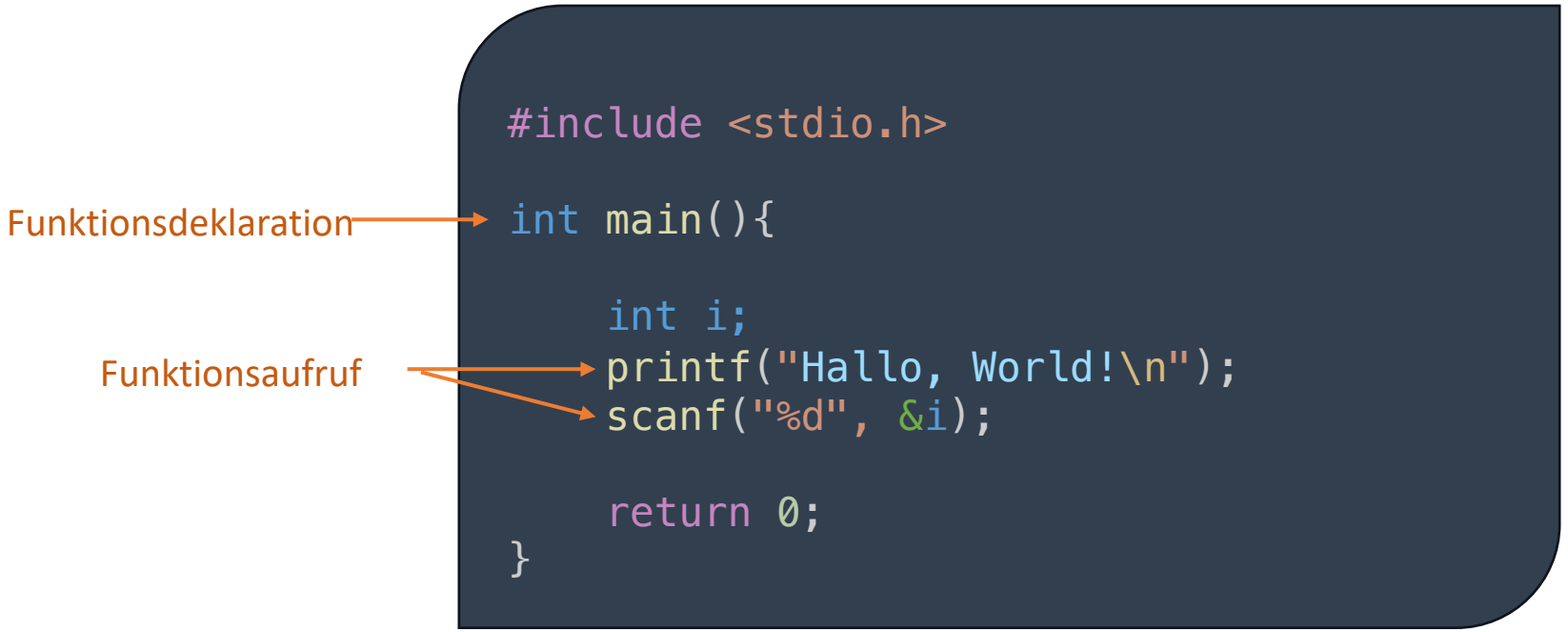
Funktionen sind Unterprogramme oder Subroutinen, die es ermöglichen den Code in verschiedene Teile aufzuteilen

- Verbesserte Lesbarkeit
- Sich wiederholende Routinen können in eine Funktion ausgelagert werden, was Schreibarbeit spart und die Fehleranfälligkeit reduziert
- Entsprechend lassen sich Fehler schneller finden und Veränderungen schneller umsetzen, da der Code nur noch an einer Stelle angepasst werden muss
- Die Funktionen können in Funktionsbibliotheken gesammelt werden und sind dadurch wiederverwendbar

Funktionen

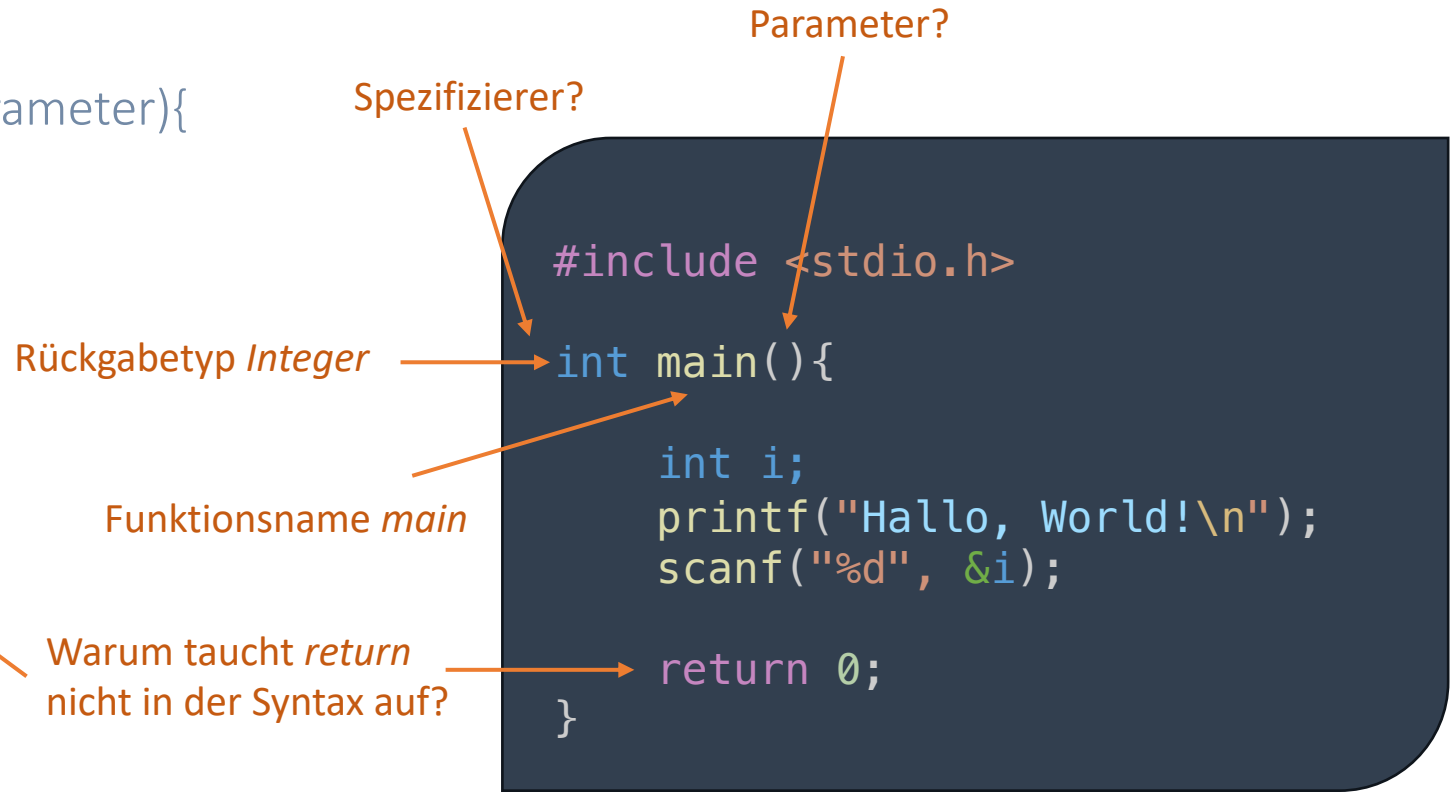
Wir haben schon verschiedene Funktionen kennengelernt:

- main-Funktion
- printf-Funktion
- scanf-Funktion



Syntax einer Funktionsdefinition

```
[Spezifizierer] Rückgabetyp Funktionsname(Parameter){
        // Anweisungsblock
}
```



Funktionen
Definition von Funktionen: Funktionsname

Syntax einer Funktionsdefinition

- **Funktionsname**: Mit dem Funktionsnamen wird die Funktion von einer anderen Stelle aufgerufen. Der Name muss daher eindeutig und kein Funktionsname aus einer Bibliothek sein (printf z.B.)

keine Parameter, also *void* (leer). Bei der Parameterliste kann void auch weggelassen werden.

```
void name(void){}
```

Rückgabetyp *void:* kein Rückgabewert

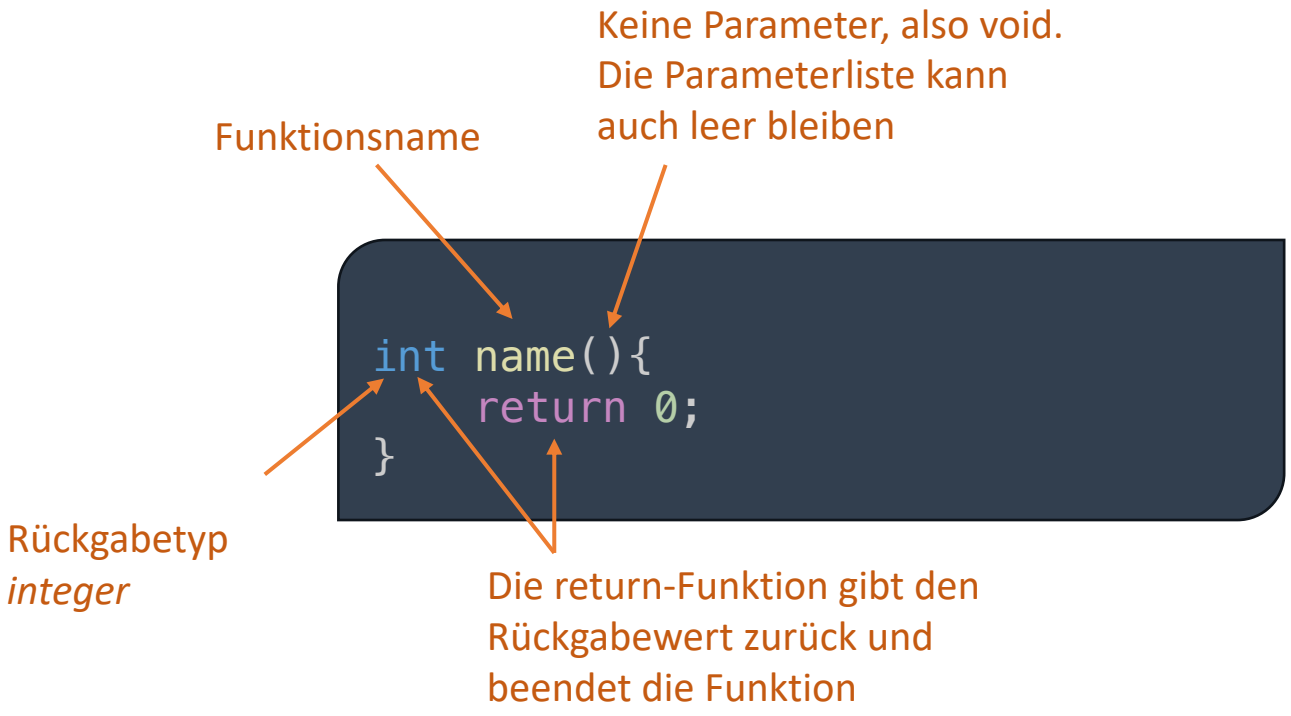
Funktionsname

Was könnte in einer Funktion ohne Parameter und ohne Rückgabetyp stehen?

Funktionen
Definition von Funktionen: Rückgabetyp & Return

Syntax einer Funktionsdefinition

- **Rückgabetyp**: Hier wird der Datentyp des Rückgabewerts festgelegt. Bei der Main-Funktion ist das z.B. eine Integer.
- **Return**: Return ist ein Schlüsselwort, das die Funktion beendet bzw. einen Wert zurückgibt und dann die Funktion beendet.
- Im Fall der Main-Funktion beendet Return das Programm.
- Der Rückgabetyp legt also fest, welcher Datentyp von der Funktion zurückgegeben wird (bzw. welche Art von Daten aus der Funktion nach außen gegeben wird)




```
int getIntFromFunction; // Variable

// Speichern des Rückgabewerts
getIntFromFunction = name();
```


Nicht-C-Programmierer: In C kann immer nur ein Wert zurückgegeben werden!

Funktionen
Definition von Funktionen: Parameter

Syntax einer Funktionsdefinition

- **Parameter:** Eine Funktion kann, muss aber keine Parameter aufnehmen.
- Parameter sind Werte, die von außen an die Funktion weitergereicht werden
- Eine Funktion kann mehrere Parameter verwenden. Beim Aufruf muss man dann auf die richtige Reihenfolge achten, sowie auf den Datentyp der Parameter

Parameter: Datenfluss in die Funktion hinein

```
int name(int var1, char var2){
    return 0;
}
```

Wertrückgabe: Datenfluss aus der Funktion hinaus

example_function_1.c, function_biggestNumber.c

Funktionen
Programmablauf

```
#include <stdio.h>

int addNumbers(int a, int b){
    return a + b;
}

int main(){

    int x = 3, y = 4;
    int result = 0;

    result = addNumbers(x, y);

    return 0;
}
```

1. Das Programm startet bei der Main-Funktion, welche sequentiell abgearbeitet wird

2. Bei Aufruf der addNumbers-Funktion springt das Programm in die addNumbers-Deklaration und arbeitet diese ab

3. Nachdem die addNumber—Funktion abgearbeitet wurde, springt das Programm zurück in die Main-Funktion und arbeitet den Rest des Codes ab

Funktionen
Funktionsdeklaration

Aufgerufene Funktionen müssen dem Programm immer bekannt sein, das heißt sie müssen **vor** ihrem Aufruf deklariert werden. Hierfür gibt es zwei Optionen:

- Vollständige Deklaration der Funktion vor der Main-Funktion
- Vorwärtsdeklaration

```c
#include <stdio.h>

int addNumbers(int a, int b){
    return a + b;
}

int main(){

    int x = 3, y = 4;
    int result = 0;

    result = addNumbers(x, y);

    return 0;
}
```

Funktionen
Funktionsdeklaration: Vorwärtsdeklaration

Aufgerufene Funktionen müssen dem Programm immer bekannt sein, das heißt sie müssen **vor** ihrem Aufruf deklariert werden. Hierfür gibt es zwei Optionen:

- Vollständige Deklaration der Funktion vor der Main-Funktion
- Vorwärtsdeklaration
  - Die Funktion wird ohne Anweisungsblock vor der Main-Funktion deklariert (Semikolon nicht vergessen!)
    → Funktionsprototyp
  - Die vollständige Deklaration der Funktion erfolgt dann nach der Main-Funktion
  - Eignet sich für umfangreichere Funktionen sowie Funktionen, die auch in anderen Funktionen aufgerufen werden

```c
#include <stdio.h>

int addNumbers(int a, int b);

int main(){

    int x = 3, y = 4;
    int result = 0;

    result = addNumbers(x, y);

    return 0;
}

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

Funktionen
Noch ein Wort zu Parametern

Syntax einer Funktionsdefinition

- **Parameter** sind Platzhalter für Werte, die von außen an die Funktion weitergereicht werden. Die Parameter werden in der Funktionsdeklaration festgelegt, hier (int a, int b)
- Beim **Aufruf** der Funktion, werden dann **Argumente** übergeben, hier (x, y)
- Parameter und Argumente "leben" nicht im selben *Geltungsbereich (scope)* des Programms, das heißt, Parameter und Argumente sind für einander nicht sichtbar (außer bei Call-by-Reference)

```c
#include <stdio.h>

int addNumbers(int a, int b);

int main(){

    int x = 3, y = 4;
    int result = 0;

    result = addNumbers(x, y);

    return 0;
}

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

Funktionen sind Unterprogramme, die der Struktur und Übersichtlichkeit dienen

Variante 1

```
void printHashes (void) {
        int numRows;
        for (numRows = 0; numRows < 5; numRows++) {
                int numHashes;
                for (numHashes = 0; numHashes < 5; numHashes++) {
                        printf("#");
                }
                printf("\n");
        }
}
```

Variante 2

```
void printHashes (void) {
        int numRows;
        for (numRows = 0; numRows < 5; numRows++) {
                printHashedLine();
        }
```

Code: Prof. Dr. Ruben Jubeh

Funktionen Zwischenfazit

- Funktionen sind Unterprogramme, die der Struktur und Übersichtlichkeit dienen
- Funktionen können Einzeiler sein oder selbst mehrere Funktionen beinhalten
- Funktionen können ganz ohne Parameter und Rückgabewert auskommen:

```
void printText(){
    print("blub");
}
```

oder Werte aufnehmen, diese verarbeiten und wieder ausgeben:

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

und alles zwischendrin.

Laufzeitbibliotheken enthalten häufig genutzte Funktionen, printf & scanf z.B., aber auch für mathematische Anwendungen, Grafik, Datum, Zeitrechnung, … Deshalb ist es immer sinnvoll erst einmal zu überprüfen, ob es eine Funktion schon gibt, bevor man sie selbst programmiert

Geltungsbereich der Variablen (Scope)

- Variablen in C sind nicht im ganzen Programm gültig bzw. sichtbar, sondern nur innerhalb ihrer Funktion / ihres Anweisungsblocks
- Das heißt, dass die Variablen x und y nur innerhalb der Main-Funktion bestehen. Die Variablen a und b wiederum bestehen nur innerhalb der Funktion addNumbers. Dies gilt auch, wenn die Variablen denselben Namen haben

```c
#include <stdio.h>

int addNumbers(int a, int b);

int main(){

    int x = 3, y = 4;
    int result = 0;

    result = addNumbers(x, y);

    return 0;
}

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

example_functionScope.c

Geltungsbereich von Funktionen (**Call-by-Value**)

- Call by Value: Der Grund für den beschränkten Geltungsbereich der Variablen in Funktionen ist, dass die Variablen mittels call-by-value übergeben werden
- Das heißt, dass der Funktion nur eine **Kopie der Variablen** als Parameter übergeben wird
- Anders gesagt: Für die Funktion wird beim Aufruf ein dynamischer Speicherbereich (stack frame) angelegt, auf dem die Kopien der Werte gespeichert werden. Ist die Funktion beendet, wird der Speicherbereich wieder gelöscht

```c
#include <stdio.h>

int addNumbers(int a, int b);

int main(){

    int x = 3, y = 4;
    int result = 0;

    result = addNumbers(x, y);

    return 0;
}

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

Geltungsbereich in Anweisungsblöcken

- Der beschränkte Geltungsbereich gilt auch für Anweisungsblöcke: Wird eine Variable **innerhalb eines Anwendungsblocks deklariert**, ist sie auch nur innerhalb des Blocks sichtbar. Wird der Variablen innerhalb des Blocks ein Wert zugewiesen, beeinflusst das eine andere, gleichnamige Variable außerhalb des Blocks nicht.
- Wird jedoch innerhalb des Anwendungsblocks eine außerhalb des Blocks deklarierte Variable verändert, so bleibt die Wertänderung bestehen

```c
int main(){

    int i = 22;

    if(i == 22){
        int i = 99;
        printf("Innerhalb der\
        If-Anweisung: %d\n", i);
    }

    printf("Außerhalb der If- \
    Anweisung: %d\n", i);

    return 0;
}
```

Die "lokalste" Variable gilt

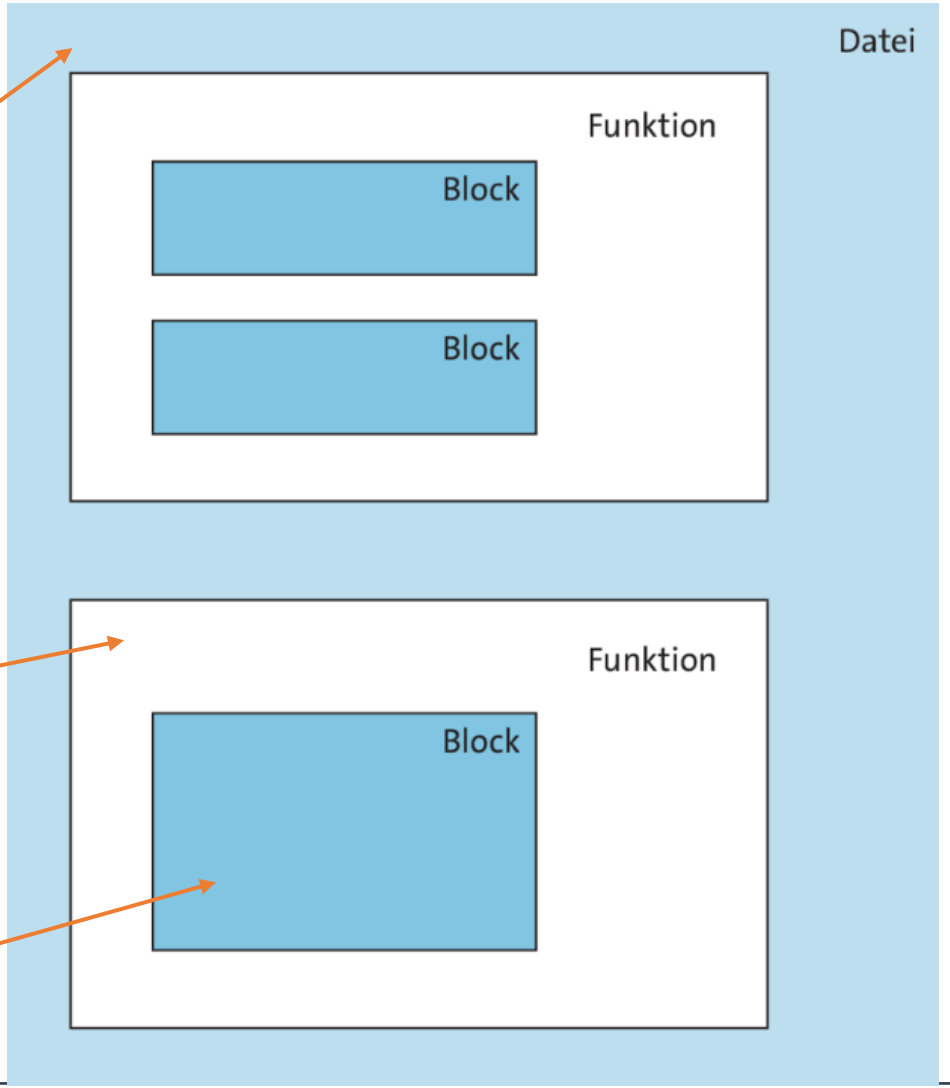
example_functionScopeIf.c

Funktionen
Funktionsgeltungsbereich

Die Lebensdauer und der Geltungsbereich einer Variablen hängt ab von:

- der Position der Deklaration
- dem Speicherklassen-Spezifizierer

Variablen, die außerhalb von Funktionen und Anweisungsblöcken deklariert werden, gelten vom Punkt der Deklaration bis zum Dateiende
→ globale Variablen



Funktionsvariablen gelten innerhalb der Funktion, es sei denn es wird eine lokalere Variable innerhalb eines Blocks deklariert – diese gilt nur innerhalb des Blocks

Variablen, die innerhalb eines Anwendungsblocks deklariert werden, gelten auch nur innerhalb des Blocks

example_scopeBrackets.c

Funktionen
Globale Variablen

- Sollen Variablen in der ganzen Datei, also auch in allen Funktionen, sichtbar sein, so können globale Variablen verwendet werden
- Globale Variablen werden mittels Vorwärtsdeklaration vor der Main-Funktion deklariert
- Aber auch bei globalen Variablen gilt, dass immer die "lokalste" Variable bevorzugt wird
- Grundsätzlich sollten globale Variablen sparsam eingesetzt werden, da sie zu Fehlern führen können

Funktionen
Globale Variablen

- Sollen Variablen in der ganzen Datei, also auch in allen Funktionen, sichtbar sein, so können globale Variablen verwendet werden
- Globale Variablen werden mittels Vorwärtsdeklaration vor der Main-Funktion deklariert
- Aber auch bei globalen Variablen gilt, dass immer die "lokalste" Variable bevorzugt wird
- Grundsätzlich sollten globale Variablen sparsam eingesetzt werden, da sie zu Fehlern führen können

Best Practices:

- Globale Variablen können dann sinnvoll sein, wenn die Variable read-only verwendet wird (also nur gelesen bzw. verarbeitet aber nicht geändert wird), bzw. eine Konstante ist
  - String-Literale wie Namen, GUI-Abmessungen, Default-Werte
- Variablen die im Verlauf des Programms geändert werden sollten nicht global sein! Genauso wenig wie "unwichtige" Variablen, Counter z.B.

example_functionGlobalVar.c, example_ functionGlobalVar_2.c

Funktionen
Statische Variablen

Statische Variablen ähneln globalen Variablen:

- globale Variablen: gültig im ganzen Programm, auch wenn dieses aus verschiedenen Dateien besteht
- statische Variablen: gültig in der ganzen Datei, aber nicht über Datei-Grenzen hinaus

Funktionen
Statische Variablen

Syntax der statischen Variablen: static int i = 98;

- Gültig in der ganzen Datei, aber nicht über Datei-Grenzen hinaus:
  Wird eine statische Variable innerhalb einer Funktion deklariert und initialisiert, behält die statische Variable ihren Wert, auch wenn die Funktion beendet wurde. Das heißt, statische Variablen werden nur einmal deklariert und initialisiert und existieren dann während der gesamten Programmausführung
- Statische Variablen müssen bei der Deklaration immer initialisiert werden

example_functionStaticVar.c

Funktionen
Konstante Variablen

- Konstante Variablen werden einmal initialisiert und können dann nicht mehr verändert werden
- Syntax: const int WERT = 10;
- Das Schlüsselwort *const* bezeichnet man als Typ-Qualifizierer
- Typischerweise schreibt man konstante Variablen in Großbuchstaben

Best Practices:

- Konstanten immer dann einsetzen, wenn ein Wert im Programm nicht geändert wird → Konstante Werte können vom Compiler besser optimiert werden
- Im Gegensatz zu globalen Variablen kann man mit Konstanten verschwenderisch umgehen

example_functionConstVar.c

Funktionen
Konstante Variablen

- Konstante Variablen können auch global auf Compiler-Level definiert werden, mittels
  #define NAME value

  also z.B.
  #define PI 3.14
  oder
  #define CELLS 24*8 //der Compiler ersetzt den Begriff CELLS im Code mit 24*8

- Welche Variante sinnvoller ist, hängt vom Einzelfall ab
- Für mathematische Konstanten (Pi z.B.) gibt es entsprechende Header-Dateien, z.B. math.h

example_functionConstVar_define.c

Funktionsgeltungsbereich & Variablen Zwischenfazit

- Der Geltungsbereich einer Variable hängt ab von
  - der Position, an der sie deklariert wird (Anweisungsblock, Funktion, global)
  - sowie dem Speicherklassen-Spezifizierer (static, const)
    Es gibt noch weitere Speicherklassen-Spezifizierer, die aber nur in Grenzfällen eingesetzt werden, wenn überhaupt (extern, auto, register)
- Globale Variablen sind überall im Programm (über Dateigrenzen hinweg) sichtbar
- Konstante Variablen werden einmalig initialisiert und können nicht mehr geändert werden
- Statische Variablen behalten ihren Wert auch außerhalb ihres Geltungsbereichs und sind in der ganzen Datei verwendbar

function_celciusToFahren.c, function_celciusToFahren_mod.c

Einschub
Speicherklassen-Spezifizierer für Funktionen

Neben Speicherklassen-Spezifizierer für Variablen gibt es auch welche für Funktionen

- *extern*: Funktionen, bei denen kein Spezifizierer angegeben wird, sind automatisch extern spezifiziert. Das heißt, die Funktion kann auch von einer anderen Datei aus gelesen werden
- *static*: Das Gegenteil von extern – die Funktion kann nur innerhalb der Datei genutzt werden
- *volatile*: Verhindert, dass der Compiler den Quellcode optimiert

Rekursion

Rekursion

- Eine rekursive Funktion ist eine Funktion, die sich selbst immer wieder neu aufruft und sich selbst immer wieder neu definiert
- Wirklich einen Einsatzzweck haben Rekursionen z.B. bei Suchalgorithmen, Sortierverfahren und KI-Algorithmen
  Das heißt, die in der Vorlesung gezeigten Beispiele könnte man ohne Rekursion besser lösen
- Um Rekursion zu verstehen, muss man einerseits den Algorithmus an sich und andererseits die Speicherverwaltung von Funktionsaufrufen verstehen

Rekursion
Der Algorithmus

Das ist keine generelle Vorlage für rekursive Algorithmen, sondern soll die Idee illustrieren

```
int recursiveFunction(int x, int y){
    if(Bedingung){
        return(wert + recursiveFunction(xChanged, y));
    return 0;
    }
}
```

Wird die Abbruchbedingung erfüllt, wird ein letzter Wert oder ein Text ausgegeben

Bei jedem Durchlauf wird ein Wert "gespeichert" (siehe Folien zum Stack)

Solange die Abbruchbedingung nicht erfüllt wird, ruft sich die rekursive Funktion neu auf, mit veränderten Parametern

Rekursion
Beispiel: Division ohne Division

Funktionsdeklaration

Solange $x \geq y$ ist, wird der Wert 1 gespeichert und die Funktion erneut aufgerufen. Diesesmal mit $x = (x-y)$, y behält seinen Wert bei:
x = 6, y = 2
return 1 + (divide(4, 2))
return 1 + (divide(2, 2))
return 1 + (divide(0, 2))
return 0
→ 1 + 1 + 1 + 0 = 3

Ist $x \neq 0$, aber $x < y$, dann ist x nicht durch y ohne Rest teilbar.
In diesem Fall wird der Wert ausgegeben, den x bei diesem Durchgang hat

Die Abbruchbedingung
Ist $x < y$, dann wird die rekursive Funktion beendet und eine 0 ausgegeben. Im Hintergrund werden alle bisher ausgegebenen Werte zusammengezählt und als return an die Main-Funktion zurückgegeben.

```c
int divide(int x, int y){
    if(x >= y){
        // printf("x: %d\n", x);
        return(1 + divide(x - y, y));
    }
    if(x){
        printf("\nZahl nicht teilbar");
        printf("Rest der Division: %d\n", x);
    }
    return 0;
}

int main(){
    printf("Das Ergebnis ist: %d \n\n", \
    divide(10, 2));

    return 0;
}
```

Funktionsaufruf

recursion_division.c

Rekursion
Beispiel: Berechnung der Fakultät

Solange n > 0 ist, wird der Wert n "gespeichert" und die Funktion erneut aufgerufen, mit n-1:
n = 4;
return 4 * factorial(3)
return 3 * factorial(2)
return 2 * factorial(1)
return 1 * factorial(0)
return 1
→ 4*3*2*1*1

Ist n = 0, so wird eine 1 ausgegeben (Da 0! = 1; könnte man aber auch weglassen, da es keinen Einfluss auf das Ergebnis hat)

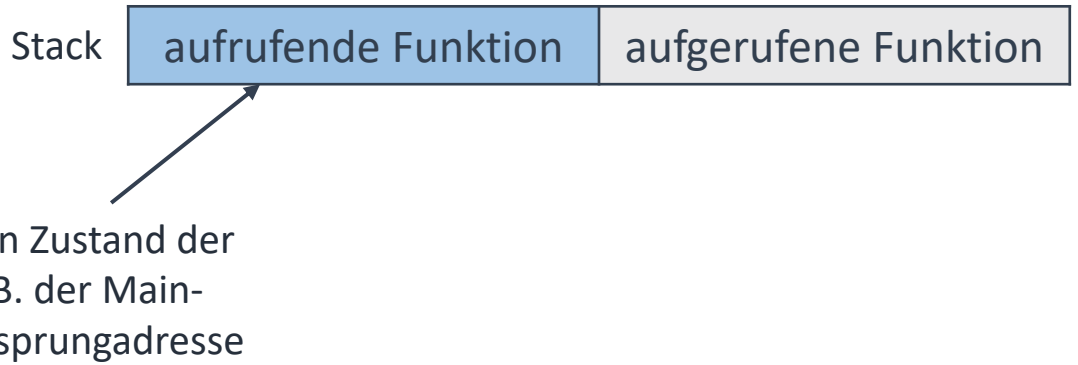
```c
long int factorial(long int n){
    if(n){
        return n * factorial(n-1);
        }
    return 1;
}

int main(){
    printf("Die Fakultät von 10 ist: %ld \n\n",\
    factorial(10));
    return 0;
}
```

recursion_factorial.c

Rekursion
Stack

Um Rekursionen genauer zu verstehen, ist es notwendig einen Einblick in die Speicherverwaltung zu gewinnen

- **Stack**: Speicherbereich, der Funktionsaufrufe verwaltet
- Der Stack ist ein dynamischer Speicher, der je nach Bedarf wächst und schrumpft
- Wird eine Funktion aufgerufen, werden alle für den Funktionsaufruf notwendigen Daten im Stack abgelegt
- **Datenblock / Stack Frame**: Der Block im Speicher, der die Funktionsparameter sowie die Rücksprungadresse der aufrufenden Funktion enthält



Rekursion
Stack

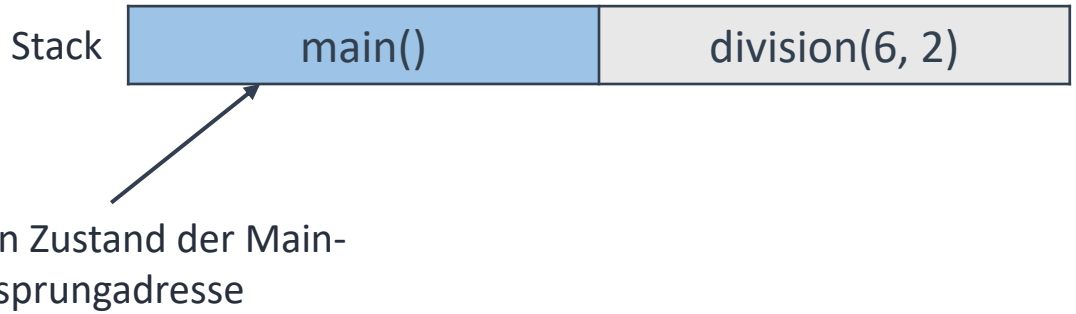
Um Rekursionen genauer zu verstehen, ist es notwendig einen Einblick in die Speicherverwaltung zu gewinnen

- **Stack**: Speicherbereich, der Funktionsaufrufe verwaltet
- Der Stack ist ein dynamischer Speicher, der je nach Bedarf wächst und schrumpft
- Wird eine Funktion aufgerufen, werden alle für den Funktionsaufruf notwendigen Daten im Stack abgelegt
- **Datenblock / Stack Frame**: Der Block im Speicher, der die Funktionsparameter sowie die Rücksprungadresse der aufrufenden Funktion enthält
- Wird die Funktion beendet, wird der Datenblock gelöscht
- Wird jedoch innerhalb der Funktion eine weitere Funktion aufgerufen (wie das bei rekursiven Funktionen der Fall ist) wird ein weiterer Datenblock auf den ersten Block gepackt

Wird jedoch innerhalb der Funktion eine weitere Funktion aufgerufen (wie das bei rekursiven Funktionen der Fall ist) wird ein weiterer Datenblock auf den ersten Block gepackt

Am Beispiel der Division:

1. Aufruf



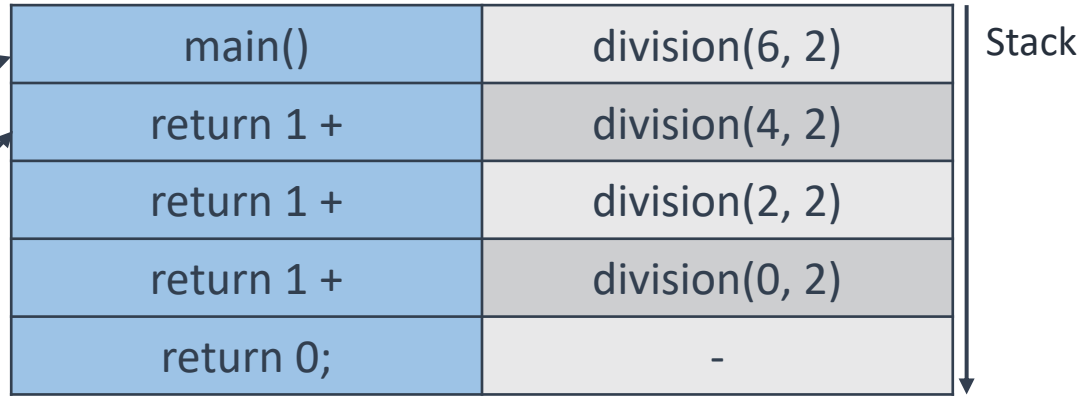
Wird jedoch innerhalb der Funktion eine weitere Funktion aufgerufen (wie das bei rekursiven Funktionen der Fall ist) wird ein weiterer Datenblock auf den ersten Block gepackt

Am Beispiel der Division:

2.-4. Aufruf

Enthält den eingefrorenen Zustand der Main-Funktion, sowie die Rücksprungadresse

Enthält den Rückgabewert 1 sowie die Rücksprungadresse

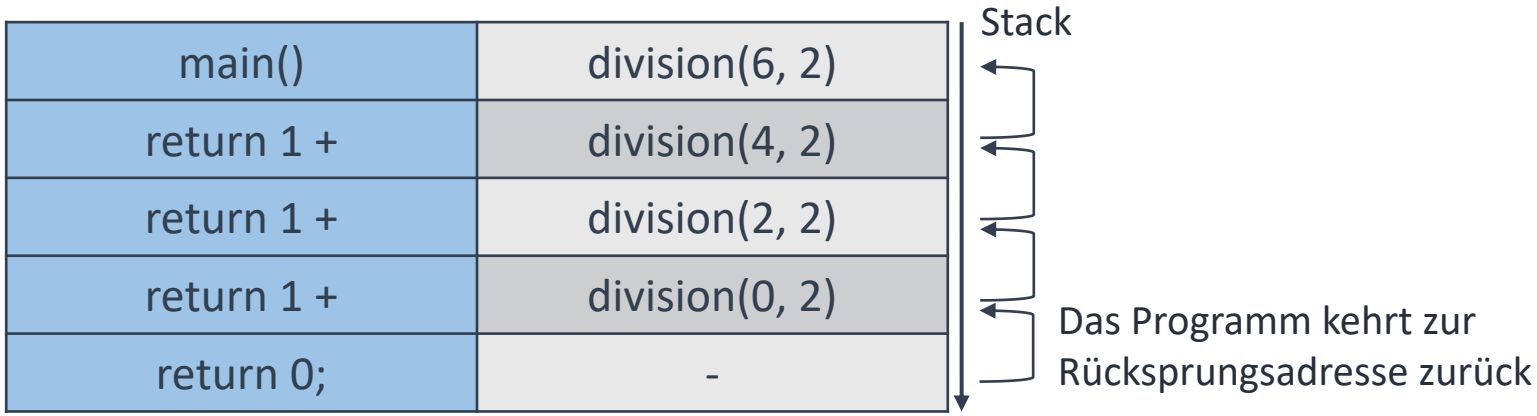


Rekursion
Stack

Wird jedoch innerhalb der Funktion eine weitere Funktion aufgerufen (wie das bei rekursiven Funktionen der Fall ist) wird ein weiterer Datenblock auf den ersten Block gepackt. Wird die Funktion beendet, wird der Stack rückabgewickelt

Am Beispiel der Division:

Rückabwicklung: 0 + 1 + 1 + 1 = 3



Hausaufgabe zu rekursiven Funktionen

- Ich würde Ihnen grundsätzlich raten, den Vorlesungsstoff sowie die gezeigten Code-Beispiele daheim durchzugehen und nachzucoden
- Bei rekursiven Funktionen würde ich Ihnen ganz besonders dazu raten: Gehen Sie die Beispiele durch, nutzen Sie den Debugger, versuchen Sie die Funktionen eigenständig nachzucoden, auf dem Papier und im Texteditor / der IDE
- Schreiben Sie die rekursiven Funktionen so um, dass sie ohne Rekursion gelöst werden, überlegen Sie sich Beispiele in denen rekursive Funktionen sinnvoll sein könnten
- In der Praxis ist es nicht oft so, dass eine rekursive Funktion die beste Wahl ist – aber es kommt immer mal wieder vor

Fazit

- Kontrollstrukturen brechen den sequentiellen Ablauf von Programmen auf
- Funktionen sind ein Mittel um den Code sinnvoll zu strukturieren und mehrfach verwendete Funktionalitäten nur einmal zu implementieren
- Der Geltungsbereich von Variablen hängt von verschiedenen Faktoren ab, Art der Variable, Ort der Deklaration, …
- Speicherklassen-Spezifizierer können für Variablen verwendet werden, die einen bestimmten Zweck erfüllen, Konstanten oder globale Variablen z.B.
- Rekursion: Eine Funktion, die sich selbst wiederholt aufruft

Wie geht man ein Programmierproblem an

- Beginnen Sie mit der grundsätzlichen Funktion und überlegen Sie sich grob, wie Sie diese umsetzen können
- Zu diesem frühen Zeitpunkt ist es noch nicht so wichtig, auf Struktur und Best Practices zu achten, sehen Sie es als Ideensammlung
- Versuchen Sie die grundlegende Funktion zu implementieren, angefangen mit der wichtigsten
- Testen Sie die Funktionsweise mittels printf / Debugger
- Räumen Sie den Code auf:
  - Refrakturieren (refactoring): Der systematische Prozess, Code zu verbessern ohne neue Funktionalitäten hinzuzufügen (clean code)
  - Variablen- und Funktionsnamen verbessern, Kommentare überarbeiten (hinzufügen), Überflüssiges entfernen, etc.
- Fügen Sie weitere Funktionen hinzu bzw. überlegen Sie sich was für den Anwender / die Anwenderin / die Funktionsweise des Programms sinnvolle Ergänzungen wären
- Code aufräumen
- …

Best Practices

Best practices sind immer nur Anregungen, keine Gesetze

- Kontrollsequenzen sollten nicht (mehrfach) verschachtelt werden: schlecht zu lesen & oft kann das Problem in mehrere simple Funktionen aufgeteilt werden
  Versuchen Sie mehr als zwei Einrückungslevel zu vermeiden
- Der Funktionsname sollte Auskunft darüber liefern, was in der Funktion passiert
  z.B. readData(), addNumbers(), printName(), …
- Eine Funktion sollte immer nur eine Aufgabe erledigen
  Also nicht: readWriteSaveData()

→ Gut gewählte Namen geben einen Rückschluss darauf, ob der Umfang der Funktion sinnvoll ist oder nicht. Hat man z.B. zwei Funktionen mit fast gleichen Namen, so könnte es sinnvoll sein sie zusammenzulegen

- Schreiben Sie Ihren Code so, dass ihn auch eine andere Person versteht (im Zweifel Sie selbst in zwei Wochen)
  - Variablen- und Funktionsnamen sollten einen kommentierenden Charakter haben
  - Werden Werte übergeben, sollte klar sein von welchem Datentyp sie sein dürfen
  - Nutzen Sie Abkürzungen sehr sparsam
  - Kommentieren Sie wo sinnvoll (oft vergisst man, die Kommentare zu aktualisieren)