Programmieren in C
für Elektrotechniker

Kapitel 6: Funktionen

- **Eigene Funktionen**
- Unterprogramme
- Gültigkeit von Namen
- Rekursion und Iteration
- Modularer Quellcode

Definition eigener Funktionen

- Größere Programm in Funktionen strukturieren/unterteilen.
  - übersichtlicherer Code
  - Wiederverwendung von Code
  - parallele Entwicklung

- Beispiel

Eingabeparameter

$$summe = \sum_{i=1}^{n} i$$

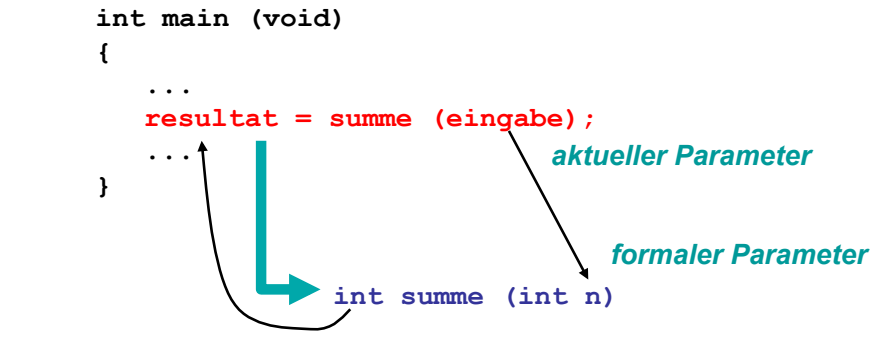
```
#include <stdio.h>

int summe (int n)
{
   int i;
   int zwsumme = 0;
   for (i = 1; i <= n; i++)
      zwsumme += i;
   return zwsumme;
}

int main (void)
{
   int eingabe;
   int resultat;
   printf ("Eingabe der oberen Grenze: ");
   scanf ("%d", &eingabe);
   resultat = summe (eingabe);
   printf ("Summe 1 bis %d: %d\n", eingabe, resultat);
   return 0;
}
```


- C-Funktionen bestehen aus
  - **Funktionskopf**
    <Rückgabewert> <Funktionsname> (<Parameterliste mit Datentypen>).
    Beispiel: `int summe (int n)`
    → Schnittstelle: Sicht von außen (des Aufrufers)
  - **Funktionsrumpf**

```
{
   int i;
   int zwsumme = 0;
   for (i = 1; i <= n; i++)
      zwsumme += i;
   return zwsumme;
}
```

→ lokale Variablendefinitionen
→ Anweisungen

- Funktionsaufruf



Funktionsergebnis (→ `return`-Anweisung)

- Vorteil Kapselung
  - **Realisierung der Funktion ist für Aufrufer ohne Bedeutung**
    → Realisierung leicht austauschbar
  - Beispiel
    Summe über Gaußformel $\sum_{i=1}^{n} i = \frac{n}{2}(n+1)$

```
int summe (int n)
{
   int i;
   int zwsumme = 0;
   for (i = 1; i <= n; i++)
      zwsumme += i;
   return zwsumme;
}
```

```
int summe (int n)
{
   return n * (n + 1) / 2;
}
```

- **Funktionen (Vertiefung)**
  - Zusammenfassung von Teilfunktionalitäten eines Programms unter einem sinnvollen Namen.
    - Strukturierung eines Programms
    - Modularisierung
    - Mehrfachverwendung *(mit i.d.R. verschiedenen Parametern)*
  - Mathematische Funktion: $y = f(\vec{x})$
  - C-Funktion: `<Ausgabe> = f (<Eingabe>)` mit
    - Eingabedaten: - an (formale) Parameter übergebene Werte (aktuelle Parameter)
      - Werte globaler Variablen
    - Ausgabedaten: - Rückgabewerte der Funktion
      - Änderung von Variablen, deren Adressen als Parameter übergeben wurden
      - Änderung von globalen Variablen
  - Tipp (SW-Engineering-Regel):
    Möglichst keine oder nur wenige globale Variablen verwenden,
    da Kapselung und Übersicht verloren gehen und
    Fehler schwerer zu entdecken sind.

- **Syntax**

```
<Rückgabewert> funktionsname (typ_1 form_par_1,
                              typ_2 form_par_2,
                              ...
                              typ_n form_par_n)
{
 ...
}
```

Funktions-kopf

Funktions-rumpf

- In C kann die Parameterliste variabel lang sein.
  → Auslassung bzw. Ellipse *(s.u.)*.
  → Beispiel: `printf(…)` kann beliebig viele Argumente ausgeben.
- Keine Eingabeparameter: Angabe des Datentyps `void`.
  Beispiel:

```
int fkt_ohne_parameter (void)
{
   ...
}
```

Aufruf:

```
fkt_ohne_parameter();
```

- **Syntax**

```
<Rückgabewert> funktionsname (typ_1 form_par_1,
                              typ_2 form_par_2,
                              ...
                              typ_n form_par_n)
{
 ...
}
```

  - Funktion ohne Rückgabewert *(→ nur Seiteneffekt)*: Funktionstyp **`void`**
    - „Prozedur“ in anderen Programmiersprachen
    - **`return`** ohne Datum
    - Beispiel:

```
void fkt_ohne_rueckgabe (…)
{
   ...
   return;
}
```

  - In der Praxis nur selten, da meist Rückgabe von Fehlercodes.
  - Fehlt der Funktionstyp, wird implizit **`int`** angenommen.

- **Parameterzuweisung**
  - Formale Parameter sind spezielle lokale Variablen.
    → unterschiedliche Bezeichner verwenden

```
void nichtgut (int x)
{
    int x;
    . . . .
}
```

  - Beim Aufruf einer Funktion mit Parametern findet eine Zuweisung statt:

```
typ_n formaler_parameter_n = aktueller_parameter_n;
```

  - Beispiel:

```
doSomething (float zahl)
{
   ...
}
doSomething (3.14);
```

entspricht

Start von doSomething

```
{ float zahl = 3.14;
   ...
}
```

    - Variable vom Typ **`typ_n`** wird auf den Stack *(stack frame der Funktion)* angelegt *(s.u.)*.
      Der Wert des aktuellen Parameters wird dieser Variablen zugewiesen.

- **Parameterzuweisung**
  - Beispiel:

```
doSomething (float zahl)          Start von doSomething
{                                 { float zahl = 3.14;
   ...          entspricht            ...
}                                 }
doSomething (3.14);
```

  - **call by value**
    - Es wird ein Wert übergeben.
    - Die Funktion arbeitet auf einer lokalen Kopie
      (der aktuelle Parameter wird nicht verändert).
    - Der aktuelle Parameter kann auch ein (komplexer) Ausdruck sein.
  - Die Anzahl aktueller Parameter muss mit der Anzahl formaler Parameter übereinstimmen.
  - Die Typen der aktuellen und formalen Parameter müssen nicht übereinstimmen.
    → implizite Typwandlung soweit möglich (wie bei Zuweisung).
  - Tipp: Besser explizite Typwandlung durch Cast-Operator.

- **Ergebnisrückgabe**

→ **`return`**-Anweisung

- Falls Funktionstyp ≠ void,
  sollte **`return`**-Statement einen passenden Wert zurückgeben.

```
return <Ausdruck>;
```

- Rückgabetyp von **`<Ausdruck>`** sollte mit dem Funktionstyp übereinstimmen
  → ggf. implizite Typwandlung.
- Falls Funktionstyp nicht **`void`** ist und es keinen Rückgabewert gibt,
  ist undefiniert, welcher Wert vom „Aufrufer“ verwendet wird.
- Besitzt eine Funktion kein **`return`**-Statement, wird die Funktion beim Erreichen des Endes des Funktionsrumpfs (‚`}`‘) beendet.
  → nur bei Funktionstyp **`void`** sinnvoll
- „Aufrufer“ muss den Rückgabewert einer Funktion nicht abholen.
  Beispiel: `printf()` gibt Anzahl ausgegebener Zeichen zurück
  → wird meist ignoriert.

call by reference

- Sind formale Parameter Referenzparameter, so erfolgt jede Operation auf den Referenzparametern tatsächlich auf den zugehörigen aktuellen Parameter.
  → Alias-Namen für die referenzierten Variablen (aktuelle Parameter).
- In C wird call by reference durch Übergabe von Pointern nachgebildet
  → call-by-value-Übergabe von Adressen
- Beispiel:

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

  → Parameterübergabe entspr. `int * alpha = &a;`
  → `*alpha = 10;` ändert den Wert von `a` auf `10`.

Vorwärtsdeklaration von Funktionen: Funktionsprototypen

- Funktion (-Kopf) muss vor Aufruf der Funktion bekannt sein.
  → Eigentliche Funktion (Rumpf) kann später definiert werden.
- Beispiel:

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

- Funktionsprototyp entspr. Funktionskopf mit
  - Namen der formalen Parameter müssen nicht übereinstimmen *(gleicher Typ genügt)*
  - Namen der formalen Parameter können im Funktionsprototyp weggelassen werden.
    → Funktionsprototyp `void init (int *);` ist ausreichend.

Vorwärtsdeklaration von Funktionen: Funktionsprototypen

- Bibliotheksfunktionen
  - Mit den header-Dateien **`*.h`** werden die Prototypen der entsprechenden Bibliotheksfunktionen eingebunden.
    Beispiel: **`#include <stdio.h>`**
    → Präprozessor kopiert die Funktionsprototypen aus **`stdio.h`** an diese Stelle.

- Fehlende Funktionsprototypen
  - Fehlt ein Funktionsprototyp, so wird auf die C-Version von Kernighan&Ritchie zurückgeschaltet *(evtl. Warnung)*
    → implizite Funktionsdeklaration
    - Ausgabetyp: **`int`**
    - Ausschalten der Überprüfung der Parameter

Vorwärtsdeklaration von Funktionen: Funktionsprototypen

- Beispiele

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

→ Compiler-Fehler, da die Typen der Funktion ungleich sind.

▪ Vorwärtsdeklaration von Funktionen: Funktionsprototypen

- Beispiele

```
/*Datei: quadrat1.c */
#include <stdio.h>
int main (void) {
    int resultat;
    double x = 5;
    resultat = quadrat (x);
    printf ("%d", resultat);
    return 0;
    }
/*Datei: quadrat2.c */
double quadrat (double n) {
    return n * n;
}
```

Keine Compiler-Reaktion, da in separaten Dateien

(→ Typen während Übersetzung unbekannt)

→ zur Laufzeit falsche Ausgabe, da keine implizite Typwandlung möglich ist.

▪ Ellipse „…“ – variable Parameteranzahl

- Eine Funktion muss mindestens einen explizit angegebenen Parameter enthalten.
- Danach ist eine variable Anzahl von Parametern durch Angabe der Ellipse „…“ möglich.
- Beispiel:

```
int var_func (int zahl1, double zahl2, ...);
....
int z1 = 3;
double z2 = 5.4;
var_func (z1, z2); /* keine zusaetzlichen Parameter */
var_func (z1, z2, "String"); /* 1 zusaetzlicher String */
var_func (z1, z2, 19, 27); /* 2 zusaetzliche Integerwerte */
var_func (z1); /* !! Fehler: nur 1 Parameter !! */
```

→ Keine Typprüfung der ausgelassenen Parameter möglich.

▪ Ellipse „…“ – variable Parameteranzahl

- Problem: Anzahl aktueller Parameter ist in der Funktion unbekannt.
  - entweder festen Parameter für die Anzahl verwenden
  - oder Bibliothek **<stdarg.h>** verwenden
    → stellt Hilfen **va_start**, **va_arg**, **va_end** bereit.
- Beispiel:

```
const double SCHWELLE = 3.0, ENDE = -1;

double qualitaet (double schwellwert, ...) {
    int schlecht = 0;  /* Anzahl schlechter Teile */
    double wert = 0.;
    int i = 0;
    va_list listenposition;
    for (va_start (listenposition, schwellwert);
         (wert = va_arg (listenposition, double)) != ENDE;
         i = i+1) {
        if (wert > schwellwert) schlecht = schlecht + 1;
    }
    va_end (listenposition);
    return (double) schlecht / i;
}

int main (void) {
    printf ("\n\nDer Ausschuss betraegt %5.2f %%", 100 *
            qualitaet (SCHWELLE, 2.5, 3.1, ENDE));
    printf ("\nDer Ausschuss betraegt %5.2f %%", 100 *
            qualitaet (SCHWELLE, 4.2, 3.8, 3.4, 2.9, ENDE));
}
```

Datenstruktur für variable Parameter

Initialisiert Zeiger auf ersten var. Parameter

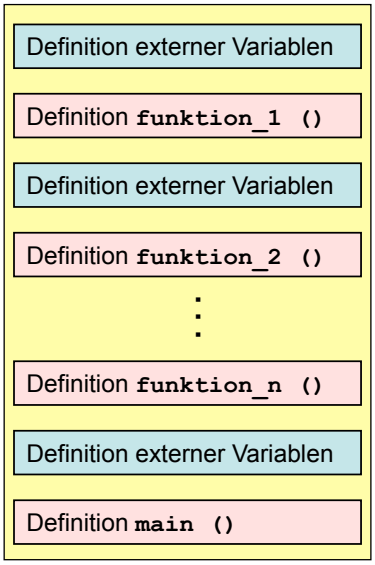
liefert Wert des Parameters „listenposition“ und setzt Zeiger auf nächsten var. Parameter

Gibt Speicher für var. Parameter wieder frei

▪ Struktur einer Quelldatei

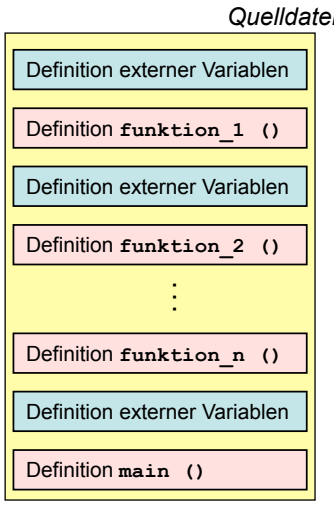
- C-Programme können auf mehrere Dateien aufgeteilt werden
  → modularer Entwurf
  - Vorteile: - Übersichtlichkeit
    - Arbeitsteilung.
- Funktionen müssen komplett in einer Datei stehen.
- C-Programme bestehen im Wesentl. aus Definitionen von Funktionen und globalen (externen) Variablen.

Quelldatei



Struktur einer Quelldatei

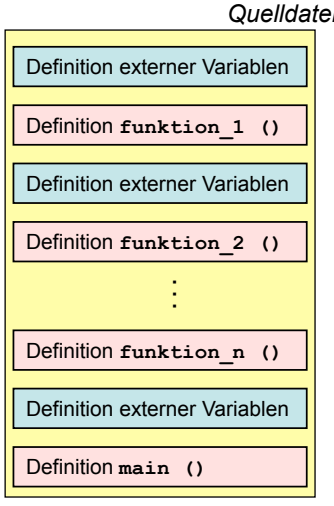
- C-Programme bestehen im Wesentl. aus Definitionen von Funktionen und globalen (externen) Variablen.
  - Externe bzw. globale Variablen: Variablen, die für alle Funktionen sichtbar sind.
  - Interne bzw. lokale Variablen: Variablen, die innerhalb von Funktionen definiert und sichtbar sind.
- Funktionen in C können nicht geschachtelt werden
  → Funktionen in C sind immer extern zu anderen.



- Compiler liest und prüft eine Quelldatei immer von vorne nach hinten
  → alle Namen müssen vor Verwendung definiert sein
  → benutzt eine Funktion eine externe Variable, muss die Variablendefinition vor der Funktionsdefinition stehen
  → Definition aufgerufener Funktionen vor Definition aufrufender Funktionen.

Struktur einer Quelldatei

- Compiler liest und prüft eine Quelldatei immer von vorne nach hinten
  → Definition aufgerufener Funktionen vor Definition aufrufender Funktionen.
- Problem bei zyklischen Aufrufen:
  `function_a` ruft **`function_b`** und
  `function_b` ruft **`function_a`** auf
  → *Was tun?*
- Verwendung von Funktionsprototypen *(s.o.)*:
  nur Funktionskopf und „;“
  z.B. `int summe (int a, b);`

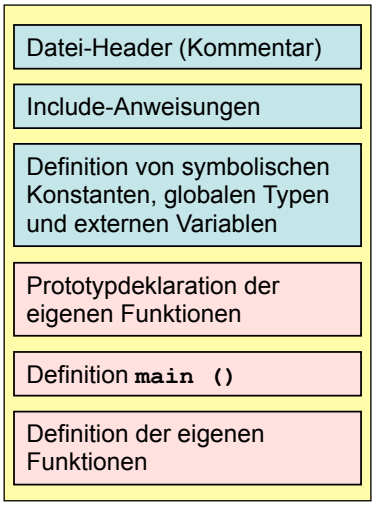


Struktur einer Quelldatei

- Weitere Komponenten einer Quelldatei
  - Präprozessoranweisungen
    - #include *(s.o.)*
    - #define *(→ symbolische Konstanten)*
  - Globale Typdefinitionen
  - Funktionsprototypen *(s.o.)*

→ Quelldatei sollte wie hier strukturiert sein

Quelldatei



Programmieren in C
für Elektrotechniker

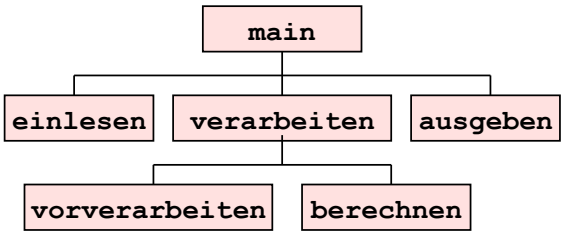
Kapitel 6: Funktionen

- Eigene Funktionen
- **Unterprogramme**
- Gültigkeit von Namen
- Rekursion und Iteration
- Modularer Quellcode

Unterprogramme

Unterprogramme

- Mittel zur Programmstrukturierung
  → Programme sollten modular geschrieben werden
- Hauptprogramm: - wird zu Programmstart ausgeführt (→ **`main`**)
  - ruft (meist) Unterprogramme auf
- Unterprogramm: - wird über seinen Namen aufgerufen
  - kann wiederum weitere Unterprogramme aufrufen
- Aufrufhierarchie eines Programms



Hauptprogramm

Unterprogramme

Unterprogramme

- Vorteile
  - Wiederverwendung von Programmteilen.
  - Abstraktion von Details *(s.o.)*.
- Bibliotheken
  - Zusammenfassung von Unterprogrammen aus zusammenhängenden Aufgabengebieten.
  - Werden meist mehreren Programmen zur Verfügung gestellt.
  - Einige Standard-Bibliotheken werden mit dem Compiler ausgeliefert (z.B. **`stdio`**), andere werden speziell angeboten.

▪ Unterprogramme

- Zwei Klassen von Unterprogrammen
  - Funktionen *(s.o.)*

    <Rückgabetyp> <Funktionsname> (<Parameterliste mit Datentypen>)

    - z.B. `int summe (int a, b)`
  - Prozeduren
    - nicht in C
    - ohne Typ des Ergebnisses
    - ggf. Ergebnisrückgabe über Parameterliste

▪ Unterprogramme

- Schrittweise Verfeinerung
  - Aufrufhierarchie wird meist top-down entworfen:
    Zerlegung eines Problems in Teilprobleme
    → grobe Strukturen werden immer weiter verfeinert
  - Aufrufhierarchie kann auch bottom-up entworfen werden
    → vgl. Bibliothekskonzept

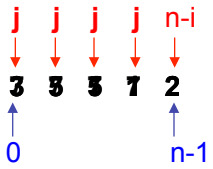
▪ Unterprogramme auf Prozessorebene

Beispiel: Bubble Sort
Sortieren einer Liste von Zahlen durch paarweises Vertauschen.

```
int b[];

bubblesort(int n) {
// Liste b[0..n-1] durchlaufen und
// evtl. benachbarte Elemente austauschen
int i, j;
int x, y;
for (i=1; i<n; i++)
   for (j=0; j<n-i; j++) {
      x = b[j]; y = b[j+1];
      swapIfGreater(x, y);
      b[j] = x; b[j+1] = y;
      }
}
```

Beispiel:

i = 1



Programmieren in C
Exkurs: Prozessorarchitektur
(nicht klausurrelevant)

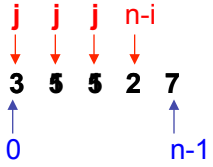
▪ Unterprogramme auf Prozessorebene

Beispiel: Bubble Sort
Sortieren einer Liste von Zahlen durch paarweises Vertauschen.

```
int b[];

bubblesort(int n) {
// Liste b[0..n-1] durchlaufen und
// evtl. benachbarte Elemente austauschen
int i, j;
int x, y;
for (i=1; i<n; i++)
   for (j=0; j<n-i; j++) {
      x = b[j]; y = b[j+1];
      swapIfGreater(x, y);
      b[j] = x; b[j+1] = y;
      }
}
```

Beispiel:

i = 2



6. Funktionen
Bernd Schürmann

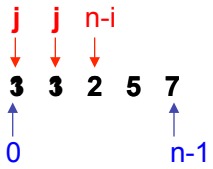
▪ Unterprogramme auf Prozessorebene

Beispiel: Bubble Sort
Sortieren einer Liste von Zahlen durch paarweises Vertauschen.

```
int b[];

bubblesort(int n) {
// Liste b[0..n-1] durchlaufen und
// evtl. benachbarte Elemente austauschen
int i, j;
int x, y;
for (i=1; i<n; i++)
   for (j=0; j<n-i; j++) {
      x = b[j]; y = b[j+1];
      swapIfGreater(x, y);
      b[j] = x; b[j+1] = y;
      }
}
```

Beispiel:

i = 3



Programmieren in C
Exkurs: Prozessorarchitektur
(nicht klausurrelevant)

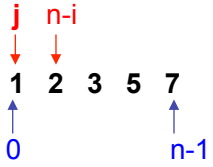
▪ Unterprogramme auf Prozessorebene

Beispiel: Bubble Sort
Sortieren einer Liste von Zahlen durch paarweises Vertauschen.

```
int b[];

bubblesort(int n) {
// Liste b[0..n-1] durchlaufen und
// evtl. benachbarte Elemente austauschen
int i, j;
int x, y;
for (i=1; i<n; i++)
   for (j=0; j<n-i; j++) {
      x = b[j]; y = b[j+1];
      swapIfGreater(x, y);
      b[j] = x; b[j+1] = y;
      }
}
```

Beispiel:

i = 4



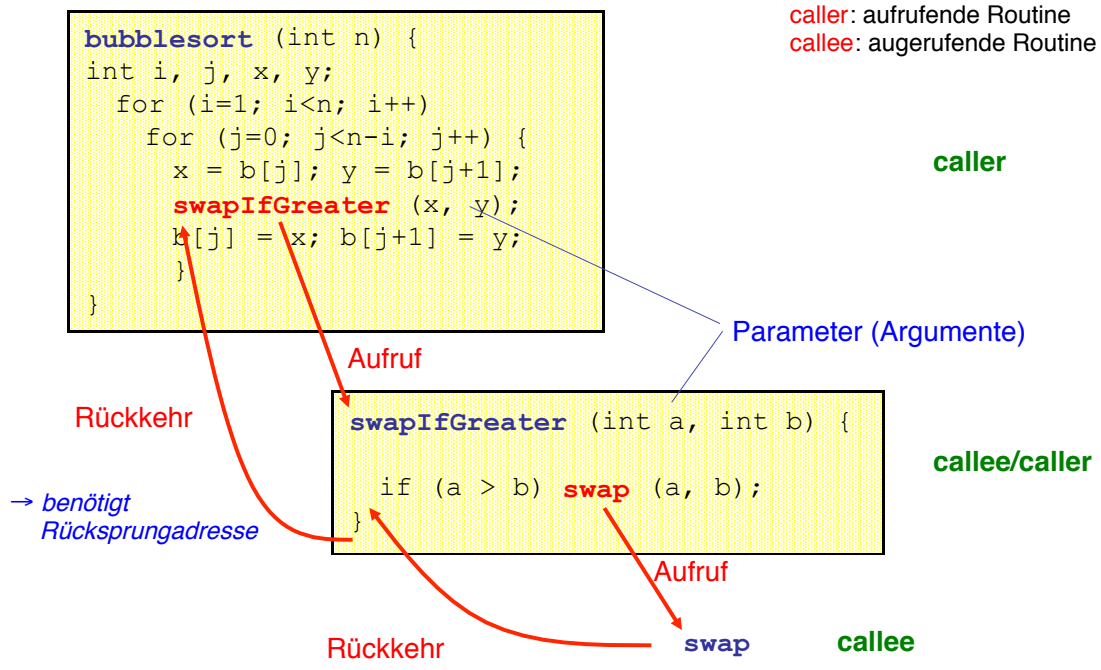
▪ Unterprogramme auf Prozessorebene

Beispiel: Bubble Sort
Sortieren einer Liste von Zahlen durch paarweises Vertauschen.

```
int b[];

bubblesort(int n) {
// Liste b[0..n-1] durchlaufen
// evtl. Elemente austauschen
int i, j;
int * x, * y;
for (i=1; i<n; i++)
   for (j=0; j<n-i; j++) {
      x = &b[j]; y = &b[j+1];
      swapIfGreater(x, y);
      b[j] = x; b[j+1] = y;
      }
}
```

```
swapIfGreater(int* a, int* b) {
   if (*a > *b) swap(a, b);
}

swap(int* n, int* m) {
int tmp;
   tmp = *n;
   *n = *m;
   *m = tmp;
}
```

"call by reference":
→ Übergabe der Adressen/Pointer



Unterprogramm (UP) / Funktion:

Abgeschlossenes Programmfragment (Routine), das
- von „Caller“ über Parameter mit Daten versorgt und gestartet wird,
- auf lokalen (und globalen) Daten arbeitet,
- bei Beendigung die Ergebnisse in Rückgabeparameter schreibt und
- zur Aufrufstelle zurückspringt.

⇒ Schritte bei UP-Ausführung:

1. (Caller) Eingabeparameter so speichern, dass UP *(Callee)* darauf zugreifen kann.
2. (Caller) Übergabe der Kontrolle ans UP *(an Callee)*.
3. (Callee) Anforderung von Speicherplatz für UP-Ausführung.
4. (Callee) UP ausführen.
5. (Callee) Ausgabeparameter so speichern, dass aufrufende Routine *(Caller)* darauf zugreifen kann.
6. (Callee) Übergabe der Kontrolle zur Stelle des Aufrufs *(direkt hinter den Aufruf)*.

Übergabe von Parametern und der Kontrolle

Parameterübergabe

Register sind die schnellste Möglichkeit, Daten zu speichern / zu übergeben
⇒ sollten so häufig wie möglich verwendet werden
→ in MIPS: Reservierung einiger (der 32) Register zur Parameterübergabe

Beispiel: MIPS

MIPS reserviert sechs (von 32) Register für Parameter:
- $a0 ... $a3 : 4 Argumentregister zur Parameterübergabe
- $v0 ... $v1 : reserviert für (2) Rückgabewerte *(C benötigt nur 1)*

#	Name	#	Name	#	Name	#	Name	#	Name	#	Name	#	Name	#	Name
0	$zero	4	$a0	8	$t0	12	$t4	16	$s0	20	$s4	24	$t8	28	$gp
1	$at	5	$a1	9	$t1	13	$t5	17	$s1	21	$s5	25	$t9	29	$sp
2	$v0	6	$a2	10	$t2	14	$t6	18	$s2	22	$s6	26	$k0	30	$fp
3	$v1	7	$a3	11	$t3	15	$t7	19	$s3	23	$s7	27	$k1	31	$ra

Übergabe von Parametern und der Kontrolle

Kontrollübergabe

zur Rückkehr aus UP muss Kontrolle wissen, wohin gesprungen werden soll
→ Sprung zum Befehl hinter dem Aufruf
→ MIPS: Rücksprungadressregister $ra speichert (Adresse des Folgebefehls)

Beispiel: MIPS

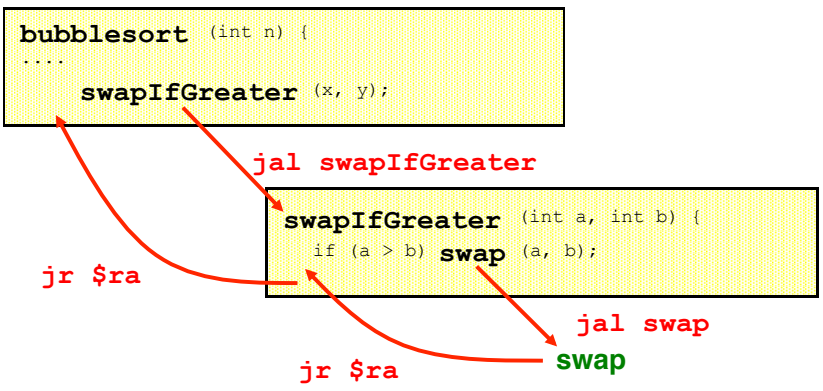
- $ra : *(return address register)* speichert Rücksprungadresse

#	Name	#	Name	#	Name	#	Name	#	Name	#	Name	#	Name	#	Name
0	$zero	4	$a0	8	$t0	12	$t4	16	$s0	20	$s4	24	$t8	28	$gp
1	$at	5	$a1	9	$t1	13	$t5	17	$s1	21	$s5	25	$t9	29	$sp
2	$v0	6	$a2	10	$t2	14	$t6	18	$s2	22	$s6	26	$k0	30	$fp
3	$v1	7	$a3	11	$t3	15	$t7	19	$s3	23	$s7	27	$k1	31	$ra

Übergabe von Parametern und der Kontrolle

zwei Sprungbefehle für UP-Aufruf und Rücksprung

- jal <Unterprogrammadresse> *(jump and link)*:
  speichert Adresse des Folgebefehls in $ra
  und springt an angegebene Unterprogrammadresse
- jr $ra *(jump register)*:
  jr springt an die in $ra angegebene Adresse



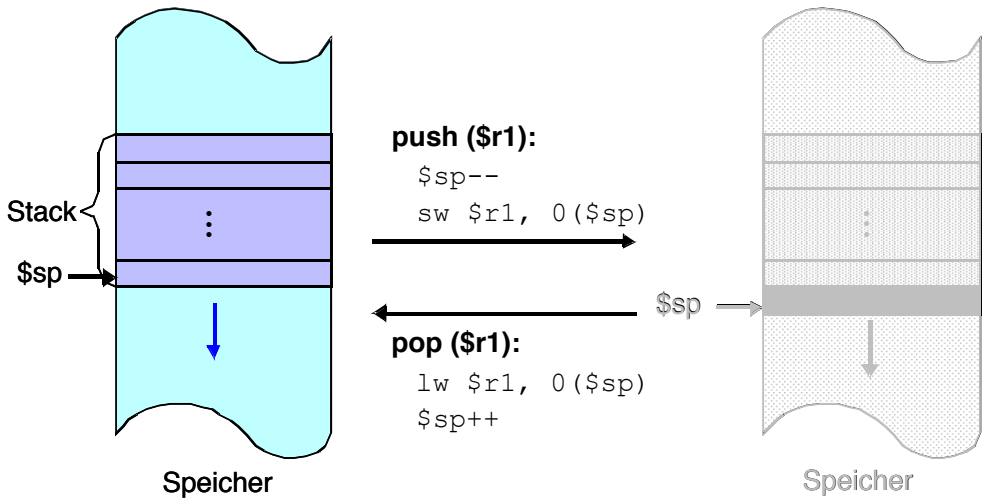
MIPS besitzt 32 Universalregister - davon einige (per Konvention) reserviert

#	Name	#	Name	#	Name	#	Name	#	Name	#	Name	#	Name	#	Name
0	$zero	4	$a0	8	$t0	12	$t4	16	$s0	20	$s4	24	$t8	28	$gp
1	$at	5	$a1	9	$t1	13	$t5	17	$s1	21	$s5	25	$t9	29	$sp
2	$v0	6	$a2	10	$t2	14	$t6	18	$s2	22	$s6	26	$k0	30	$fp
3	$v1	7	$a3	11	$t3	15	$t7	19	$s3	23	$s7	27	$k1	31	$ra

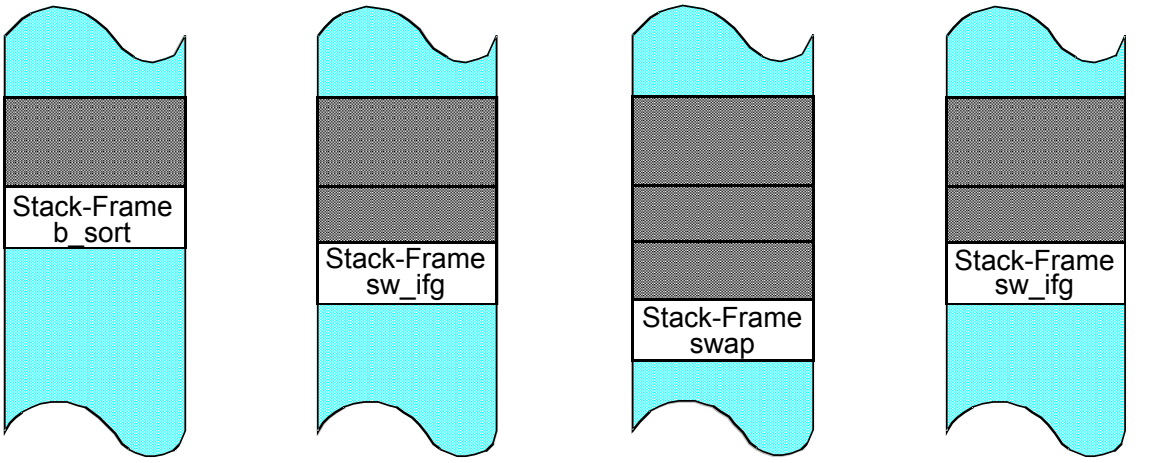
- Problem 1: Wohin mit den Daten, wenn mehr als
  - 4 Eingabeparameter
  - 2 Ausgabeparameter
  - 18 Variablen

  benötigt werden?
- Problem 2: Mehrfach geschachtelte UP (z.B. rekursive Funktionen)
  → Mehrfachverwendung der Register

Lösung: Stack



Lösung: Stack



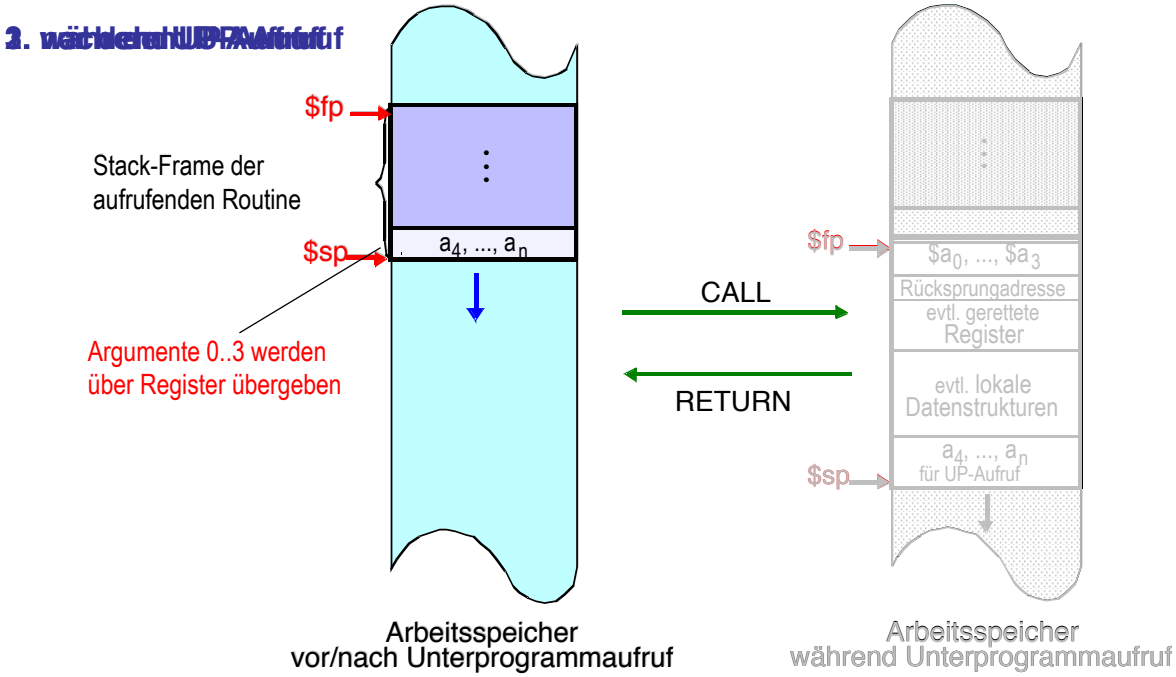
Stack bei Aufruf von *b_sort*.

Stack bei Aufruf von *sw_ifg*.

Stack bei Aufruf von *swap*.

Stack nach Rückkehr von *swap*.

Lösung: Stack



Arbeitsspeicher vor/nach Unterprogrammaufruf

Arbeitsspeicher während Unterprogrammaufruf

Geschachtelte Unterprogramme

→ wenn mehr als ein UP aufgerufen wird, müssen Registerinhalte sowie Parameter und Rücksprungadresse gerettet werden.

Typische Schritte *(etwas vereinfacht):*

- ***Caller* vor Unterprogrammaufruf:**

1. Alle noch benötigten Argumentregister $a0 … $a3 retten.
2. Argumente übergeben:
   - ersten 4 Parameter in Register *$a0-$a3*
   - Rest auf Stack. a_4 *bis* a_n stehen direkt vor dem Stack-Frame des Callee.
3. Sprung zum Unterprogramm: *jal <UP-addr>.*

Programmieren in C Exkurs: Prozessorarchitektur

Geschachtelte Unterprogramme

→ wenn mehr als ein UP aufgerufen wird, müssen Registerinhalte sowie Parameter und Rücksprungadresse gerettet werden.

Typische Schritte *(etwas vereinfacht):*

- ***Caller* vor Unterprogrammaufruf (vereinfacht):**

1. Argumente übergeben:
   - Aktuelle Parameter/Argumente auf Stack
   - a_0 *bis* a_n stehen direkt vor dem Stack-Frame des Callee.
2. Sprung zum Unterprogramm: *jal <UP-addr>.*

Geschachtelte Unterprogramme

→ wenn mehr als ein UP aufgerufen wird, müssen Registerinhalte sowie Parameter und Rücksprungadresse gerettet werden.

Typische Schritte *(etwas vereinfacht):*

- ***Callee* direkt nach Unterprogrammaufruf:**

1. Speicherplatz allokieren: *$sp = $sp - <Größe des Stack-Frame >*.
2. Notwendige Register retten.
   → *allgemeine Register, $ra* *(falls notwendig)*.

Programmieren in C Exkurs: Prozessorarchitektur

Geschachtelte Unterprogramme

→ wenn mehr als ein UP aufgerufen wird, müssen Registerinhalte sowie Parameter und Rücksprungadresse gerettet werden.

Typische Schritte *(etwas vereinfacht):*

- ***Callee* direkt vor Rücksprung:**

1. Ergebnis der Routine in *$v0* (und *$v1*) übergeben.
2. Gerettete Register *(Callee-Saved Registers)* „restaurieren“.
3. Stackpointer zurücksetzen.
4. Rücksprung zum caller: *jr $ra.*

Beispiel Bubble Sort

```
int b[] = {4, 2, 8, 22, 1};

int main (void) {
  int anz = sizeof(b) div 4;
  bubblesort (anz);
}

bubblesort(int n) {
// Liste b[0..n-1] durchlaufen
// evtl. Elemente austauschen
int i, j;
int * x, * y;
for (i=1; i<n; i++)
   for (j=0; j<n-i; j++) {
      x = &b[j]; y = &b[j+1];
      swapIfGreater(x, y);
      }
}
```

```
swapIfGreater(int* a, int* b) {
   if (*a > *b) swap(a, b);
}

swap(int* n, int* m) {
int tmp;
   tmp = *n;
   *n = *m;
   *m = tmp;
}
```

"call by reference":
→ Übergabe der Adressen/Pointer

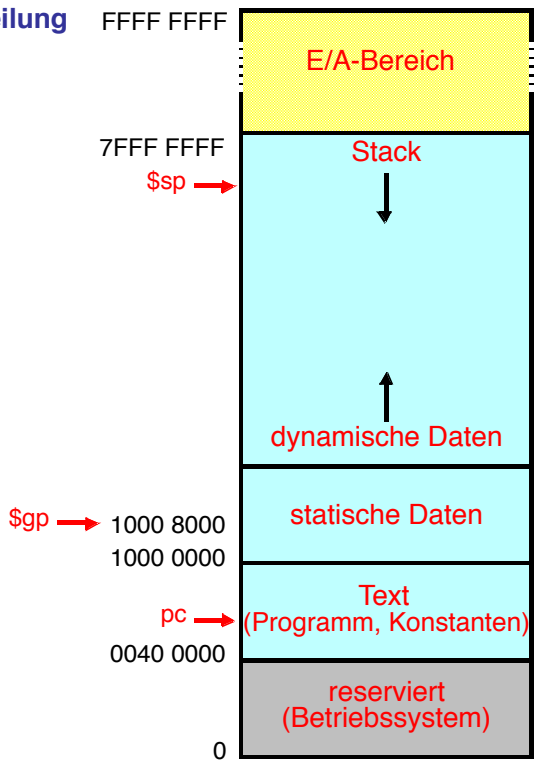
Programmieren in C Beispiel

Beispiel Bubble Sort

Globale Daten

- lokale Daten:
  - im Stack-Frame der aktuellen Routine
  - nur für aktuelle Routine und deren Unterprogramme sichtbar
  - werden bei Beendigung der Routine mit dem Stack-Frame gelöscht
  - werden nicht initialisiert
- globale Daten:
  - liegen in einem Speicherbereich außerhalb des Stack-Bereichs
  - bei der MIPS über *$gp* (*global pointer*) adressiert
  - sind von allen Routinen sichtbar
  - „überleben“ das Beenden der Routine, die sie angelegt hat
  - weitergehende Unterscheidung in
    - durch Variablendeklaration statisch angelegte Variablen (static data)
      → werden mit Null initialisiert
    - durch Allokationsfunktion dynamisch reservierter Speicher (in C: *malloc*) .
      → werden nicht initialisiert

MIPS-Speicheraufteilung



→ *$gp beliebig setzbar, um einfach auf globale Daten eines gewissen Bereichs zuzugreifen*

→ *mit 10008000_{16} initialisiert*
→ *16-Bit-Offset adressiert*
10000000_{16} ... $1000FFFF_{16}$

Programmieren in C
für Elektrotechniker

Kapitel 6: Funktionen

- Eigene Funktionen
- Unterprogramme
- **Gültigkeit von Namen**
- Rekursion und Iteration
- Modularer Quellcode

Gültigkeitsbereich von Namen

- Compiler übersetzt dateiweise. Namen in anderen Dateien sind unbekannt.
- In einer Datei gibt es vier Gültigkeitsbereiche (Scopes) von Namen:
  - Datei,
  - Funktion,
  - Block,
  - Funktionsprototyp.
- Innerhalb einer Datei gilt:
  - Externe Variablen sind ab ihrer Deklaration bekannt
    (ab der Definition einer externen Variablen bzw. einer extern-Deklaration).
    Gültigkeit: bis zum Ende der Datei.
  - Funktionen sind ab ihrer Deklaration bekannt
    (ab Funktionsdefinition, ab Funktionsprototyp oder ab extern-Deklaration).
    Gültigkeit: bis zum Ende der Datei.
  - In Blöcken eingeführte Namen, verlieren am Blockende ihre Bedeutung.
  - Formale Parameter gelten nur innerhalb der Funktion.
    Gültigkeit: Funktionsrumpf.
  - Formale Parameter in Prototypen sind nur dort bekannt *(→ unnötig)*.
  - Marken können in goto-Anweisungen in gesamter Funktion auftreten.
    Gültigkeit: Funktionsrumpf.

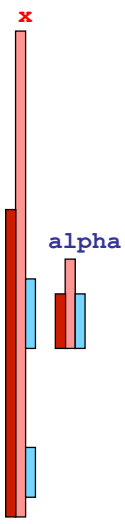
Gültigkeit, Sichtbarkeit, Lebensdauer

- Lebensdauer einer Variablen
  Zeitspanne, in der das Laufzeitsystem der Variablen einen Speicherplatz zur Verfügung stellt:
  - globale Variablen während der gesamten Programmlaufzeit,
  - lokale Variablen (in einem Block) während Ausführung des Blocks.
- Gültigkeit einer Variablen
  Der Name der Variablen ist an der betrachteten Stelle im Programm durch eine Vereinbarung bekannt.
- Sichtbarkeit einer Variablen
  Die Variable wird von der betrachteten Stelle im Programm gesehen, sodass auf sie zugegriffen werden kann.
- Gültigkeit und Sichtbarkeit
- Eine Variable kann gültig aber nicht sichtbar sein, wenn sie durch eine andere Variable mit gleichem Namen verdeckt wird.
  - Zugriff auf eine verdeckte Variable möglich?
    Auf verdeckte Variablen kann nur über die Adresse *(→ Pointer)*, nicht über den Namen zugegriffen werden.

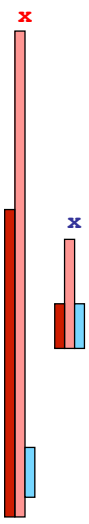
Gültigkeit, Sichtbarkeit, Lebensdauer

- Beispiel

```
#include <stdio.h>

int main (void)
{
}

int x;
void sub1 (void)
{
    int alpha = 4;
    ....
}

void sub2 (void)
{
    ...
}
```




```
#include <stdio.h>

int main (void)
{
}

int x;
void sub1 (void)
{
    int x;
    ....
}

void sub2 (void)
{
    ...
}
```



Gültigkeit, Sichtbarkeit und Lebensdauer von Variablen

▪ Gültigkeit, Sichtbarkeit, Lebensdauer

- Zusammenfassung

Variable	Sichtbarkeit	Gültigkeitsbereich	Lebensdauer
lokal	im Block, einschließlich der inneren Blöcke	im Block, einschließlich der inneren Blöcke	Block
global (extern)	im Programm ab Definition bzw. ab **`extern`**-Deklaration	im Programm ab Definition bzw. ab **`extern`**-Deklaration	Programm

▪ Namensraum

- Funktionen, formale Parameter und Variablen haben den selben Namensraum.
  → Lokale Variablen und formale Parameter verdecken globale (externe) Variablen und Funktionen mit gleichem Namen.

Beispiel:

```
double quadrat (double n)
{
    return n * n;
}
```

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

verdeckt Funktion ***quadrat***

C++: Fehlermeldung

▪ Initialisierung von Variablen

- Lokale Variablen (in einem Block) werden nicht initialisiert.
- Freigegebene Speicherbereiche sind weiterhin mit den alten Werten belegt.
  → Werden freigegebene Speicherbereiche durch neue Variablen eines anderen Blocks belegt, können diese mit den alten Werten *(d.h. sinnvollen Werten)* initialisiert sein.
  Damit darf man aber nicht rechnen.

Programmieren in C
für Elektrotechniker

Kapitel 6: Funktionen

- Eigene Funktionen
- Unterprogramme
- Gültigkeit von Namen
- **Rekursion und Iteration**
- Modularer Quellcode

Iterative und rekursive Funktionen

- Ein Algorithmus heißt
  - iterativ, falls Abschnitte des Algorithmus mehrfach durchlaufen werden.
  - rekursiv, falls Abschnitte des Algorithmus sich selbst wieder aufrufen (meist mit reduzierter Problemgröße).
- Theoretisch sind Iteration und Rekursion äquivalent.
  → Man kann sie ineinander umformen.
- Beispiel: Fakultät

$1! = 1$
$n! = (n-1)! \cdot n$

$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$

rekursiv:

```
int fak_rek (int n) {
   if (n <= 1)
      return 1;
   else
      return fak_rek(n-1) * n;
}
```

iterativ:

```
int fak_it (int n) {
   int i, ergebnis = 1;
   for (i = 1; i <= n; i++)
      ergebnis = ergebnis * i;
   return ergebnis;
}
```

```
int main (void) {
   int n, z;
   scanf("%d", &n);
   z = fak_rek(n);
   printf("%d %d", n, z);
}

int fak_rek (int n) {
   if (n <= 1)
      return 1;
   else
      return n * fak_rek(n-1);
}

int fak_rek (int n) {
   if (n <= 1)
      return 1;
   else
      return n * fak_rek(n-1);
}

int fak_rek (int n) {
   if (n <= 1)
      return 1;
   else
      return n * fak_rek(n-1);
}
```

fak_rek(3)

fak_rek(2)

fak_rek(1)

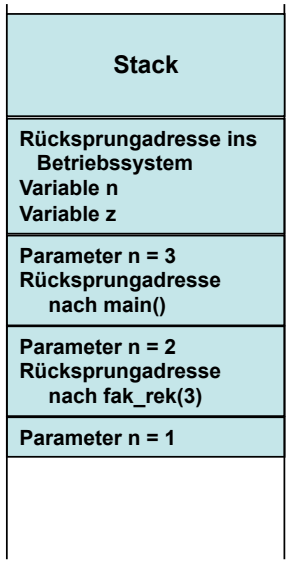
*3*2*1*

*2*1*

1

Nachteil der rekursiven Lösung:
mehr Speicherplatzbedarf

→ rekursive Aufrufe werden auf den Stack gelegt

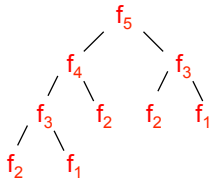


Fibonacci-Zahlen (iterative Lösung schneller durch Datenstruktur)

$f_0 = 0$
$f_1 = 1$
$f_n = f_{n-1} + f_{n-2}$ für $n \geq 2$

Wichtige Funktion zur Beschreibung von Naturphänomenen, Strategien des Mischens, ...

```
int fib_rek (int n) {
   if (n <= 0)
      return 0;
   else if (n == 1)
      return 1;
   else
      return fib_rek(n-1) + fib_rek(n-2);
}
```



Zeitbedarf wächst stark mit n, da viele Zw.-Ergebnisse mehrfach

```
int fib_it (int n) {
   int i, cur = 1, pre = 0;
   if (n > 0) {
      for (int i = 1; i < n; i++) {
         tmp = cur;
         cur = cur + pre;
         pre = tmp;
      }
      return cur;
   }
   else
      return 0;
}
```

Iterative und rekursive Funktionen

- In der Regel ist - Iteration schneller
  - Rekursion eleganter und leichter zu verstehen.
- Beispiel: dez2bin *(Umrechnung Dezimal- in Dualzahl - s. Übung)*

```
void d2b_rek (unsigned int zahl)
{
    if (zahl > 0) {
        d2b_rek (zahl / 2);
        printf ("%1d", zahl % 2);
    }
}


int main() {
    unsigned int zahl;
    printf ("\nDezimalzahl: ");
    scanf ("%d", &zahl);
    printf ("Binaerdarstellung:\n");
    d2b_rek (zahl);
    return 0;
}
```

```
void d2b_iter (unsigned int zahl)
{
    int array [sizeof(int)*8];
    int i;
    for (i=0; i<(sizeof(int)*8); i=i+1) {
        array[i] = 0;
    }
    for (i=0; zahl1!=0; i=i+1) {
        array[i] = zahl % 2;
        zahl = zahl / 2;
    }
    for (i=i-1; i>=0; i=i-1) {
        printf ("%1d", array[i]);
    }
}
```

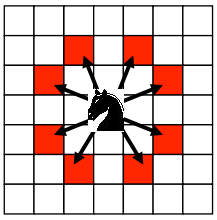

Iterative und rekursive Funktionen

- Backtracking: rekursiver Ansatz von Suchproblemen
  - Backtracking ist bei einer baumartigen Lösungsuche die Rückkehr aus einer Sackgasse zu einer vorhergehenden Stelle im Lösungsbaum, von der aus ein erneuter Lösungsversuch gestartet werden soll.
- Beispiel: Springerproblem
  - theoretisch löst Backtracking das Schachproblem, aber: Problemgröße macht Ansatz unbrauchbar.
  - Teilprobleme des Schachspiels werden damit gelöst
  - hier: Aufgabenstellung/Fragestellung:
    Kann der Springer von einer gegebenen Ausgangsposition alle Felder eines nxn-Spielfelds genau einmal besuchen?

Backtracking

Beispiel: Wege des Springers

Aufgabenstellung/Fragestellung:
Kann der Springer von einer gegebenen Ausgangsposition alle Felder eines nxn-Spielfelds genau einmal besuchen?



erlaubte Züge:

```
int springerX[] = {2, 1, -1, -2, -2, -1, 1, 2};
int springerY[] = {1, 2, 2, 1, -1, -2, -2, -1};
```

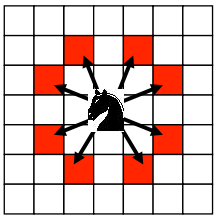
Lösungsidee (Algorithmus):

Alle legalen Züge ausführen und
von dort aus erneut versuchen,
bis eine Lösung gefunden wurde oder
alle möglichen Züge versucht wurden.

Aufwand: in der Größenordnung 8^n Züge,
d.h. exponentiell viele Züge

Backtracking

Beispiel: Wege des Springers



Beispiel:
5x5-Feld

1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

```
const int springerX [ ] = {2, 1, -1, -2, -2, -1, 1, 2};
const int springerY [ ] = {1, 2, 2, 1, -1, -2, -2, -1};

int schachbrett [n] [n];  /* Werte 0 bis n*n */

enum boolean versuchen (int x, int y, int nr) {
    int xNeu, yNeu, versuch;
    schachbrett[x][y] = nr;   /* Feld besetzen */
    if (nr == n*n)   /* alle Felder wurden erfolgreich besucht */
        return true;
    else
        for (versuch = 0; versuch < 8; versuch++) {
            /* 8 Durchläufe */
            xNeu = x + springerX[versuch];
            yNeu = y + springerY[versuch];
            if (0 <= xNeu && xNeu < n && 0 <= yNeu && yNeu < n &&
                        /* xNeu und yNeu sind auf dem Brett */
               schachbrett[xNeu][yNeu] == 0 &&   /* noch unbesucht */
               versuchen(xNeu, yNeu, nr+1))  /* Erfolg */
            return true;
        };
    schachbrett[x][y] = 0;    /* Feld zurücksetzen */
    return false;             /* alle 8 Versuche blieben erfolglos */
}
```

Programmieren in C
für Elektrotechniker

Kapitel 6: Funktionen

- Eigene Funktionen
- Unterprogramme
- Gültigkeit von Namen
- Rekursion und Iteration
- **Modularer Quellcode**

- **C unterstützt das getrennte Übersetzen von „Modulen“**
  - Module in C: Separat übersetzbare Einheiten *(→ Dateien)*: Übersetzungseinheiten
    → können jeweils eine oder mehrere Funktionen enthalten.
  - Verbindung von Modulen über globale/externe Variablen und Funktionen.
  - Bezüge zwischen Modulen werden durch den Binder aufgelöst
    → Ergebnis: Einheitlicher Adressraum für gesamtes Programm.

Externe (globale) Variablen

- Externe Variablen werden
  - in einer Datei definiert *(→ legt Speicherbereich und Adresse fest)* und
  - in allen anderen Dateien vor ihrer Verwendung über **`extern`**-Deklaration bekannt gemacht.
- Beispiel

Datei ext1.c

```
int main (void) {
    f1 ();
    ...
}

int n = 6;

void f1 (void) {
    ...
}
```

Binder

Datei ext2.c

```
extern int n;

void f2 (void) {
    n = 8;
}
```

- Externe Variablen können nur bei ihrer Definition manuell initialisiert werden.

Funktionsdeklarationen

- Funktionen müssen vor ihrer Verwendung bekannt gemacht werden.
- Beispiel

Deklaration von n, da main() vor Definition von n.

Datei ext1.c

```
void f1 (void);
extern void f2 (void);
extern int n;

int main (void) {
    n = n + 1;
    f1 ();
    f2 ();
}

int n = 6;

void f1 (void) {
    ...
}
```

Datei ext2.c

```
extern int n;

void f2 (void) {
    n = n + 2;
}
```

Funktionsdeklarationen

- Funktionen müssen vor ihrer Verwendung bekannt gemacht werden.
- Beispiel

n nicht in f1() sichtbar.

Datei ext1.c

```
void f1 (void);
void f2 (void);

int main (void) {
    extern int n;
    n = n + 1;
    f1 ();
    f2 ();
}

void f1 (void) {
     ...
}

int n = 6;
```

Datei ext2.c

```
extern int n;

void f2 (void) {
     n = 8;
}
```

- **`extern`** bei Funktionen überflüssig, da Funktionsprototypen immer implizit als **`extern`** angenommen werden.

Funktionsdeklarationen

- **`extern`**-Deklarationen meist außerhalb aller Funktionen, typ. am Datei-Anfang
  → meist in einer header-Datei, die am Dateianfang inkludiert wird.
- Beispiel

Datei ext1.c

```
#include "ext.h"

int main (void) {
    n = n + 1;
    f1 ();
    f2 ();
}

void f1 (void) {
     ...
}

int n = 6;
```

Datei ext.h

```
void f1 (void);
void f2 (void);
extern int n;
```

Datei ext2.c

```
#include "ext.h"

void f2 (void) {
     n = 8;
}
```

Schutz vor externem Zugriff

- Sollen Variablen und Funktionen extern nicht verwendet werden dürfen, werden diese mit **`static`** geschützt.
- Beispiel:

Datei ext1.c

```
#include "ext.h"

int main (void) {
    n = n + 1;
    f1 ();
    f2 ();
}

void f1 (void) {
     ...
}

static int n = 6;
```

Datei ext.h

```
void f1 (void);
void f2 (void);
extern int n;
```

Datei ext2.c

```
#include "ext.h"

static void f2 (void)
{
     n = 8;
}
```

- Fehlerfreie Übersetzung beider Module.
- Fehlermeldung durch Binder: - in **`ext1.c`** kann **`f2`** nicht aufgelöst werden,
  - in **`ext2.c`** kann **`n`** nicht aufgelöst werden.

Verwendung von Registern

- Bei lokalen Variablen und formalen Parametern kann mit dem Schlüsselwort **`register`** dem Compiler empfohlen werden, diese in Register *(statt im Speicher)* abzulegen.
  - Nur für bestimmte Datentypen möglich *(char, short, int und Pointer)* .
  - Nur zur Laufzeitoptimierung – bei häufig verwendeter Variablen.
  - Compiler muss sich nicht daran halten.
  - Adressoperator ‚&' auf Register-Variablen nicht möglich.
  - Meist überflüssig, da dies i.d.R. die Compileroptimierung automatisch macht.

Dauerhafte lokale Variablen

- Lokale Variablen erhalten mit **`static`** ein Gedächtnis.
- **`static`** Variablen werden nicht auf dem Stack abgelegt.
- Sie sind nur im aktuellen Block sichtbar, aber während der gesamten Programmausführung im globalen Speicherbereich angelegt.
- Manuelle Initialisierung nur beim ersten Aufruf.
- Beispiel:

`a` protokolliert die Anzahl an Aufrufen von „`beispiel`“.

```
void beispiel (void)
{
    static int a = 1;
    printf ("\na = %d", a);
    a = a + 1;
}

int main (void)
{
    int i;
    for (i = 0; i <= 2; i++)
       beispiel ();
    return 0;
}
```


Zusammenfassung

Variablen-klasse	Gültigkeit	Lebens-dauer	autom. Initialisierung	Speicher-Segment
`register`	Block	Block	nein	Register (oder Stack)
Lokale V. (**`auto`**)	Block	Block	nein	Stack
`static` lokale V.	Block	Programm	mit 0	Statischer Datenbereich
`static` externe V.	In Datei ab Definition, in anderen Dateien nicht	Programm	mit 0	Statischer Datenbereich
`extern` (nicht **`static`**)	Im Programm ab Definition bzw. ab extern-Deklaration	Programm	mit 0	Statischer Datenbereich

Zusammenfassung

- Folgende Tabelle zeigt, welche Speicherklassen eingesetzt werden können, wenn man einen bestimmten Gültigkeitsbereich für Variablen erzeugen möchte.

Gültigkeitsbereich	Variablenklasse
Nur in Funktion oder Block.	• lokale Variablen (`auto`), • `auto static` (lokal, permanent), • `Register.`
Externe Variablen nur in aktueller Datei.	`static` bei Definition.
Externe Variablen im gesamten Programm.	Nicht `static` bei Definition und `extern` in anderen Dateien.

Styleguide

- Jede Funktion, die außerhalb der Datei benutzt werden soll, in der sie definiert ist, erhält einen Prototypen in einer Header-Datei.
- Jede Datei, die einen Aufruf einer Funktion enthält, „inkludiert“ die entsprechende Header-Datei.
- Die Quelldatei, die die Definition einer Funktion enthält, soll die Header-Datei ebenfalls „inkludieren“.

→ *später mehr*