Einführung in die Programmierung

Bachelor of Science

Prof. Dr. Rethmann

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

WS 2009/10

Übersicht

- *die main-Funktion*
- Speicherverwaltung
- Rekursive Funktionen
- Gültigkeitsbereiche

main

Definition nach ANSI C:

```
int main(int argc, char *argv[]) {
    ....
    return 0;
}
```

Zeichenfolgen als Parameter übergeben:

- `argc` (argument count): Anzahl der beim Aufruf angegebenen Zeichenfolgen inkl. Programmname
- `argv` (argument vector): Vektor von Zeigern auf die angegebenen Zeichenfolgen
- `argv[0]` = Programmname

Rückgabewert 0: erfolgreich ausgeführtes Programm

main

```
#include <stdio.h>

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

   for (i = 0; i < argc; i++)
      printf("Parameter %d = %s\n", i, argv[i]);
}
```

Wird das Programm als `arg.c` gespeichert und mit `arg eins 2 drei` ausgeführt, so wird folgende Ausgabe erzeugt:

```
Parameter 0 = arg
Parameter 1 = eins
Parameter 2 = 2
Parameter 3 = drei
```


main

üblich: String-Array zur Abfrage von Umgebungsvariablen.

```
#include <stdio.h>
void main(int argc, char *argv[], char *env[]) {
    int i;

    for (i = 0; env[i] != NULL; i++)
        printf("Variable %d = %s\n", i, env[i]);
}
```

Erzeugt beim Aufruf unter Linux bspw. die Ausgabe:

```
Variable 0 = TERM=xterm
Variable 1 = SHELL=/bin/bash
Variable 2 = PAGER=/usr/bin/less
Variable 3 = LANG=de_DE.UTF-8
...
```


main

Umwandeln der Programmparameter: `sscanf(char *s, ...)`

- äquivalent zu `scanf`, aber Eingabezeichen stammen aus der Zeichenkette `s`.
- Rückgabewert: Anzahl der umgewandelten Eingaben.

Beispiel:

```
int i, n, p;

for (i = 0; i < argc; errno = 0, i++) {
   p = sscanf(argv[i], "%d", &n);
   if (p == 0)
      printf("failed to convert %s\n", argv[i]);
   else if (errno == ERANGE)
      printf("%s is out of range\n", argv[i]);
   else printf("%d: value %d\n", i, n);
}
```


Übersicht

- die main-Funktion
- *Speicherverwaltung*
- Rekursive Funktionen
- Gültigkeitsbereiche

Speicherverwaltung

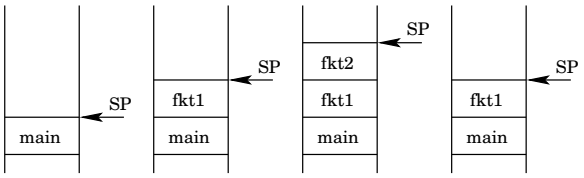
Der von einem C-Programm während seiner Ausführung belegte Speicher ist in vier Bereiche aufgeteilt:

- *Code-Segment*: enthält das übersetzte Programm, also die auszuführenden Maschinenbefehle.
- *Statische Daten*: während des gesamten Lebenszyklus verfügbare Daten wie globale Variablen oder statische, lokale Variablen.
- *Heap:* dynamisch allozierter Speicher (`malloc`)
- *Stack:* Informationen über Funktionsaufrufe

Speicherverwaltung

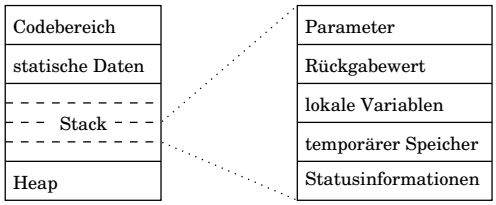
Für die lokalen Variablen und Argumente einer Funktion wird erst beim Aufruf der Funktion Speicherplatz reserviert.

Nach Beenden der Funktion wird der Speicherplatz wieder freigegeben.



Ein CPU-Register (SP: Stack Pointer) enthält die Adresse des nächsten freien Speicherplatzes.

Speicherverwaltung



Alle Informationen, die zum Ausführen einer Funktion notwendig sind, werden in einem *Stack-Frame* abgelegt:

- *temporärer Speicher:* Evaluierung von Ausdrücken
- *Statusinformationen:* Programmzähler, ...

Speicherverwaltung

Vorsicht: Zeiger auf lokale Variablen als Rückgabewert einer Funktion liefern keinen definierten Wert!

```
char* intToString(int a) {
    char s[10];
    int i, t;

    for (t = 1; t < a; t *= 10)
        ;

    for (t /= 10, i = 0; t > 0; t/= 10, i++)
        s[i] = (a / t) % 10 + '0';
    s[i] = '\0';

    return s;
}
```


Speicherverwaltung

```
int * extremwerte(int *a, int n) {
    int i, res[2];
    int min = a[0];
    int max = a[0];

    for (i = 1; i < n; i++) {
        if (a[i] < min)
            min = a[i];
        if (a[i] > max)
            max = a[i];
    }
    res[0] = max;
    res[1] = min;

    return res;
}
```


Übersicht

- die main-Funktion
- Speicherverwaltung
- *Rekursive Funktionen*
- Gültigkeitsbereiche

Rekursive Funktionen

Rekursion in der Mathematik:

- *Binomialkoeffizienten:*

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \text{ mit } \binom{n}{n} = 1 \text{ und } \binom{n}{0} = 1$$

- *Fibonacci-Zahlen:*

$$F_n = F_{n-1} + F_{n-2} \text{ mit } F_0 = 0 \text{ und } F_1 = 1$$

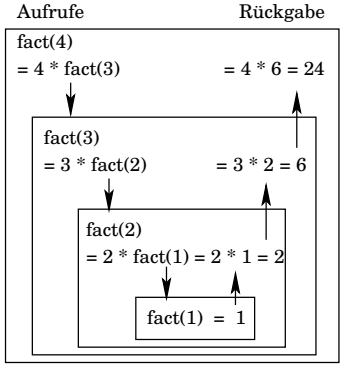
- *Determinante:* Die Adjunkte A_{ik} ist die Determinante $(n-1)$ter Ordnung, die durch Streichen der i-ten Zeile und k-ten Spalte, multipliziert mit $(-1)^{i+k}$ entsteht. Entwickeln nach der i-ten Zeile:

$$D(a_{ij}) = \sum_{k=1}^{n} a_{ik} A_{ik}$$

Fakultät – C-Code

```
#include <stdio.h>

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

void main() {
   int n;

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

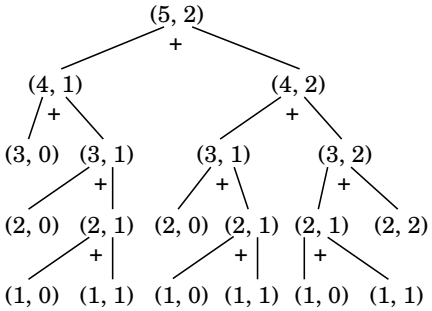

Fakultät – Programmablauf



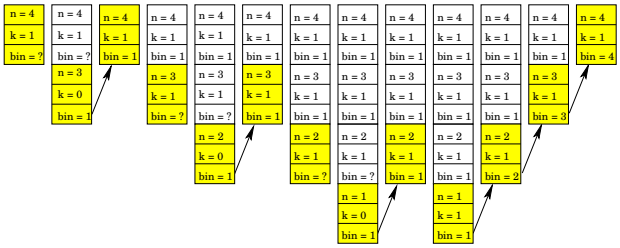
Binomialkoeffizienten – C-Code

```c
#include <stdio.h>

int bin(int n, int k) {
   if (n < k || k < 0)
      return -1;

   if (k == 0 || k == n)
      return 1;

   return bin(n-1, k-1) + bin(n-1, k);
}

void main() {
   printf("bin(%d,%d) = %d\n", 7, 3, bin(7, 3));
}
```


Binomialkoeffizienten – Programmablauf



Binomialkoeffizienten – Speicherverwaltung



Quersumme

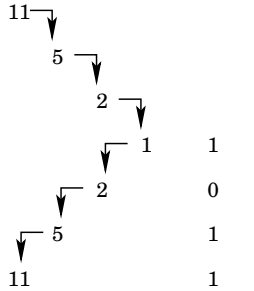
```c
#include <stdio.h>

int qsum(unsigned int zahl) {
   int ziffer;

   if (zahl < 10)
      return zahl;

   ziffer = zahl % 10;
   return ziffer + qsum(zahl / 10);
}

void main(void) {
   unsigned int i;

   for (i = 731; i < 750; i++)
      printf("Quersumme(%d)= %d\n", i, qsum(i));
}
```


Umwandeln in Binärdarstellung

C-Code:

```c
#include <stdio.h>

void intToBin(int n) {
    if (n < 2) {
        printf("%1d", n);
        return;
    }

    intToBin(n / 2);
    printf("%1d", n % 2);
}

void main(void) {
    intToBin(11);
}
```

Ablauf:



Rekursion und Mathematik

Rekursion kennen wir auch an anderer Stelle in der Mathematik: Beweis durch vollständige Induktion

Beispiel:

```c
#include <stdio.h>

int sum(unsigned int n) {
    if (n == 1)
        return n;
    return n + sum(n-1);
}

void main(void) {
    printf("Summe 1+...+10 = %d\n", sum(10));
}
```


Rekursion und Mathematik

Beispiel:

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

Induktionsanfang für $n = 1$: $\longrightarrow$ Rekursionsende

$$\sum_{i=1}^{1} i = 1 \stackrel{!}{=} \frac{1 \cdot 2}{2} = 1$$

Induktionsschluss $n \rightsquigarrow n+1$: $\longrightarrow$ rekursiver Aufruf

$$\sum_{i=1}^{n+1} i = \sum_{i=1}^{n} i + (n+1) \stackrel{I.A.}{=} \frac{n \cdot (n+1)}{2} + (n+1)$$
$$= \frac{(n+1) \cdot (n+2)}{2}$$

Übersicht

- die main-Funktion
- Speicherverwaltung
- Rekursive Funktionen
- *Gültigkeitsbereiche*

Gültigkeitsbereich von Bezeichnern

Gültigkeitsbereich (scope): Der Teil des Programms, wo ein Bezeichner benutzt werden kann.

Die Bedeutung einer Deklaration ist abhängig von deren Stelle im C-Code. Mögliche Stellen für Deklarationen:

- außerhalb von Funktionen: *globale Variablen*
- im Funktionskopf: *formale Parameter einer Funktion*
- innerhalb eines Blocks: *lokale Variablen*

Funktionen können nur außerhalb von Funktionen deklariert werden.

Gültigkeitsbereich von Bezeichnern

Module: C-Quellcode auf mehrere Dateien aufteilen:

- eine Datei enthält Vereinbarungen und/oder Funktionen
- nur eine Datei darf die Funktion `main` enthalten

Beispiel: Liste

- `liste.h`: enthält die Datenstruktur und die Deklaration aller Zugriffsmethoden
- `liste.c`: Implementierung aller Zugriffsmethoden
- `haupt.c`: Programm verwendet die Liste, enthält `main`

⇒ gleichzeitiges Arbeiten am Source-Code möglich

Gültigkeitsbereich von Bezeichnern

`liste.h`

```c
typedef struct node {
   struct node *pred, *succ;
   int value;
} node_t;

typedef struct list {
   node_t *first, *last;
   int size;
} list_t;

list_t * create(void);
void enqueue(list_t *l, int val);
int dequeue(list_t *l);
char isEmpty(list_t *l);
```


Gültigkeitsbereich von Bezeichnern

`liste.c`

```c
#include <stdlib.h>
#include "liste.h"

list_t * create(void) {
   list_t *l;

   l = (list_t *) malloc(sizeof(list_t));
   l->first = (node_t *) malloc(sizeof(node_t));
   l->last = (node_t *) malloc(sizeof(node_t));

   l->first->succ = l->last;
   l->first->pred = NULL;
   l->last->succ = NULL;
   l->last->pred = l->first;
   l->size = 0;

   return l;
}
```


Gültigkeitsbereich von Bezeichnern

```c
void enqueue(list_t *l, int val) {
   node_t *neu;
   node_t *p = l->last->pred;

   neu = (node_t *) malloc(sizeof(node_t));
   neu->value = val;
   neu->succ = l->last;
   neu->pred = p;
   p->succ = neu;
   l->last->pred = neu;
   l->size += 1;
}

char isEmpty(list_t *l) {
   return l->size == 0;
}
```


Gültigkeitsbereich von Bezeichnern

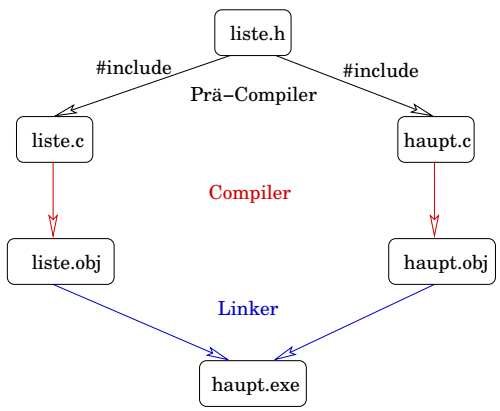
```c
int dequeue(list_t *l) {
   int res = -1;
   node_t *f;

   if (isEmpty(l))
      return res;

   f = l->first->succ;
   l->first->succ = f->succ;
   f->succ->pred = l->first;
   res = f->value;
   l->size -= 1;

   free(f);
   return res;
}
```


Gültigkeitsbereich von Bezeichnern

`haupt.c`

```c
#include <stdio.h>
#include "liste.h"

int main(void) {
   int i;
   list_t *list;

   list = create();
   for (i = 3; i < 20; i++)
      enqueue(list, i);

   while (!isEmpty(list))
      printf("%d\n", dequeue(list));

   return 0;
}
```


Gültigkeitsbereich von Bezeichnern



Gültigkeitsbereich von Bezeichnern

Es gibt vier mögliche Gültigkeitsbereiche:

- der Compiler kennt nur, was er bereits gelesen hat, aber keine anderen Dateien → globale Variablen und alle Funktionen gelten vom Deklarationspunkt bis zum Ende der Datei
- formale Funktionsparameter gelten vom Deklarationspunkt bis zum Ende der Funktion bzw. des Prototypen
- lokale Variablen am Beginn eines Blocks gelten bis zum Ende des Blocks
- Anweisungsmarken gelten innerhalb der Funktion, in der die Deklaration erfolgt

Gültigkeitsbereich von Bezeichnern

```
int global;      // gilt bis zum Ende der Datei

// x: gilt nur innerhalb der Klammern
double sin(double x);

int main(int argc, char *argv[]) {
   int lokal;    // gilt nur innerhalb von main
   ...
   for (i = 0; i < 10; i++) {
      int j;     // gilt nur innerhalb des Blocks
      for (j = 0; j < i; j++) {
         ...
      }
   }

   // Label ende gilt nur innerhalb von main
   ende: ...
}
```


Lebensdauer von Objekten

Wie lange bleibt ein Objekt im Speicher und kann über seinen Bezeichner angesprochen werden?

- *statische Lebensdauer:* Objekte sind stets verfügbar
  - Funktionen
  - globale Variablen
  - statische Variablen
- *automatische Lebensdauer:* Objekte, die zu Beginn eines Blocks angelegt werden, sind nach dem Verlassen des Blocks nicht mehr verfügbar, sofern nicht anders definiert.

  eine Funktion ist auch ein Block
- *dynamische Lebensdauer:* Der Speicherbereich wird mittels Bibliotheksfunktionen angelegt und freigegeben (`malloc` und `free`).

Lebensdauer von Objekten

```
void fkt(int x) {
   static int s = 1; // statische Lebensdauer
   int a = 1;        // automatische Lebensdauer

   printf("x= %d, s= %d, a= %d\n", x, s++, a++);
}

int main(void) {
   int i, *p;

   p = (int *) malloc(10 * sizeof(int));
   for (i = 0; i < 10; i++) {
      p[i] = i + 1; // dynamische Lebensdauer
      fkt(p[i]);
   }
   free(p);
   return 0;
}
```


Speicherklasse

Die Speicherklasse eines Objekts hat Einfluss auf

- die Lebensdauer und
- den Gültigkeitsbereich des Objekts und
- erlaubt Einschränkungen für den Zugriff auf Daten.

Speicherklasse: `extern`

Variablen, die außerhalb von Funktionen deklariert werden, sind implizit vom Typ `extern`.

Lebensdauer: Für die gesamte Laufzeit des Programms wird Speicherplatz reserviert.

Gültigkeit: Globale Variablen gelten von dem Punkt, an dem sie vereinbart werden, bis zum Ende der Datei.

Ein Objekt einer externen Deklaration ist beim Binden auch Programmteilen aus anderen Dateien bekannt.

Speicherklasse: `extern`

In der Regel ist Datenaustausch über eine Parameterliste gegenüber globalen Variablen vorzuziehen:

- Global gültige Daten führen oft zu Programmen mit vielen, schwer durchschaubaren Datenpfaden zwischen Funktionen (mit unerwünschten Nebenwirkungen).
- Der Aufbau von Bibliotheken mit allgemein gültigen Funktionen ist nicht möglich.

Weniger Fehleranfällig: Prinzip der Datenkapselung

- Daten sind nur innerhalb eines Moduls sichtbar
- Zugriff auf Daten nur über definierte Zugriffsmethoden
- mehr dazu später

Speicherklasse: `static`

Anwendung bei

- *Funktionen:* Die Funktion ist nur innerhalb der Quelldatei und *nicht* dem Linker bekannt (siehe Modulare Programmierung, Datenkapselung).
- *lokale Variablen:* Objekte behalten ihren Wert auch nach Verlassen des Blocks, in dem sie definiert sind.
- *globale Variablen:* Einschränkung des Gültigkeitsbereichs auf die Datei.

Gültigkeit: innerhalb des umschließenden Blocks bzw. bei Funktionen und globalen Variablen innerhalb der Datei.

dat1.c:

```
void f(void) {
   extern int i;
   ...
}

static int i;
int j;
void g(void) {
   i = 42;
   j = 4711;
   ...
}
```

dat2.c:

```
void h(void) {
   extern int j; // ok
   extern int i; // Fehler!

   i += 42;
   j += 4711
}
```

sehr schlechter Programmierstil!