Einführung in die Programmierung

Bachelor of Science

Prof. Dr. Rethmann

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

WS 2009/10

Übersicht

- Arrays
- Datentypen, Operatoren und Kontrollstrukturen
- *Funktionen und Zeiger*
- Zeichenketten
- Dateioperationen und Standardbibliotheken
- strukturierte Programmierung
- modulare Programmierung

Funktionen

- Funktionen erledigen eine abgeschlossene Teilaufgabe.
- Funktionen zerlegen Programme in kleinere Einheiten und erhöhen so die Übersichtlichkeit und die Wartbarkeit von Programmen! → *strukturierte Programmierung*
- Oft gebrauchte Funktionen und Daten sind in Bibliotheken (Libraries) bereitgestellt: `stdio`, `stdlib`, `string`, `math`, ...
- Bei gut entworfenen Funktionen reicht es zu wissen *was* getan wird, gleichgültig *wie* eine Aufgabe gelöst wird.

Funktionen

Syntax:

```
Typ Funktionsname( Parameterliste ) {
    Vereinbarungen
    Anweisungen
}
```

Beispiel:

```
int max(int a, int b) {
    if (a > b)
        return a;
    return b;
}
```

Eine Funktion kann mit `return` einen Wert zurückgeben.

Funktionen

```
#include <stdio.h>
long int fakult(int n) {
    int i;
    long int res = 1;

    for (i = 2; i <= n; i++)
        res *= i;
    return res;
}

void main(void) {
    int wert;

    printf("Wert? ");
    scanf("%d", &wert);

    printf("%d! = %ld\n", wert, fakult(wert));
}
```


Funktionen

```
#include <stdio.h>

int summe(int n) {
    int i, sum = 0;

    for (i = 1; i < n; i++)
        sum += i;
    return sum;
}

void main(void) {
    int wert = 42;
    int erg;

    erg = summe(wert);
    printf("sum(1+2+...+%d) = %d\n", wert, erg);
}
```


Funktionen

```
#include <stdio.h>

int binom(int n, int k) {
    if (n == k || k == 0)
        return 1;

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

void main(void) {
    int a = 7;
    int b = 3;
    int erg;

    erg = binom(a, b);
    printf("binKoeff(%d,%d) = %d\n", a, b, erg);
}
```


Funktionen

Gültigkeitsbereich von Bezeichnern:

- Parameternamen und lokal deklarierte Variablen sind nur innerhalb der Funktion bekannt und für andere Funktionen nicht sichtbar!

→ *andere Funktionen können dieselben Namen ohne Konflikte nutzen!*

Rückgabewerte:

- Eine Funktion muss keinen Rückgabewert liefern: Typ `void`
- An der aufrufenden Stelle darf der Rückgabewert einer Funktion ignoriert werden.

Funktionen

Funktionen müssen (wie Variablen) deklariert sein, bevor sie benutzt werden können.

Funktionsdeklaration:

```
int square(int x);
```

Funktionsdefinition:

```
int square(int x) {
    return x*x;
}
```

Funktionen der Standard-Bibliothek werden in *Definitionsdateien* (header file) wie `stdio.h` und `math.h` deklariert.

Definitionsdateien werden mittels `#include`-Anweisung am Anfang einer Quelldatei bereitgestellt.

Anmerkungen

Rufen sich zwei Funktionen gegenseitig auf, müssen zunächst Funktionsprototypen definiert werden:

```
int a(int y);     /* Prototyp */
int b(int z);     /* Prototyp */

int a(int x) {
    return b(x - 1) * b(x - 2);
}

int b(int x) {
    return (x <= 0) ? x * 2 : a(x - 10);
}
```

Parameternamen im Prototyp und im Funktionskopf müssen nicht übereinstimmen.

Anmerkungen

Compiler-Fehler, wenn Prototyp und Funktionskopf unterschiedliche Rückgabewerte oder Parameter haben.

Beispiel:

```
int sqr(int x);
.....
float sqr(float x) {
    return x*x;
}
```

Beispiel:

```
int fkt(int x, int y);
.....
int fkt(int x) {
    return x*x;
}
```

Aufruf einer Funktion, die vorher nicht deklariert wurde:

- Als Rückgabetyp wird `int` angenommen.
- Es werden keine Annahmen über Parameter getroffen.
- Compiler-Verhalten abhängig von der Implementierung.

Zeiger und Adressen

Ein Zeiger ist nichts anderes als eine Adresse im Hauptspeicher, an der ein Objekt steht.

Wofür braucht man das?

Zeiger und Adressen

Wofür braucht man das?

- Der Wert eines Funktionsparameters wird als Kopie an die Funktion übergeben.
- Damit ist sichergestellt, dass die Funktion den Wert im aufrufenden Programmteil nicht zerstören kann.
- Aber wie kann dann eine Funktion einen Wert im aufrufenden Programmteil ändern? (bspw. `scanf("%d", &n)`)

Zeiger und Adressen

Wofür braucht man das?

- Wollen wir ein Array an eine Funktion übergeben, z.B. um die Werte zu sortieren, müsste das gesamte Array kopiert werden.
- Damit das nicht passiert (kostet viel Zeit), wird der Funktion nur die Adresse übergeben, an der sich das Array befindet.

Zeiger und Adressen

Wofür braucht man das?

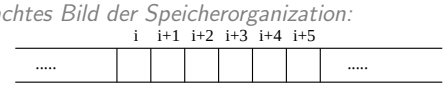
- Oft ist zu dem Zeitpunkt, an dem ein Programm geschrieben wird, noch nicht klar, wieviele Daten zu verwalten sind: sollen 100 Bücher oder 1.000.000 Bücher gespeichert werden?
- Da wir nicht auf Verdacht Platz für 1.000.000 Bücher reservieren wollen, kann während des Programmablaufs vom Betriebssystem zusätzlicher Speicherplatz bereitgestellt werden.
- Um diesen bereitgestellten Speicherplatz im Programm auch nutzen zu können, teilt uns das Betriebssystem die Adresse der ersten reservierten Speicherstelle mit.

Zeiger und Adressen

Für jeden Typ T kann man einen *Zeiger auf T* erzeugen. Der Wert eines Zeigertyps ist die Adresse eines Objekts.

Spezieller Wert: der leere Zeiger (`NULL`-Zeiger) entspricht der Konstante 0. Programme sind besser lesbar, wenn die logische Konstante `NULL` verwendet wird.

Vereinfachtes Bild der Speicherorganization:



Speicherzellen sind fortlaufend nummeriert und adressierbar, sie können einzeln oder in zusammenhängenden Gruppen bearbeitet werden.

Zeiger und Adressen

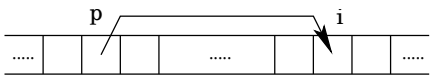
Der *Adressoperator &* liefert die Adresse eines Objekts.

Der *Inhaltsoperator ∗* liefert das Objekt, das unter einer Adresse abgelegt ist.

Beispiel:

```
int i, j;   /* Integer-Wert */
int *p;     /* Zeiger auf Integer-Wert */

p = &i;     /* p := Adresse von i */
            /* also: p zeigt auf i */
*p = 5;     /* Inhalt von p := 5 */
j = *p;     /* j := Inhalt der Adresse p */
```



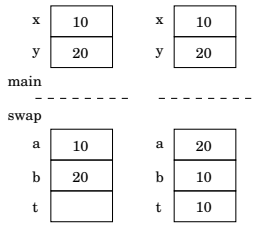
Parameterübergabe

Alle Parameter werden als Wert übergeben (*call by value*).

Innerhalb der Funktionen werden private Kopien der übergebenen Parameter angelegt, die während der Ausführung benutzt werden.

Beispiel:

```
void swap(int a, int b) {
    int t = a;
    a = b;
    b = t;
}
.....
swap(x, y);
```



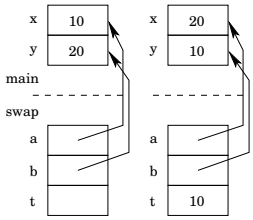
Parameterübergabe

`swap` kann die Werte beim Aufrufer nicht beeinflussen, da nur Werte (Kopien) übergeben werden.

Lösung: Der Aufrufer muss Zeiger auf die Werte, die geändert werden sollen, übergeben (*call by reference*).

Beispiel:

```
void swap(int *a, int *b)
{
    int t = *a;
    *a = *b;
    *b = t;
}
.....
swap(&x, &y);
```



Zeiger und Adressen

Die Syntax der Variablenvereinbarung `int *p` imitiert die Syntax von Ausdrücken, in denen die Variable auftreten kann:

Der Ausdruck `*p` ist ein `int`-Wert.

Daraus folgt:

- Ein Zeiger darf nur auf eine Art von Objekt zeigen.
- Jeder Zeiger zeigt auf einen festgelegten Datentyp.

Ausnahme: Ein *Zeiger auf* `void`

- nimmt einen Zeiger beliebigen Typs auf,
- darf aber nicht selbst zum Zugriff verwendet werden.

Zeiger und Adressen

```
void *v;
int i = 1;
double d = 2.0;

v = &i;                 /* v zeigt auf i */
*(int *) v += 1;        /* cast notwendig! */
                        /* also: i += 1 */
printf("%d, %d\n", i, *(int *) v);


v = &d;                 /* v zeigt auf d */
*(double *) v += 1.0;   /* cast notwendig! */
                        /* also: d += 1 */
printf("%f, %f\n", d, *(double *) v);
```