Algorithmische Mathematik I

Wintersemester 2009/2010
Prof. Dr. Mario Bebendorf
Dr. Jan Hamaekers

universitätbonn

Programmierhinweise: `structs`, Funktionen und Zeiger

Zusammengesetzte Datenstrukturen mit `struct`

In `C` kann man neben den eingebauten Datentypen wie `unsigned int`, `double` etc. auch eigene Strukturen definieren. Das geht mittels

```
struct node {
    int a;
    int b;
    unsigned int c;
};
```

Damit wird ein neuer Typ `struct node` eingeführt, der im Prinzip genauso benutzt wird wie `unsigned int`, aber mehrere Variablen enthält. Der Zugriff auf das Element `a` erfolgt über `meinNode.a`, wenn `meinNode` vom Typ `struct node` ist. Beispiel:

```
struct node meinNode;
meinNode.a = 32;
meinNode.b = -1;
meinNode.c = 7;

fprintf( "%d", meinNode.a );
```

Dynamische Speicherverwaltung und `structs`

Wir haben bereits gelernt, dass Felder vom Typ `unsigned int` angelegt werden können, obwohl ihre Größe womöglich erst zur Laufzeit bekannt ist (d.h. beispielsweise berechnet wurde). Das ging in `C` beispielsweise mit

```
unsigned int anzahl = 400;
unsigned int* einFeld =
        (unsigned int*)calloc(anzahl, sizeof(unsigned int) );

...

free(einFeld);
```

wobei jetzt `einFeld` ein *Zeiger* auf das erste Element ist. Das geht auch, wenn man Speicher für lediglich *ein* `unsigned int` *permanent* alloziieren will:

```
unsigned int* permanent =
            (unsigned int*)calloc(1, sizeof(unsigned int) );
*permanent = 7;

/* die Variable 'permanent' kann ab jetzt selbst dann noch
   abgerufen werden, wenn man sie aus einer Funktion
   zurueck gibt! */
```

```
unsigned int derWert = *permanent;
*permanent = 42;

....

free(permanent);
```

Die Freigabe mit `free()` muss seitens des Programmierers erfolgen, ansonsten bleibt die Variable solange gültig, bis das Programm beendet wird. Es ist semantisch falsch, allozierten Speicher nicht wieder freizugeben. Das `free(permanent)` muss *nicht* zwangsläufig in derselben Funktion geschehen, wie das `calloc()`.
Mit `structs` geht das genauso:

```
struct node* permanent =
            (struct node*)calloc(1, sizeof(struct node) );
(*permanent).a = 32;
(*permanent).b = -1;
(*permanent).c = 7;

/* dasselbe, nur mit weniger Tipperei: */
permanent->a = 32;
permanent->b = -1;
permanent->c = 7;

/* die Variable 'permanent' kann ab jetzt selbst dann noch
   abgerufen werden, wenn man sie aus einer Funktion
   zurueck gibt! */

fprintf( "%d", (*permanent).a );
....

free(permanent);
```

Hier muss man nur `(*permanent).a` anstelle von `meinNode.a` benutzen. Die Notation `(*permanent).a` ist äquivalent zu `permanent->a`.

Funktionen

Eine Funktion, die

- ein `struct node*` (d.h. ein Zeiger auf `struct node`) als erstes Argument akzeptiert,
- ein `unsigned int` als zweites Argument akzeptiert,
- und ein `struct node*` (d.h. ein Zeiger auf `struct node`) zurückliefert,

wird mit

```
struct node* aFunction( struct node* a, unsigned int b ) {
    struct node* returnValue = /* .... */
    ...
    return returnValue;
}
```

gebaut. Aufrufen kann man sie bspw. mit

```
int main(int argc, char** argv) {
    struct node* node1;
    struct node* node2;

    node2 = aFunction( node1, 42 );
    ...
}
```

Insbesondere ist es nun *innerhalb* von `aFunction()` möglich, mit `calloc()` Speicher zu alloziieren, der dann auch innerhalb von `main()` noch gültig ist. Das ist sonst *nicht* möglich.

Zeiger

Zeiger sind spezielle Variablen, die auf andere hinweisen. Man kann über Zeiger auf Variablen, die sonstwo stehen, lesend und/oder schreibend zugreifen. Beispiel:

```
unsigned int a = 42;
unsigned int b = 200;

/* mit '&' holt man die Adresse von 'a': */
unsigned int* zeiger_auf_a = &a;

fprintf( "%d", a ); /* liefert 42 */
a = 7;
fprintf( "%d", a ); /* liefert 7 */

*zeiger_auf_a = 16;
fprintf( "%d", a ); /* liefert 16 */

zeiger_auf_a = &b; /* zeiger auf was anderes setzen */
*zeiger_auf_a = 1;

/* liefert 16, denn der Inhalt von 'a' wurde nicht veraendert: */
fprintf( "%d", a );
```

Zeiger wurden oben schon bei dynamischer Feldverwaltung benutzt.

Weitere Hinweise

Weitere Hinweise finden Sie in der Literatur, siehe die Liste auf `http://wissrech.ins.uni-bonn.de/teaching/algmath/ws09_i`.