**Datum: 16.01.07**

# C Programmierung

## Übersicht:

- **Datenstrukturen**
- **Listen**
- **Bäume**
- **Hashtables**

Autor: Blue

Wikipedia definiert eine Datenstruktur folgendermaßen:

> Wikipedia ( http://de.wikipedia.org/wiki/Datenstruktur ):
>
> In der Informatik ist eine Datenstruktur eine bestimmte Art, Daten zu verwalten und miteinander zu verknüpfen, um in geeigneter Weise auf diese zugreifen und diese manipulieren zu können. Datenstrukturen sind immer mit bestimmten Operationen verknüpft, um eben diesen Zugriff und diese Manipulation zu ermöglichen.

Datenstrukturen dienen uns also um große Datenmengen zu speichern und zu verwalten. Sie machen Quellcodes komfortabler, verständlicher, erweiterbar und wieder verwertbar. Diese Eigenschaften machen die Strukturen unumgänglich.

**Inhaltsverzeichnis**

# 1. Einführung

## 1.1 Datenstrukturen deklarieren

```
struct <name>
{
	<Felder>
};
```

Beispiel:

```
struct Person{
	char vorname[10], nachname[10];
};
```

Strukturen können auch verschachtelt werden:

```
struct Telefon{
	struct Person p;
	int teleNR;
};
```

## 1.2 Datenstrukturen initialisieren

```
int main(void)
{
	struct Person a;
	struct Person b = {"Hans", "Mueller"};
	struct Telefon t = {("Hans", "Mueller"), 017600000};
	return 0;
}
```

## 1.3 Mit Datenstrukturen arbeiten

```
a = b;
printf("Vorname: %s\nNachname: %s\n", a.vorname, a.nachname);

strcpy(a.vorname, "Peter");
c.teleNR = 017611111;
```

## 2. Funktionen

Es können Strukturen als Funktionsparameter übergeben (4.1), wie auch an die Funktion zurückgeliefert werden (4.2).

### 2.1 Struktur als Funktionsparameter

```
void gebeDatenaus(struct Telefon t)
{
	printf("Vorname: %s\nNachname: %s\nTelefonnummer: %d\n",
		t.p.vorname, t.p.nachname, t.telNR);
}
```

### 2.2 Struktur als Rückgabewert

```
struct Telefon erstelleEintrag
		(char vorname[], char nachname[], int teleNR)
{
	struct Telefon eintrag;
	strcpy(eintrag.p.nachname, nachname);
	strcpy(eintrag.p.vorname, vorname);
	eintrag.telNR = teleNR;

	return eintrag;
}
```

# 3 Datenstrukturen optimieren

Man kann Datenstrukturen optimieren, indem sie performant sind, oder nur wenig Speicherplatz gebrauchen.

## 3.1 Zeiger auf Strukturen

Übergibt man eine Struktur an eine Funktion, so wird eine Kopie der Struktur auf dem Stack erstellt, mit der die Funktion dann arbeitet.
Bei diesem Verfahren sind die Rechenintensität und der Speicherverbrauch sehr hoch. Als Abhilfe übergibt man nun die Struktur nicht als Value, sondern als Referenz.

```
void aenderTeleNr(struct Telefon *t, int neueNummer)
{
	(*t).telNR = neueNummer;
}
```

```
int main(void)
{
	[…]
	aenderTeleNr(&a, 90088);
	[…]
}
```

## 3.2 Pfeil Operator

Da die Operation (*<Struktur>).<Feld> umständlich erscheint, hat man den Pfeil Operator (->) eingeführt.

```
void aenderTeleNr(struct Telefon *t, int neueNummer)
{
	t->telNR = neueNummer;
}
```

# 4 Dynamische Datenstrukturen

Bisher waren wir nur in der Lage eine statische Anzahl von Objekten zu erstellen. Um dies abzuschaffen und zu realisieren, dass man eine variable Anzahl von Objekten hat, wurden die folgenden Funktionen in die Runtime Environment aufgenommen

- malloc()
- calloc()
- realloc()
- free()

Bei der dynamischen Erstellung von Objekten, sind wir gezwungen, den Speicher wieder frei zu geben, dies geschieht mit free().

## 4.1 malloc()

Diese Funktion allokiert Speicher. Es dient dazu, Speicher zu reservieren, wenn wir ihn gebrauchen. Initiieren wir also ein Objekt wie üblich mit

```
struct Telefon Eintrag;
```

wird der Speicher direkt reserviert. Es kann also sein, dass wir Speicher für Objekte reservieren, die nicht gebraucht werden.

Dynamisch lässt sich dies mit malloc() lösen.
Dazu legen wir einen Zeiger auf unsere Struktur an. malloc() dient uns nun dazu, variable Mengen an Speicherplatz zu binden.

```
struct Telefon *Eintrag;
Eintrag = (struct Telefon*)malloc(sizeof(struct Eintrag));
[…]
Eintrag->telNR = 110;
```

Die malloc() Funktion kann natürlich analog auch auf andere Datentypen angewandt werden.

```
char *String;
int StringSize = 20;
String = (char *) malloc(StringSize);
strcpy(String, "hallo");
[…]
free(Eintrag);
```

## 4.2 calloc()

Wenn wir eine größere Menge an Variablen benötigten, dann haben wir uns vorher den Arrays bedient.

```
const unsigned int Elemente = 20;
struct Telefon Eintrag[Elemente];

Eintrag[0].telNR = 4556677;
```

Das Problem liegt wieder darin, dass den Speicher reservieren, ihn aber wohlmöglich nicht benutzen.

Hier kann die Funktion calloc() Abhilfe schaffen. Ihr muss zusätzlich noch die Anzahl Elemente übergeben als Funktionsparameter übergeben werden:

```
struct Telefon *Eintrag;
const unsigned int Elemente = 20;

Eintrag = (struct Telefon*)calloc(
      Elemente,
      sizeof(struct Telefon));

Eintrag->telNR = 4556677;

free(Eintrag);
```

Analog kann dieses ebenso auf andere Datentypen angewandt werden.

## 4.3 realloc()

Stelle man sich vor, dass man genau so viele Elemente initiiert haben will, wie Einträge vorhanden sind. Man müsste also, wenn ein Neues Element benötigt wird, ein größeres Array erstellen, die Daten aus dem alten Array kopieren und das neue Element anfügen. Das veraltete Array kann dann zerstört werden mit free().

An dieser Stelle tritt realloc() in Kraft. Es ist in der Lage, die Größe eines Arrays während der Laufzeit zu ändern, ohne das Datenverluste auftreten (Es sei denn, man Entfernt gewollt Elemente, die Daten beinhalten).

```
struct Telefon *Eintrag = NULL;
int Elemente = 0;
char vorname[20], nachname[20];
unsigned int mtknr;

do {
      fflush(stdin);
      printf("Vorname: ");
```

```
      gets(vorname);
      printf("Nachname: ");
      gets(nachname);
      printf("Mtknr.: ");
      scanf("%d", &mtknr);
      Elemente++;

      Eintrag = (struct Telefon*)realloc(
            Eintrag,
            Elemente * sizeof(struct Telefon));

      strcpy(Eintrag[Elemente-1].p.vorname, vorname);
      strcpy(Eintrag[Elemente-1].p.nachname, nachname);
      Eintrag[Elemente-1].telNR = mtknr;

} while(mtknr != 0);

for(int n = 0; n<Elemente; n++)
      ausgabe(Eintrag[n]);

free(Eintrag);
```