Praxis der Programmierung

Zusammengesetzte Datentypen, dynamische Speicherverwaltung

Institut für Informatik und Computational Science
Universität Potsdam

Henning Bordihn

Einige Folien gehen auf A. Terzibaschian zurück.

Zusammengesetzte Datentypen

Zusammengestzte Datentypen – Motivation

- Aufgabe: Kundenverzeichnis anlegen
  Für jeden Kunden erfassen:
  - Name,
  - Adresse,
  - Kundennummer,
  - Telefonnummer,
  - Umsatz,
  - ...

- Wie kann man das in C umsetzen?

Zusammengesetzte Datentypen – Motivation (2)

- Array für jeden Kunden?

  $\rightsquigarrow$ unmöglich wegen der versch. Datentypen

- Array für jedes Merkmal?

```
char* name[1024];
char* address[1024];
short phone[1024];
short id[1024];
float b_volume[1024];
...
```

  $\rightsquigarrow$ Zusammengehörgkeit der Variablen nicht sichergestellt

  $\rightsquigarrow$ Übergabe an Funktionen nur einzeln

Zusammengesetzte Datentypen – Motivation (2)

- Array für jeden Kunden? $\rightsquigarrow$ unmöglich wegen der versch. Datentypen
- Array für jedes Merkmal?

```
char* name[1024];
char* address[1024];
short phone[1024];
short id[1024];
float b_volume[1024];
...
```

$\rightsquigarrow$ Zusammengehörgkeit der Variablen nicht sichergestellt

$\rightsquigarrow$ Übergabe an Funktionen nur einzeln

- **Strukturen (structs)** zur Definition komplexer Datentypen, die sich aus mehreren Datentypen zusammensetzen

Stukturen

- **Strukturtyp:** - selbst definierter Datentyp
  - zusammengesetzt aus Komponenten *verschiedener Typen*

- Variable von Typ Struktur (Verbund/Record) kann Datensatz speichern

Kundennr.	Nachname	Vorname	Straße	Hausnr.	PLZ	Wohnort	Umsatz
`short`	`char[64]`	`char[64]`	`char[64]`	`short`	`short`	`char[64]`	`float`

- *Komponenten* haben einen eigenen Namen und Typ (statt Index)

Deklaration eines Strukturtyps

- ```
  struct name {
    komponententyp_1 komponentenname_1;
    komponententyp_2 komponentenname_2;
                          ⋮
    komponententyp_n komponentenname_n;
  };
  ```
- `struct` ist Schlüsselwort $\rightsquigarrow$ **Datentyp:** `struct name`
- Anzahl der Komponenten bei Deklaration festgelegt
- Semikolon nach } (kein Anweisungsblock!)
  ```

Deklaration eines Strukturtyps – Beispiel

```
char* name[1024];
char* address[1024];
short phone[1024];
short id[1024];
float b_volume[1024];
```

```
struct customer {
   short id;
   char name[64];
   char adress[64];
   short phone;
   float b_volume;
};

struct customer ctms[1024];
```

Strukturtypen – Warum?

Strukturtypen

- definieren immer einen Datentyp
- bilden die Zusammengehörigkeit von Daten ab
- bilden die Realität direkter ab

  $\rightsquigarrow$ bessere Modellierung von Anwendungsdomänen
  $\rightsquigarrow$ erleichtertes Code-Verständnis
  $\rightsquigarrow$ erleichterte Wartung des Codes

Strukturvariablen

- **Deklaration:** `struct name variablenname;`
  $\rightsquigarrow$ Variable vom zusammengesetzten Typ `struct name`

- besteht aus mehreren Komponentenvariablen
  $\rightsquigarrow$ bei Stukturtypdeklaration vereinbart

- gleichzeitige Vereinbarung von Strukturtyp und -variablen möglich:

```
struct point {          // Def. neuer Datentyp (Strukturtyp)
  float x;
  float y;              // zwei Komponenten (Member) x und y
} pt1;

struct point pt2, pt3;
```

  $\rightsquigarrow$ `pt1, pt2, pt3` als (Struktur-) Variablen dieses neuen Datentyps deklariert

Zugriff auf Komponentenvariablen

1. **Punktoperator:** *Strukturvariable*.*Komponentenvariable*

   - Beispiele: `pt1.x`
     `meyer.name`
     `meyer.address`

   - lesender und schreibender Zugriff:

```
strcpy(meyer.name, "Meyer, Jens");
strcpy(meyer.address, "14482 Potsdam, A-Bebel-Str. 89");
printf("%s\n\%s\verb\n",
  meyer.name, meyer.address);
```

2. **Pfeiloperator:** *Pointer_auf_Strukturvariable*->*Komponentenvariable*

   - Beispiel: `(&pt1)->x`
     `(&meyer)->name`

   - Pfeil: Minuszeichen und Größerzeichen

   - lesender und schreibender Zugriff:
     ```
     printf("Adresse: %s\n", (&meyer)->address);
     ```

   - Wie kann die Adresse in ihre Komponenten zerlegt werden?
     $\rightsquigarrow$ Strukturen als Komponenten von Strukturen

Strukturen als Komponenten von Strukturen

```
struct address {
   char street[64];
   short number;
   short zip_code;
   char city[64];
};

struct customer {
   char name[64];
   struct address adr;
   ...
};
```

```
struct customer meyer;

strcpy(meyer.name, "Meyer, Jens");
strcpy(meyer.adr.street,
                    "A.-Bebel-Str.");
meyer.adr.number = 89;
```

Initialisierung von Strukturvariablen

1. mit Punkt- oder Pfeiloperator
2. mit Initialisierungslisten
   - nur direkt bei der Definition der Strukturvariablen
   - wie bei Arrays (mit Ausdrücken passenden Typs), z.B.:

```
struct customer meyer = {
    "Meyer, Jens",
    { "A.-Bebel-Str.", 89, 14482, "Potsdam" }
    ...
};
```

3. Zuweisung von struct-Variablen:

```
customer krause = meyer;  // Kopie der WERTE aller Elemente!!!
```

Strukturen als Parameter und Rückgabewerte von Funktionen

- Voraussetzung: Deklaration des Strukturtyps *außerhalb* und *vor* den Funktionen
- Übergabe wie einfache Datentypen (call-by-value beachten!):

```
void print_customer (struct customer c) {
   printf("%s\n", c.name);
   printf("%s %d\n", c.adr.street, c.adr.number);
   printf("%d %s\n", c.adr.zip_code, c.adr.city);
}

struct point new_point () {
   return struct point new = {0, 0};
}
```

$\rightsquigarrow$ oft unnötiges Kopieren großer Datenmengen im Speicher

Pointer auf Struktuen als Parameter

- Simulation von call-by-reference durch Übergabe von Pointern:

```
void print_customer (struct customer * c) {
   printf("%s\n", c->name);
   printf("%s %d\n", c->adr.street, c->adr.number);
   printf("%d %s\n", c->adr.zip_code, c->adr.city);
}

int main() {
   struct customer meyer = { ... };
   print_customer(&meyer);
}
```

- Veränderung der Werte des Originals (c) möglich
- Gefahr: Parameter kann nicht oder mit `NULL` initialisiert sein (*Laufzeitfehler*)

Vereinbarung eigener Typnamen

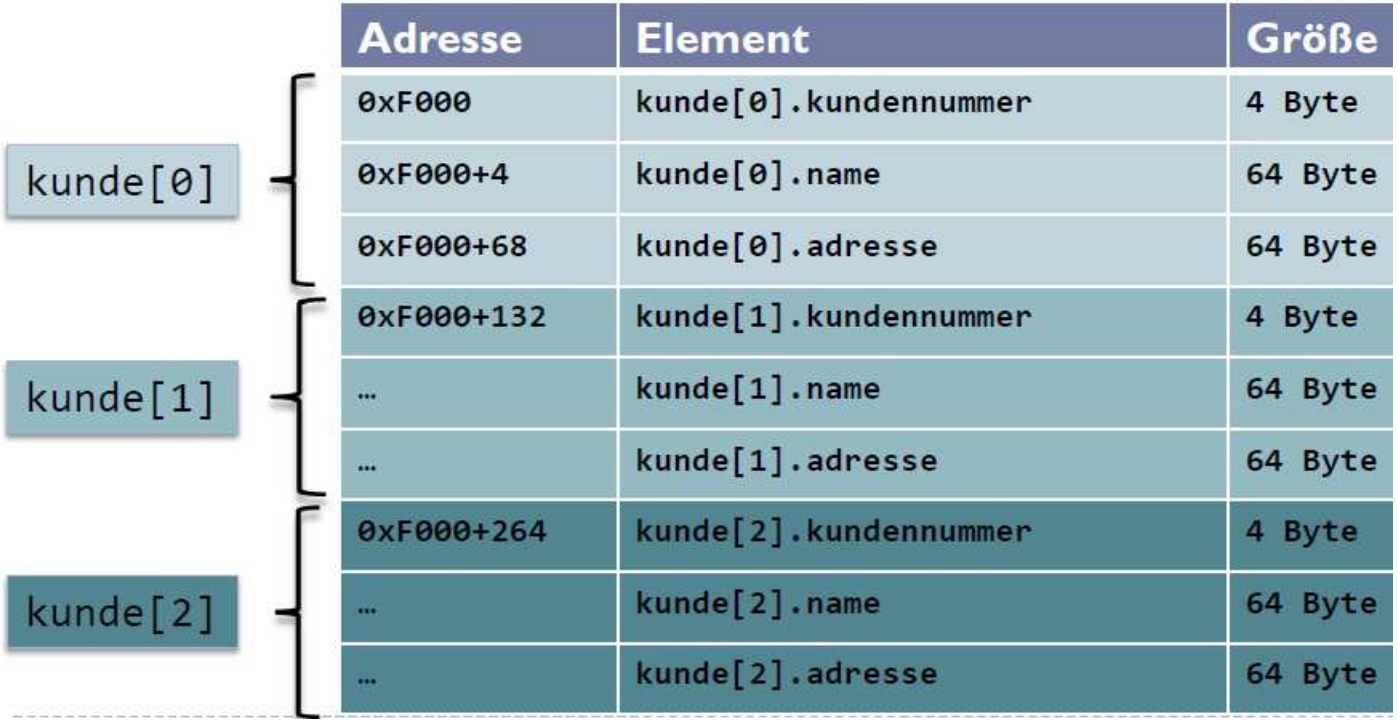
- Vereinbarung eines *Aliasnamens* für bereits deklarierte Datentypen
- `typedef` *Datentyp Aliasname*`;`
- `typedef int integer;`
- Anwendung:
  - Vereinfachung von Typnamen
    - `typedef struct customer Customer;`
      `Customer meyer;`
    - `typedef unsigned long long ULL`
  - Vorbereitung Portierung maschinenabhängiger Datentypen
    - `typedef int INT;` $\rightsquigarrow$ `typedef short INT;`

Strukturen im Speicher

Elemente von Strukturvariablen liegen hintereinander im Speicher, z.B.:

```
struct  kt {
    int kundennummer;
    char name[64];
    char adresse[64];
};

struct kt kunde[3];
```



Adresse	Element	Größe
0xF000	kunde[0].kundennummer	4 Byte
0xF000+4	kunde[0].name	64 Byte
0xF000+68	kunde[0].adresse	64 Byte
0xF000+132	kunde[1].kundennummer	4 Byte
…	kunde[1].name	64 Byte
…	kunde[1].adresse	64 Byte
0xF000+264	kunde[2].kundennummer	4 Byte
…	kunde[2].name	64 Byte
…	kunde[2].adresse	64 Byte

Dynamische Speicherverwaltung

Datentypen im Speicher

- *bisher:* Größe und Anzahl der Variablen bei Deklaration festlegen
  - Variablen elementarer Datentypen (auch Pointer-Variablen)
  - Strings, Arrays
  - Strukturen (structs)

- **Problem:** Größe von Arrays/Strings kann oft erst zur Laufzeit bestimmt werden
  $\rightsquigarrow$ immer maximale Größe annehmen ist oft kritische Speicherverschwendung

- **Lösung:** dynamische Speicherreservierung "on demand"

- **Nachteile:** komplexere Programmlogik, hohe Fehleranfälligkeit, möglicher Datenverlust

Speicherbereiche eines Programms

1. Bereich für ausführbaren Maschinencode
   - Bereich für Maschinencode (kompilierte Anweisungen)
   - Bereich für statische Daten (Konstanten und statische Strings)

Speicherbereiche eines Programms

1. Bereich für ausführbaren Maschinencode
   - Bereich für Maschinencode (kompilierte Anweisungen)
   - Bereich für statische Daten (Konstanten und statische Strings)
2. Programm-Stack (zur Compile-Zeit angelegt)
   - Variablen, Arrays, Funktionsparameter
   - temporäre Werte (bei Berechnungen)
   - Aufrufstack für Funktionen

Speicherbereiche eines Programms

1. Bereich für ausführbaren Maschinencode
   - Bereich für Maschinencode (kompilierte Anweisungen)
   - Bereich für statische Daten (Konstanten und statische Strings)
2. Programm-Stack (zur Compile-Zeit angelegt)
   - Variablen, Arrays, Funktionsparameter
   - temporäre Werte (bei Berechnungen)
   - Aufrufstack für Funktionen
3. Programm-Heap (zur Laufzeit verwaltet)
   - dynamisch als Pointer auf den Heap angefordert (in C: `malloc()`)
   - dynamisch wieder freigegeben (in C: `free()`)

Bibliotheksfunktionen zur dynamischen Speicherverwaltung

- in `<stdlib.h>`
- zum **Anfordern** von Speicher (liefert Pointer zurück), *z.B.:*

```
void * malloc(size_t size)
```

- *Beispiel:* ganze Zahlen von 1 bis N in einem Array speichern

```
int N, i;
int * nums;
scanf("%d", &N);
nums = malloc(N * sizeof(int));
for(i = 0; i < N; ++i) {
   nums[i] = i+1;
}
```

Bibliotheksfunktionen zur dynamischen Speicherverwaltung (2)

- **Freigeben** von reserviertem Speicher (Pointer auf Speicherbereich übergeben):

```
void free(void * pointer)
```

- *am Beispiel:*

```
free(nums);   // Speicherbereich zurueckgegeben
nums = NULL;  // nums zeigt nicht mehr auf den Speicherbereich
```

- keine automatische Speicherfreigabe

⇝ Zu jedem `malloc()` gehört (irgendwann) genau ein `free()`.

Probleme mit dynamischer Speicherverwaltung

- Programmierer ist voll verantwortlich
- keine Freigabe $\rightsquigarrow$ Speicher läuft voll mit Daten,
  auf die nicht mehr zugegriffen werden kann / die nicht mehr gebraucht werden
- Freigabe eines nicht (mehr) alloziierten Bereichs
  $\rightsquigarrow$ Programmabsturz
- korrekte/geeignete Stelle zur Speicherfreigabe oft nicht klar:

```
char * create_string(int n) {  // n ist Laenge des Strings
      char * str;
      str = malloc((n+1)*sizeof(char));
      return str;
      free(str); // Problem???
}
```

Dynamisch gespeicherte Daten als Rückgabewerte

```
 char * create_string(int n) {          // n ist Laenge des Strings
        char * str;
        str = malloc((n+1)*sizeof(char));
        return str;
}

void delete_string(char * s) {
        if (s != NULL) {
                free(s);
            //  s = NULL; hier nutzlos - warum?!
        }
}
```

Zu jedem Aufruf von `create_string` gehört genau ein Aufruf von `delete_string`, gefolgt von Zuweisung des `NULL`-Pointers!