Speicherverwaltung und Datenstrukturen

C - Kurs 2010

Florian Streibelt

freitagsrunde.org

16. September 2010

Inhalt

Inhalt

Wiederholung

Kurze Zusammenfassung an der Tafel:

- Arrays
- Pointer
- Pointerarithmetik
- Strings

Inhalt

Vorschau

Wir betrachten jetzt:

- dynamische Speicherbelegung
- eigene Datentypen
- dynamische Datenstrukturen
- ein Beispielprogramm, das alles nutzt
- Ausblick auf morgen

Inhalt

Arrays sind unflexibel

- Arrays haben eine feste Größe
- Man muss vorher wissen, wie groß sie sein sollen
- Das gibt Probleme bei 'Anwendungen':
  - Supermarktkasse: Wieviele Produkte?
  - Messwerterfassung: Daten wachsen ständig.
  - ...

Welche Möglichkeiten könnte es geben, diese Probleme zu vermeiden?

Arrays sind unflexibel (Lösungsideen)

- 1. Idee: Man kann den Inhalt jedes Mal in ein größeres Array kopieren... (langsam)
- 2. Idee: Man kann gleich ein genügend großes Array definieren (wie groß?)
- 3. Idee: Man kann Speicher dynamisch nach Bedarf anfordern

Welches von den Mitteln aus dem Kurs werden wir für die dritte Idee mit Sicherheit brauchen?
$\Rightarrow$ Pointer: Sie zeigen auf Speicherbereiche und sind veränderlich!

Inhalt

Dynamischer Speicher

- Über einen Aufruf fordern wir Speicher vom Betriebssystem
- Wenn genügend vorhanden ist, erhalten wir einen Pointer
- Dieser zeigt dann auf ein entsprechend großes Stück Speicher

Beispiel: Speicher anfordern

```
1 [...]
2 char *foo = (char *) malloc(1024 * sizeof(char) );
3 printf("Speicher: %s\n", foo);
```

```
Speicher: ]&DS)=*'V+PadsK...'|+&
```

... nur kann in dem Speicher noch etwas drinstehen.

malloc explained

Beispiel: Speicher anfordern

```
1 [...]
2 char *foo;
3 foo = (char *) malloc(1024 * sizeof(char) );
```

- Einen Pointer 'foo' anlegen
- 1024 byte Speicher anfordern
- Speicher auf Zieltyp des Pointers casten

calloc explained

Beispiel: Nullinitialisierten Speicher anfordern

```
1 [...]
2 char *foo;
3 foo = (char *) calloc( 1024 , sizeof(char) );
```

- Einen Pointer 'foo' anlegen
- 1024 Elemente der Größe von char anfordern
- Speicher auf Zieltyp des Pointers casten

free nicht vergessen

In C muss man Speicher manuell wieder freigeben:

Beispiel: Aufruf von free()

```
1 char *foo;
2 foo = (char *) calloc( 1024 , sizeof(char) );
3 free(foo);
```

`free()`...

- Kann nur Speicher freigeben, der mit `malloc` oder `calloc` alloziiert wurde,
- darf nicht mit anderen Pointern aufgerufen werden,
- darf nur einmal pro `malloc/calloc` aufgerufen werden, sonst 'double free'-Fehler.
- `free` am besten gleich mit calloc/malloc in den Code schreiben.

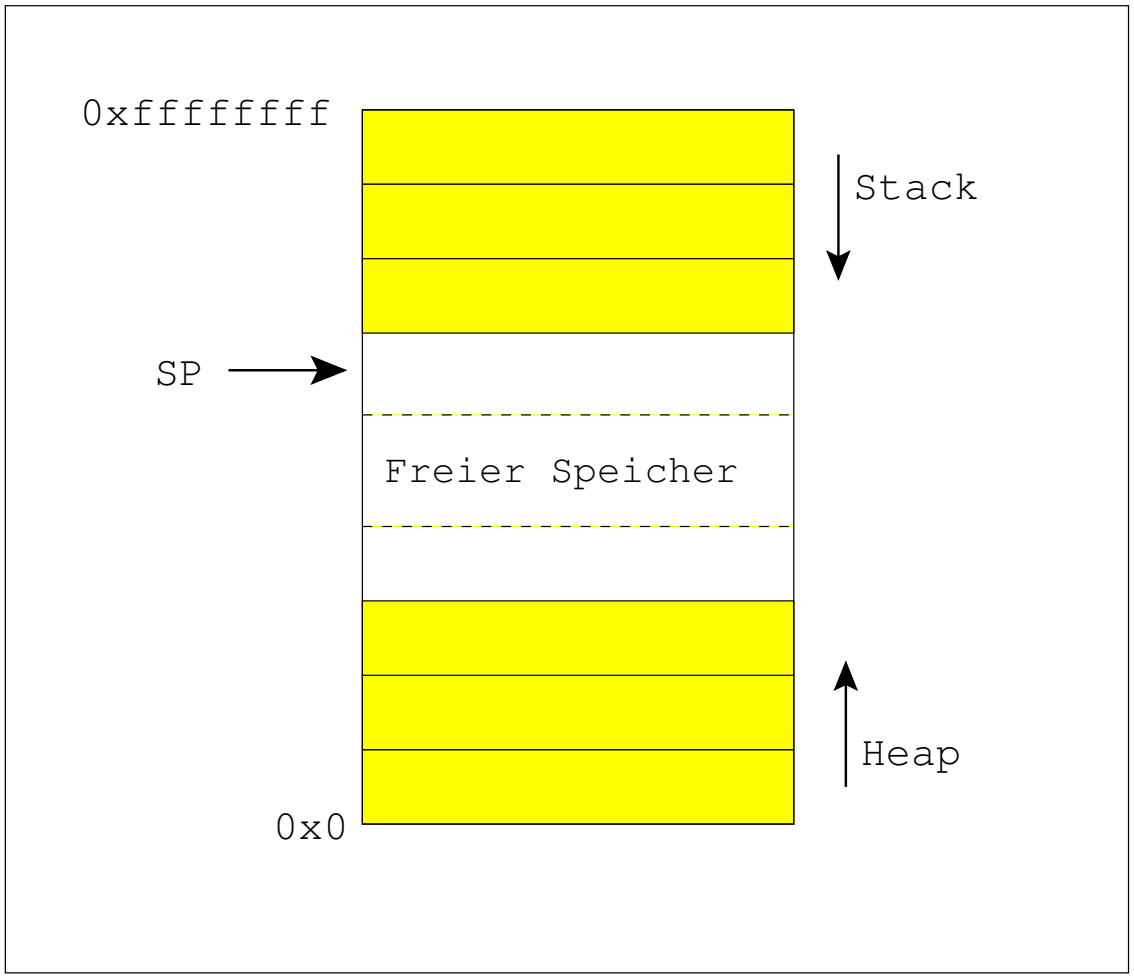
Inhalt

Heap und Stack

Was sind Heap und Stack?

- Sind nur zwei unterschiedliche Speicherbereiche
- Einer am Anfang des Adressraums, einer am Ende
- Kann je nach CPU/Betriebssystem abweichen
- Pointer für beide in Betriebssystem/CPU
- Bei Belegung wachsen sie aufeinander zu
- Heap: `malloc`, `calloc`, `free`
- Stack: lokale Variablen (und `alloca`)
  $\Rightarrow$Stackframe: Parameter fuer Funktionen, Rücksprungadresse
- Es gibt noch mehr Sonderbereiche (z.B.: `Data Segment`)

Heap und Stack (Skizze)



Finde den Bug

Beispiel: Fehlerhaftes Programm

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  char* foo(){
5      char buf[] = "Hello world!";
6      return buf;
7  }
8
9  int main(int argc, char** argv){
10     char* bar = foo();
11     // weitere Funktionsaufrufe
12     printf("%s\n",bar);
13 }
```

```
# ./a.out
¿Æ.
```

Stack: Confusion on return

Was ist hier passiert?

- Das array ist als lokale Variable in der Funktion definiert
- Sein Inhalt liegt also auf dem Stackframe der Funktion
- Beim `return` wird die Adresse zurückgegeben; sie zeigt auf den Stackframe
- Der Stack wird beim `return` abgeräumt, da er nicht mehr benutzt wird
- Ein folgender Funktionsaufruf überschreibt die Daten des Arrays
- Mancher Compiler warnt beim Übersetzen:

```
# gcc heapstack.c
heapstack.c: In function 'foo':
heapstack.c:7: warning: function returns address
of local variable
```

Korrigierte Version

Beispiel: Korrekte Fassung

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  char* foo(){
4      char *bar = calloc( 13 , sizeof(char));
5      snprintf(bar,13,"Hello world!");
6      return bar;
7  }
8  int main(int argc, char** argv){
9      char* bar = foo();
10     // ...
11     printf("%s\n",bar);
12     free(bar);
13 }
```

```
# ./a.out
Hello world!
```

Zur Benutzung von Heap und Stack...

- Zwei getrennte Speicherbereiche
- Heap: `malloc`, `calloc`, `free`
- Stack: lokale Variablen (und `alloca`)
- Niemals Pointer auf Stackvariablen zurückliefern
- Der Stack wird beim return 'vernichtet'
- Stackvariablen als Argumente für Unterfunktionen kein Problem/normal
- Immer auf den Platz achten, sonst u.U. 0verfl0ws

Einschub: Bufferoverflow

Ein gefährliches Programm

```
1  //...
2  int isAdmin(const char *username){
3      int ret=0;
4      char tmp[10];
5      strcpy(tmp,username);
6      // ...
7      return ret;
8  }
9  int main(int argc,char** argv){
10     int admin =  isAdmin(argv[1]);
11     printf("Zugriffsrechte = %i\n",admin);
12 }
```

```
#./a.out 12345678911
Zugriffsrechte = 49
```

Inhalt

Wie organisiere ich meine Daten?

Es gibt verschiedene Möglichkeiten, Daten in einem Programm zu 'organisieren'.

- Alles als globale Variablen:
  $\Rightarrow$ sehr fehlerträchtig, Probleme bei Tests, Parallelisierung
- Alle einzeln als Parameter übergeben:
  $\Rightarrow$ unübersichtliche Aufrufe, sehr viel Tipparbeit
- Als BLOB in einem Speicherbereich
  $\Rightarrow$ unübersichtlich, unklare Datenformate, ...

Außerdem können diese Ideen alle nicht mit dynamischen Daten umgehen.

Übergabe in einem 'Blob'

Negativbeispiel: Datenübergabe als Blob

```
1  void print_item(const char *blob){
2    char menge[3];
3    strncpy(menge, blob , 3);
4    char name[27];
5    strncpy(name, (blob+3) , 27);
6    //[...]
7    printf("%s mal %s \n",menge,name);
8  }
9     //[...]
10  char *listeneintrag = calloc (50,sizeof(char));
11  memcpy (listeneintrag +  0 ,"05"         , 2); // Menge
12  memcpy (listeneintrag +  3 ,"Club Mate", 9); // Name
13  memcpy (listeneintrag + 30 ,"1,00"       , 4); // Preis
14  memcpy (listeneintrag + 40 ,"5,00"       , 4); // Summe
15  print_item(listeneintrag);
```

Nachteile dieser 'Lösung'

Der Vorschlag eben war selbst eher ein Problem, denn eine Lösung:

- Das war alles sehr unübersichtlich und zerbrechlich
- Jede Menge magic numbers, die Positionen müssen stimmen
- die Größe ist doch wieder festgelegt
- auch wenn man die Daten mit 'Trennzeichen' übergibt wird es nicht besser
- einzelne Elemente zu vergrößern ist unmöglich
- keine festen Datentypen, alles 'Strings'
- nachträgliches Ändern der 'Datenstruktur' unmöglich
- ...

Ich möchte die Daten also definitiv besser strukturieren!

Vorteile eigener Datentypen

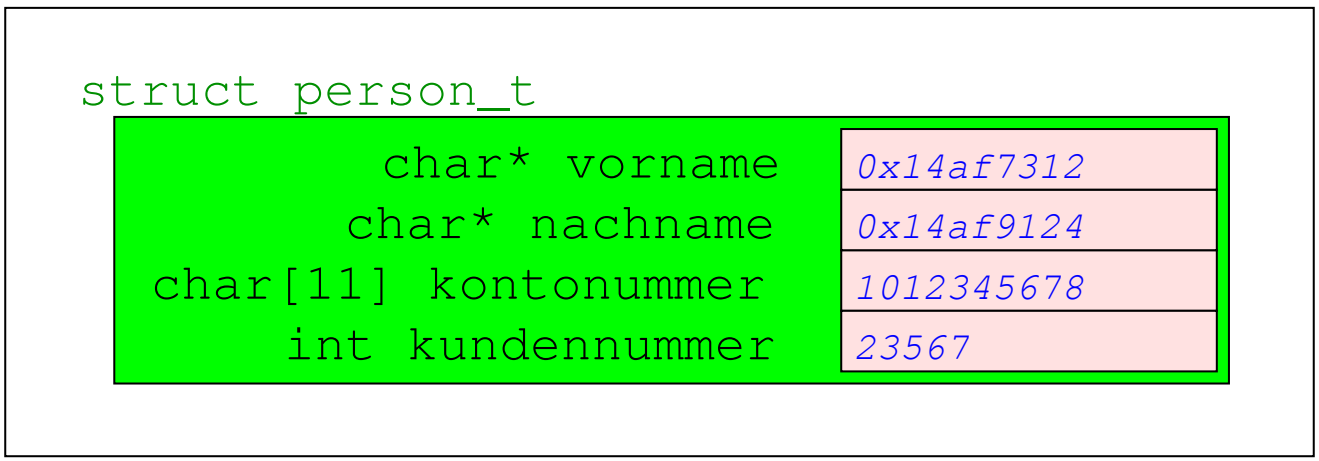
Eigene Datentypen haben viele Vorteile:

- Nur ein Parameter bei Übergabe an Funktionen
- Was zusammengehört bleibt zusammen
- Es erhöht den Überblick:
  `print_user(user);`
  vs.
  `print_user(vorname,nachname,geburtstag,email,passwort,...)`
- sie können dynamisch wachsen, erweitert werden, ...

Definition eines eigenen Datentyps

Beispiel: Person als Datentyp

```
1 typedef struct person_t {
2       char* vorname;
3       char* nachname;
4       char  kontonummer[11];
5       int   kundennummer;
6  } person;
```



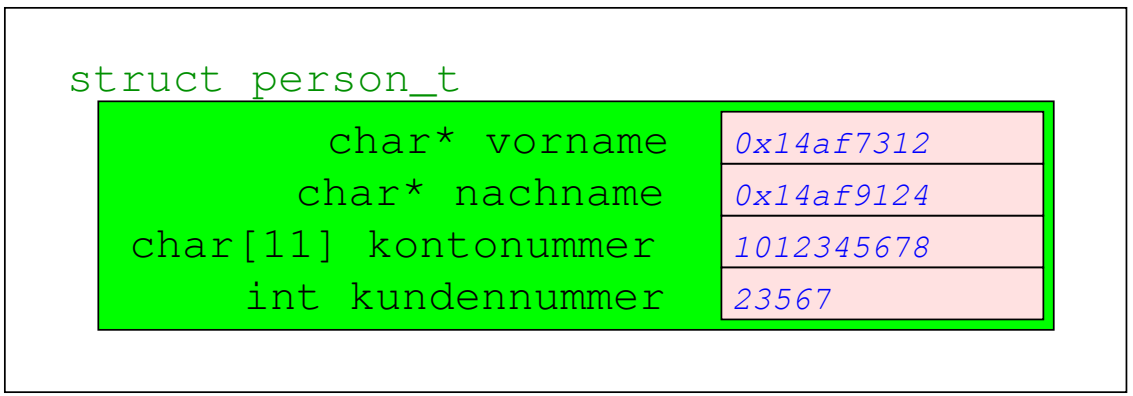
Benutzen eines eigenen Datentyps

Beispiel: struct initialisieren

```
1  typedef struct person_t {
2        char* vorname;
3        char* nachname;
4        char  kontonummer[11];
5        int   kundennummer;
6  } person;
7
8  [...]
9  person* p = (person*) calloc(1,sizeof(person));
10
11 p->kundennummer=1192;
12
13 snprintf(p->kontonummer,11,"%s","1012345678");
14
15 p->vorname=(char*)calloc(8,sizeof(char));
16 snprintf(p->vorname,8,"%s","Florian");
```

Structs im Speicher

- Alle Elemente liegen hintereinander im Speicher,
- ausgerichtet an bestimmten Adressen (evtl. padding).
- In der Reihenfolge wie in der Struct definiert.
- Man kann jedes Element mit Namen ansprechen,
- und natürlich auch einzeln ändern.



Flexibility for the win!

Das Konzept der `struct` kann nun noch erweitert werden: `unions`

- Was, wenn ich sehr ähnliche Datentypen speichern will?
- Ich kann evtl. erst zur Laufzeit den genauen Subtyp festlegen
- Ich möchte einen 'globalen' Datentyp 'Produkt'
- Aber zur Laufzeit bekomme ich Produkte die unterschiedlich abgemessen werden: Nach Stück, Volumen, Gewicht, ...

Verschiedene Produkte als structs

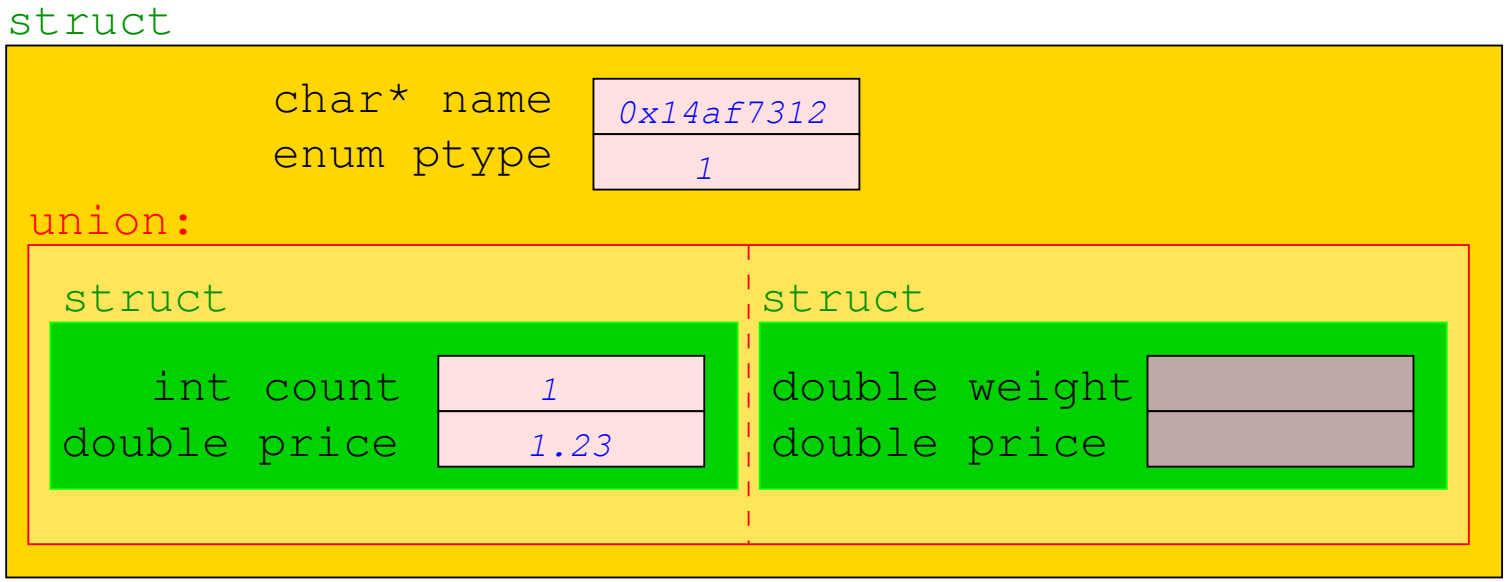
```
1  typedef struct productPieces_t {
2      char   *name;
3      int    count;
4      double price_p_piece;
5   } productPieces;
6
7  typedef struct productKGrams_t {
8      char   *name;
9      double weight;
10     double price_p_kilo;
11  } productKGrams;
12
13 typedef struct productLiters_t {
14     char   *name;
15     double liter;
16     double price_p_liter;
17  } productLiters;
```

Eine Union von structs

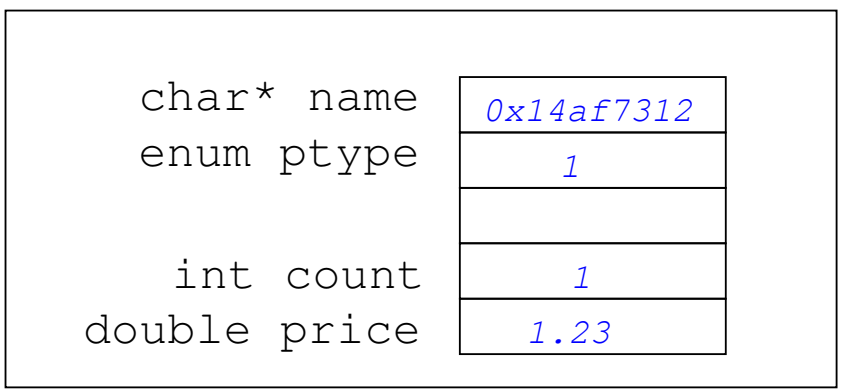
```
typedef struct product_t {
  char    *name;
  enum _punit { UNDEF, Pieces, KGrams, Liters} punit;
  union {
      struct{
          int     count;
          double  price_p_piece;
      } ppiece;
      struct{
          double  weight;
          double  price_p_kilo;
      } pkilo;
      struct {
          double  liter;
          double  price_p_liter;
      } pliter;
  };
} product;
```

Eine Union von structs

Logische Sicht:



"Speicherabbild":



Benutzung von unions

Beispiel: Ein Wirsingkohl

```
1
2  [...]
3
4  int main(int argc, char** argv){
5
6    product  *kohl = (product*) calloc(1,sizeof(product));
7    kohl->name= (char*) calloc(12,sizeof(char));
8    strcpy(kohl->name,"Wirsingkohl HKl.1");
9    kohl->punit=KGrams;
10   kohl->pkilo.weight=812;
11   kohl->pkilo.price_p_kilo=1.29;
12
13   double price = calc_price(kohl);
14
15 }
```

Benutzung von unions: Preis berechnen

```
1  double calc_price(product *foo){
2   double sum=-1;
3   switch (foo->punit){
4    default: printf ("TODO:ERRORHANDLING.\n"); break;
5    case KGrams:
6     sum=(foo->pkilo.weight * foo->pkilo.price_p_kilo)/1000;
7     break;
8    case Pieces:
9     sum=foo->ppiece.count * foo->ppiece.price_p_piece;
10    break;
11   case Liters:
12    sum=foo->pliter.liter * foo->pliter.price_p_liter;
13    break;
14  }
15  return sum;
16 }
```

Unions: Eigenschaften

- Eine `union` gibt Elementen im Speicher einen zweiten Namen und Typ.
- Die Elemente der Union überlappen sich/teilen sich den Speicher.
- Unions sind so groß wie das größte enthaltene Element.
- Greift man unter dem anderen Namen auf den Speicher zu, ändern sich die Typen
- Das kann praktisch sein, um Dateiformate zu lesen oder Daten zu konvertieren
- Wenn man nicht aufpasst, kann es unübersichtlich werden $\Rightarrow$ vorher planen.

Inhalt

Zurück zum Thema

Aber eigentlich wollten wir doch darüber sprechen, wie wir Daten unbekannter Größe verwalten? (Supermarktkasse)

Die Anforderungen:

- Unbekannte Menge an Produkten auf dem Band
- Produkte nach Gewicht oder Stück auf dem Band
- Funktionen sollen einfach aufzurufen sein
- Daten sollen sinnvoll strukturiert sein

Für die letzten beiden Punkte haben wir das Handwerkszeug, jetzt kommt noch die Lösung für das 'Fließbandproblem': dynamische Listen!

dynamische Listen

Wer hält uns davon ab, innerhalb einer Struct einen Pointer auf ein weiteres Element diesen Typs zu setzen? Niemand:

Beispiel: ein Listenelement

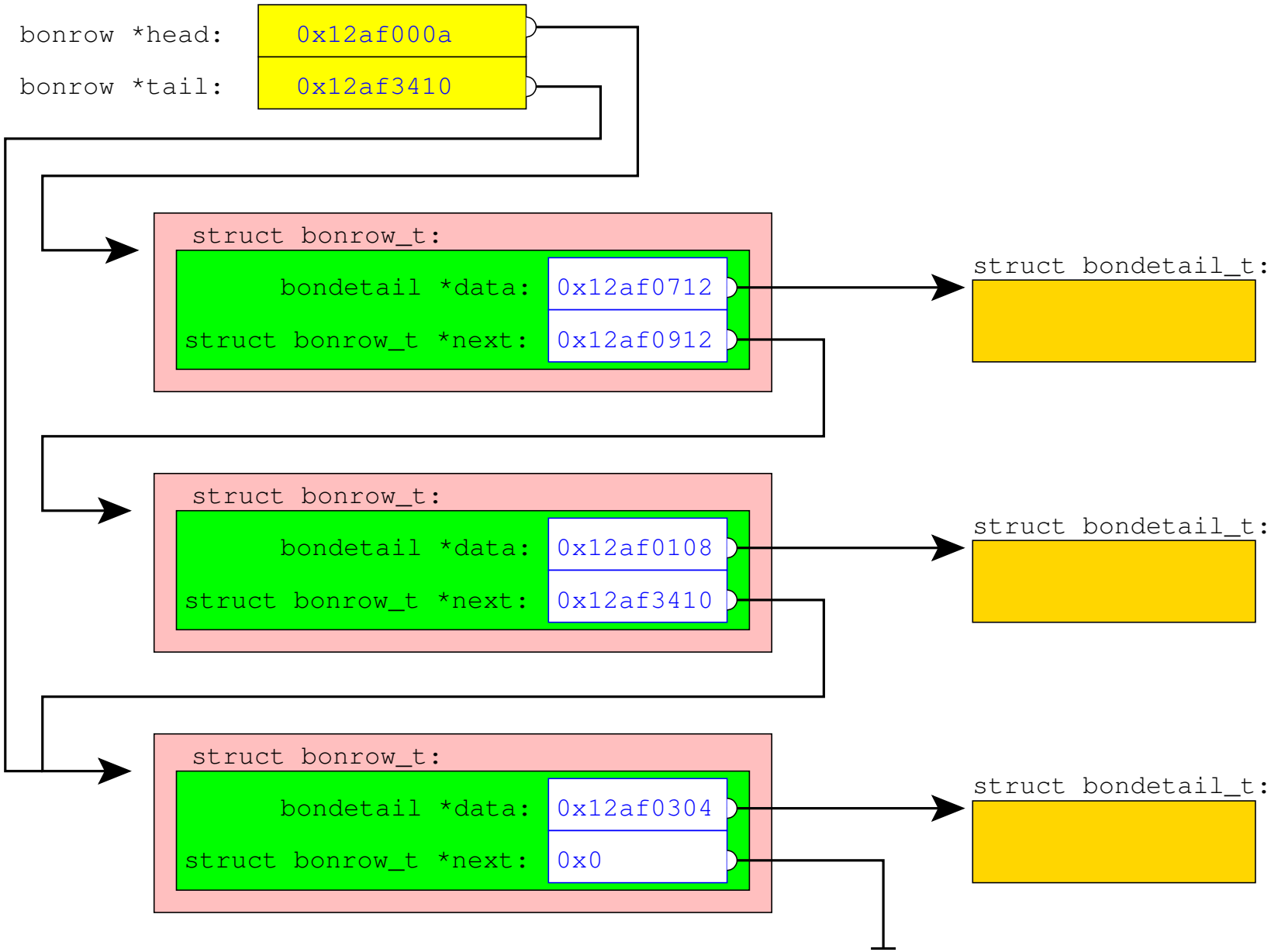
```
1
2  typedef struct bonrow_t{
3    bonelement *data;        // enthaltene Daten
4    struct bonrow_t *next;   // naechster Eintrag
5  }bonrow;
```

dynamische Listen(cont.)

Beispiel: Listendefinition

```
1
2  typedef struct bondetail_t{
3      product *prod;
4      double pricesum;
5  } bondetail;
6
7
8  typedef struct bonrow_t{
9    bondetail* data;
10   struct bonrow_t* next;
11 }bonrow;
```

dynamische Listen(Bild)



dynamische Listen (Bsp.)

Beispiel: Liste

```
1  bonrow *k_head, *k_tail;
2
3  //first element
4  k_tail = calloc(1,sizeof(bonrow));
5  k_head = k_tail;
6  k_tail->next = NULL;
7  k_tail->data = newPiecesP(5,"Club Mate",0.75);
8
9  //next element
10 k_tail->next=calloc(1,sizeof(bonrow));
11 k_tail=k_tail->next;
12 k_tail->next=NULL;
13 k_tail->data= newKilosP(750,"Wiener",4.90);
14
15 double sum = calc_sum(k_head);
16 print_bon(k_head,sum);
```

dynamische Listen (Bsp., Forts. 1)

```
1  bondetail* newKilosP(double amount, char* name,
2                                      double price)
3  {
4      product *prod =
5              (product*) calloc(1,sizeof(product));
6
7      prod->name=strdup(name);
8      prod->punit=KGrams;
9      prod->pkilo.weight=amount;
10     prod->pkilo.price_p_kilo=price;
11
12     bondetail *item =
13       (bondetail*) calloc(1,sizeof(bondetail));
14
15     item->prod=prod;
16
17     return item;
18 }
```

dynamische Listen (Ablaufen)

Beispiel: Durchlaufen der Liste

```
1  double calc_sum(bonrow* firstelement){
2    bonrow* current_row = firstelement;
3    double sum=0;
4    while(current_row!=NULL){
5      double price = calc_price(current_row->data->prod);
6      current_row->data->pricesum=price;
7      sum+=price;
8      current_row=current_row->next;
9    }
10   return sum;
11 }
```

dynamische Listen (Fallunterscheidung nach Subtyp)

Beispiel: Liste - Fallunterscheidung

```
1  double calc_price(product *foo){
2   double sum=-1;
3   switch (foo->punit){
4      default:  printf ("ERROR.\n");
5                    break;
6      case KGrams:
7      sum=(foo->pkilo.weight * foo->pkilo.price_p_kilo)/1000;
8                    break;
9      case Pieces:
10     sum=foo->ppiece.count * foo->ppiece.price_p_piece;
11                   break;
12     case Liters:
13     sum=foo->pliter.liter * foo->pliter.price_p_liter;
14                   break;
15      }
16     return sum;
17 }
```

dynamische Listen (finally)

Beispiel: Liste - die Ausgabe

```
# ./a.out
                        CKURS 2010
      Wir lieben C! Immer in Ihrer Nähe.

 Menge Produkt                              EP      GP
    5  Club Mate                            0.75    3.75
  750g Wiener                               4.90    3.68
   75L Vollmilch 3.9%                       1.90  142.50
                           MwSt (17%):       25.49
                          GESAMTPREIS:      149.93
```

Inhalt

Ausblick

- Vormittags
  - Der Compiler und seine Features
  - Was sind Headerfiles, wie nutzt man sie
  - Unser Codeassistent: der Präprozessor
- Nachmittags
  - Fehler finden, Debugging

Sie können den Computer jetzt wegwerfen...
`http://docs.freitagsrunde.org/Veranstaltungen/ckurs_2010/vortraege/04/`