

Hardwarenahe Programmierung / Angewandte Informatik

Musterlösung zu den Übungsaufgaben – 9. Januar 2017

Aufgabe 1: Objektorientierte Programmierung mit dem C-Datentyp `union`

(Klausuraufgabe aus dem Wintersemester 2014/15)

In Wintersemester 2014/15 wurde der Datentyp `union` in der Vorlesung *nicht* behandelt.)

Aus der Vorlesung ist der Verbund-Datentyp `struct` bekannt.

Die Programmiersprache C kennt noch einen weiteren Verbund-Datentyp `union`:

```
typedef union
{
    int number;
    char *name;
    uint8_t bytes[4];
} data;
```

Von der Syntax her entspricht die `union` genau dem `struct`. Insbesondere lassen sich die Datenfelder über einen Punkt ansprechen, bei Zeigern auch mit `—>`. Der Unterschied zum `struct` besteht darin, daß sich in einer `union` die Datenfelder *dieselben Speicherzellen teilen*. Alle Datenfelder einer `union` haben *dieselbe Speicheradresse*. (In einem `struct` beginnt das nächste Datenfeld immer dort, wo das vorherige aufhört.)

Bei Zuweisung eines Wertes an *ein* Datenfeld einer `union` ändern sich daher die Werte *aller* Datenfelder. Dies kann man nutzen, um Speicher zu sparen (wenn man immer nur eines der Datenfelder auf einmal nutzen möchte) oder um „verwandte“ Datentypen zu konstruieren (im Sinne der objektorientierten Programmierung) oder um die Byte-Muster von Variablen zu untersuchen.

- (a) Was bewirkt das folgende Programm ([aufgabe-1a.c](#)), wenn es auf einem LittleEndian-Rechner ausgeführt wird, und warum? (4 Punkte)

```
#include <stdio.h>
#include <stdint.h>

typedef union
{
    uint32_t number;
    char *name;
    uint8_t bytes[4];
} data;

int main (void)
{
    data x;
    x.number = 303108111;
    for (int i = 0; i < 4; i++)
        printf ("%d_", x.bytes[i]);
    printf ("\n");
    printf ("%s\n", x.name);
    return 0;
}
```

- (b) Was würde dasselbe Programm auf einem BigEndian-Rechner ausgeben? (1 Punkt)

Lösung

- (a) Was bewirkt das folgende Programm ([aufgabe-1a.c](#)), wenn es auf einem LittleEndian-Rechner ausgeführt wird, und warum?

Das Programm gibt die Zahlen

15 16 17 18

aus und stürzt anschließend mit einem Speicherzugriffsfehler ab.

Begründung:

Die Datenfelder `number`, `name` und `bytes` der `union data x` teilen sich denselben Speicherplatz:

number				
name				...
bytes[0]	bytes[1]	bytes[2]	bytes[3]	
0x0f	0x10	0x11	0x12	

(Auf einem 32-Bit-Rechner hat `name` dieselbe Größe wie `number`, nämlich 32 Bit, also vier 8-Bit-Speicherzellen; auf einem 64-Bit-Rechner ist `name` doppelt so groß.)

Die Zahl `303108111` lautet hexadezimal `0x1211100f`. In LittleEndian-Darstellung entspricht dies der Byte-Folge `0x0f`, `0x10`, `0x11`, `0x12` (hexadezimal) bzw. 15, 16, 17, 18 (dezimal) in den Speicherzellen.

Die Komponenten des Arrays `bytes` haben jeweils die Größe einer einzelnen Speicherzelle, daher enthalten sie an den Stellen (Indizes) 0 bis 3 genau diese in den Speicherzellen gespeicherten Werte. **Dies erklärt die ausgegebenen Zahlen.**

Bei einem Zugriff auf den Zeiger `name` wird die Zahl `303108111` als Adresse einer Speicherzelle interpretiert; das Programm versucht also, auf den Speicher an der Stelle `303108111` zuzugreifen und den dortigen Speicherinhalt als String auszugeben.

Da es sehr unwahrscheinlich ist, daß dieser Speicherbereich dem Programm zugeordnet ist und sinnvolle Werte enthält, **führt dies normalerweise zu einem Absturz** (Speicherzugriffsfehler).

(Auf einem 64-Bit-Rechner kommen noch einmal weitere vier Bytes mit zufälligen Inhalten zu der Speicheradresse hinzu. Dies führt zu demselben Verhalten.)

- (b) Was würde dasselbe Programm auf einem BigEndian-Rechner ausgeben?

Wie oben gesagt, lautet die Zahl `303108111` hexadezimal `0x1211100f`. In BigEndian-Darstellung entspricht dies der Byte-Folge `0x12`, `0x11`, `0x10`, `0x0f` (hexadezimal) bzw. 18, 17, 16, 15 (dezimal) in den Speicherzellen.

Das Array `bytes` enthält an den Stellen (Indizes) 0 bis 3 genau diese Werte; folglich **lautet die Ausgabe auf einem BigEndian-Rechner:**

18 17 16 15

Darauf folgt wiederum ein Absturz (Speicherzugriffsfehler).

Wir betrachten nun das folgende Programm ([aufgabe-1c.c](#)):

```
#include <stdio.h>

#define POINT 0
#define CIRCLE 1
#define TEXT 2

typedef union
{
    int radius;
    char *text;
} extra_data;

typedef struct graphics_object
{
    int type;
    void (*draw) (struct graphics_object *this);
    int x, y;
    extra_data data;
} graphics_object;

void draw_point (struct graphics_object *this)
{
    printf ("point_at_(%d,%d)\n", this->x, this->y);
}

void draw_circle (struct graphics_object *this)
{
    printf ("circle_at_(%d,%d)_with_radius_%d\n",
        this->x, this->y, this->data.radius);
}

void draw_text (struct graphics_object *this)
{
    printf ("text_at_(%d,%d):_\"%s\"\n",
        this->x, this->y, this->data.text);
}

int main (void)
{
    graphics_object a_point = { POINT, draw_point, 35, 17 };
    graphics_object a_circle = { CIRCLE, draw_circle, 20, 30 };
    a_circle.data.radius = 12;
    graphics_object some_text = { TEXT, draw_text, 42, 23 };
    some_text.data.text = "Hello,_world!";
    graphics_object *g[3] = { &a_point, &a_circle, &some_text };
    for (int i = 0; i < 3; i++)
        g[i]->draw (g[i]);
    return 0;
}
```

Dieses Programm verwaltet verschiedenartige Grafikobjekte in einem Array `graphics_object *g[3]`. Anstatt tatsächlich zu zeichnen, gibt es der Einfachheit halber als Text aus, was ggf. zu sehen wäre.

Der Datentyp `graphics_object` ist ein **struct**, in dem eins der Datenfelder eine **union** (mit weiteren Datenfeldern) ist.

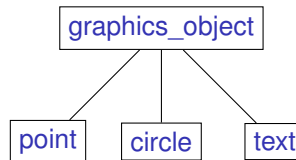
- (c) Beschreiben Sie (in Worten, evtl. mit Skizze) die in diesem Programm realisierte Objekt-Hierarchie. Worum handelt es sich insbesondere bei `void (*draw) (struct graphics_object *this)`? (3 Punkte)
 - (d) Was passiert, wenn man im Hauptprogramm `main()` die an `a_circle` übergebene Funktion `draw_circle` durch `draw_text` ersetzt ([aufgabe-1d.c](#)) und warum? (2 Punkte)
 - (e) Erweitern Sie das Programm so, daß es einen weiteren Grafikobjekttyp `SQUARE` kennt, bei dem man eine Kantenlänge `a` angibt und das beim „Zeichnen“ den Text „square at (x,y) with edge length a“ (mit den korrekten Zahlen anstelle von `x`, `y` und `a`) ausgibt. (5 Punkte)
- (Hinweis für die Klausur: Abgabe auf Datenträger ist erlaubt und erwünscht, aber nicht zwingend.)

Lösung

- (c) **Beschreiben Sie (in Worten, evtl. mit Skizze) die in diesem Programm realisierte Objekt-Hierarchie. Worum handelt es sich insbesondere bei `void (*draw) (struct graphics_object *this)`?**

Es handelt sich um eine **Basisklasse mit drei abgeleiteten Klassen**.

Die Klassen sind im Programm nicht explizit genannt, aber anhand der Bezeichner können wir die Basisklasse als `graphics_object` und die abgeleiteten Klassen als `point`, `circle` und `text` bezeichnen.



Bei `void (*draw) (struct graphics_object *this)` handelt es sich um einen **Zeiger auf eine Funktion**, der hier als **virtuelle Methode** eingesetzt wird.

- (d) **Was passiert, wenn man im Hauptprogramm `main()` die an `a_circle` übergebene Funktion `draw_circle` durch `draw_text` ersetzt (aufgabe-1d.c) und warum?**

Anstatt der Ausgabe

```
point at (35,17)
circle at (20,30) with radius 12
text at (42,23): "Hello, world!"
```

sehen wir nur

```
point at (35,17)
```

gefolgt von einem Absturz (Speicherzugriffsfehler).

Begründung:

Der Austausch der virtuellen Methode bewirkt, daß nun die Funktion `draw_text()` auf ein Objekt vom Typ `a_circle` angewendet wird.

Die Funktion `draw_text()` interpretiert den Inhalt der `extra_data-union data` als einen Zeiger auf Zeichen (String). Da dort vorher die Zahl 12 abgelegt wurde, versucht das Programm nun, auf Speicherzelle Nr. 12 zuzugreifen und den dortigen Speicherinhalt als String auszugeben.

Da es sehr unwahrscheinlich ist, daß dieser Speicherbereich dem Programm zugeordnet ist und sinnvolle Werte enthält, **führt dies normalerweise zu einem Absturz** (Speicherzugriffsfehler).

- (e) **Erweitern Sie das Programm so, daß es einen weiteren Grafikobjekttyp `SQUARE` kennt, bei dem man eine Kantenlänge `a` angibt und das beim „Zeichnen“ den Text „square at (x,y) with edge length a“ (mit den korrekten Zahlen anstelle von `x`, `y` und `a`) ausgibt.**

Folgende Schritte sind notwendig:

- Einführen einer neuen Konstanten für die Typkennung, z. B. `#define SQUARE 3`
- Einführen eines Datenfeldes für die Kantenlänge in die `union extra_data`, z. B. `int edge` (Theoretisch könnte man auch das Feld `radius` für diesen Zweck wiederverwenden; hierdurch verlöre der Code jedoch an Übersichtlichkeit, ohne daß es irgendeinen Gewinn an Rechenzeit und/oder Speicherplatzverbrauch brächte.)
- Schreiben einer Funktion `draw_square()` zur Darstellung eines Quadrats
- Deklaration eines Objekts vom Typ `SQUARE`, z. B. `a_square`
- Vergrößern des Arrays `object[]` von 3 auf 4 Elemente
- Aufnahme eines Zeigers auf `a_square` in das Array
- Verlängern der Schleife von 3 auf 4

Aus den letzten Punkten wird klar, weshalb es oft besser ist, die Größe eines Arrays durch den Compiler selbst ermitteln zu lassen (`object[]` statt `object[4]`) und in der Schleife nicht die Größe, sondern einen zusätzlich gespeicherten `NULL`-Zeiger als Abbruchkriterium zu verwenden.

Eine Umsetzung dieser Schritte als C-Code finden Sie in der Datei `loesung-1e.c`.

Aufgabe 2: Objektorientierte Tier-Datenbank

Diese Übung ist eine Ergänzung zu Aufgabe 4 der Übungen vom 19. 12. 2016 ([hp-uebung-20161219.pdf](#)).

Wir betrachten das korrigierte Programm (wahlweise Ihre eigene Lösung von Teilaufgabe (d) oder eine der Musterlösungen [loesung-4-1.c](#) oder [loesung-4-2.c](#)).

- (e) Schreiben Sie das Programm so um, daß es keine expliziten Typumwandlungen mehr benötigt.
Hinweis: Verwenden Sie **union**. (4 Punkte)
- (f) Schreiben Sie das Programm weiter um, so daß es die Objektinstanzen **duck** und **cow** dynamisch erzeugt.
Hinweis: Verwenden Sie **malloc()** und schreiben Sie Konstruktoren. (4 Punkte)
- (g) Schreiben Sie das Programm weiter um, so daß die Ausgabe nicht mehr direkt im Hauptprogramm erfolgt, sondern stattdessen eine virtuelle Methode **print()** aufgerufen wird.
Hinweis: Verwenden Sie in den Objekten Zeiger auf Funktionen, und initialisieren Sie diese in den Konstruktoren. (4 Punkte)

Lösung

- (e) **Schreiben Sie das Programm so um, daß es keine expliziten Typumwandlungen mehr benötigt.**

Hinweis: Verwenden Sie **union.**

Siehe [loesung-2e.c](#).

Diese Lösung basiert auf [loesung-4-2.c](#), da diese bereits weniger explizite Typumwandlungen enthält als [loesung-4-1.c](#).

Arbeitsschritte:

- Umbenennen des Basistyps **animal** in **base**, damit wir den Bezeichner **animal** für die **union** verwenden können
- Schreiben einer **union animal**, die die drei Klassen **base**, **with_wings** und **with_legs** als Datenfelder enthält
- Umschreiben der Initialisierungen: Zugriff auf Datenfelder erfolgt nun durch z. B. **a[0]—>b.name**. Hierbei ist **b** der Name des **base**-Datenfelds innerhalb der **union animal**.
- Auf gleiche Weise schreiben wir die **if**-Bedingungen innerhalb der **for**-Schleife sowie die Parameter der **printf()**-Aufrufe um.

Explizite Typumwandlungen sind nun nicht mehr nötig.

Nachteil dieser Lösung: Jede Objekt-Variable belegt nun Speicherplatz für die gesamte **union animal**, anstatt nur für die benötigte Variable vom Typ **with_wings** oder **with_legs**. Dies kann zu einer Verschwendung von Speicherplatz führen, auch wenn dies in diesem Beispielprogramm tatsächlich nicht der Fall ist.

- (f) **Schreiben Sie das Programm weiter um, so daß es die Objektinstanzen **duck** und **cow** dynamisch erzeugt.**

Hinweis: Verwenden Sie **malloc() und schreiben Sie Konstruktoren.**

Siehe [loesung-2f.c](#).

Arbeitsschritte:

- Einbinden von **stdlib.h**, um **malloc()** verwenden zu können
- Schreiben der Konstruktoren unter Verwendung von **malloc()** mit der jeweils richtigen Größe
- Umschreiben der Initialisierung des Arrays, so daß anstelle expliziter Zuweisungen die Konstruktoren aufgerufen werden

Mit dieser Version des Programms ist der in Teilaufgabe (f) entstandene Nachteil (mögliche Speicherplatzverschwendung) wieder behoben.

- (g) Schreiben Sie das Programm weiter um, so daß die Ausgabe nicht mehr direkt im Hauptprogramm erfolgt, sondern stattdessen eine virtuelle Methode `print()` aufgerufen wird.

Hinweis: Verwenden Sie in den Objekten Zeiger auf Funktionen, und initialisieren Sie diese in den Konstruktoren.

Siehe `loesung-2g.c`.

Arbeitsschritte:

- Verdopplung des Bezeichners `animal` in der Deklaration der `union animal`; zusätzlich Forward-Deklaration zu Programmbeginn
- Einfügen des virtuellen Methodenfelds `void (*print) (union animal *this)` in den Basistyp
- Einfügen des virtuellen Methodenfelds `void (*print) (union animal *this)` in die abgeleiteten Typen `with_wings` und `with_legs` an derselben Stelle innerhalb der Struktur
- Schreiben der virtuellen Methoden `print_with_wings()` und `print_with_legs()`
- Ergänzen der Zuweisung der virtuellen Methoden in den Konstruktoren
- Ersetzen des Inhalts der `for`-Schleife (`if`-Kette) durch einen Aufruf der virtuellen Methode:
`a[i]—>b.print (a[i])`

Die Typkennung `type` wird nun nicht mehr benötigt und könnte daher entfallen. Alternativ könnte man auch in den virtuellen Methoden (und ggf. an anderen Stellen) mit Hilfe dieses Datenfelds die Konsistenz der Daten prüfen und ggf. das Programm kontrolliert mit einer Fehlermeldung beenden, anstatt abzustürzen.