

Hardwarenahe Programmierung

Musterlösung zu den Übungsaufgaben – 14. Dezember 2023

Aufgabe 1: Dynamisches Bit-Array

Schreiben Sie die folgenden Funktionen zur Verwaltung eines dynamischen Bit-Arrays:

- **void bit_array_init (int n)**
Das Array initialisieren, so daß man *n* Bits darin speichern kann.
Die Array-Größe *n* ist keine Konstante, sondern erst im laufenden Programm bekannt.
Die Bits sollen auf den Anfangswert 0 initialisiert werden.
- **void bit_array_set (int i, int value)**
Das Bit mit dem Index *i* auf den Wert *value* setzen.
Der Index *i* darf von 0 bis *n* – 1 gehen; der Wert *value* darf 1 oder 0 sein.
- **void bit_array_flip (int i)**
Das Bit mit dem Index *i* auf den entgegengesetzten Wert setzen, also auf 1, wenn er vorher 0 ist, bzw. auf 0, wenn er vorher 1 ist.
Der Index *i* darf von 0 bis *n* – 1 gehen.
- **int bit_array_get (int i)**
Den Wert des Bit mit dem Index *i* zurückliefern.
Der Index *i* darf von 0 bis *n* – 1 gehen.
- **void bit_array_resize (int new_n)**
Die Größe des Arrays auf *new_n* Bits ändern.
Dabei soll der Inhalt des Arrays, soweit er in die neue Größe paßt, erhalten bleiben.
Neu hinzukommende Bits sollen auf 0 initialisiert werden.
- **void bit_array_done (void)**
Den vom Array belegten Speicherplatz wieder freigeben.

Bei Bedarf dürfen Sie den Funktionen zusätzliche Parameter mitgeben, beispielsweise um mehrere Arrays parallel verwalten zu können. (In der objektorientierten Programmierung wäre dies der implizite Parameter *this*, der auf die Objekt-Struktur zeigt.)

Die Bits sollen möglichst effizient gespeichert werden, z. B. jeweils 8 Bits in einer `uint8_t`-Variablen.

Die Funktionen sollen möglichst robust sein, d. h. das Programm darf auch bei unsinnigen Parameterwerten nicht abstürzen, sondern soll eine Fehlermeldung ausgeben.

Die folgenden **Hinweise** beschreiben einen möglichen Weg, die Aufgabe zu lösen. Es sieht Ihnen frei, die Aufgabe auch auf andere Weise zu lösen.

- Setzen Sie zunächst voraus, daß das Array die konstante Länge 8 hat, und schreiben Sie zunächst nur die Funktionen `bit_array_set()`, `bit_array_flip()` und `bit_array_get()`.
- Verallgemeinern Sie nun auf eine konstante Länge, bei der es sich um ein Vielfaches von 8 handelt.
- Implementieren Sie nun die Überprüfung auf unsinnige Parameterwerte. Damit können Sie sich gleichzeitig von der Bedingung lösen, daß die Länge des Arrays ein Vielfaches von 8 sein muß.
- Gehen Sie nun von einem statischen zu einem dynamischen Array über, und implementieren sie die Funktionen `bit_array_init()`, `bit_array_done()` und `bit_array_resize()`.

Lösung

Die hier vorgestellte Lösung folgt den Hinweisen.

- **Setzen Sie zunächst voraus, daß das Array die konstante Länge 8 hat, und schreiben Sie zunächst nur die Funktionen `bit_array_set()`, `bit_array_flip()` und `bit_array_get()`.**

Siehe: [loesung-1-1.c](#)

Wir speichern in jedem der acht Bit einer `uint8_t`-Variablen jeweils eine Zahl, die 0 oder 1 sein kann. Dies geschieht durch Setzen bzw. Löschen bzw. Umklappen einzelner Bits in der Variablen.

Das Programm enthält zusätzlich eine Funktion `output()`, mit der man sich den Inhalt des Arrays anzeigen lassen kann, sowie ein Hauptprogramm `main()`, um die Funktionen zu testen.

- **Verallgemeinern Sie nun auf eine konstante Länge, bei der es sich um ein Vielfaches von 8 handelt.**

Siehe: [loesung-1-2.c](#)

In diesem Programm setzen wir die Länge auf konstant `LENGTH` Bits, wobei es sich um eine Präprozessor-Konstante mit dem Wert 32 handelt.

Um `LENGTH` Bits zu speichern, benötigen wir ein Array der Länge `LENGTH / 8` Bytes.

Um auf ein einzelnes Bit zuzugreifen, müssen wir zunächst ermitteln, in welchem der Bytes sich befindet. Außerdem interessieren wir uns für die Nummer des Bits innerhalb des Bytes. Den Array-Index des Bytes erhalten wir, indem wir den Index des Bits durch 8 dividieren. Der bei dieser Division verbleibende Rest ist die Nummer des Bits innerhalb des Bytes.

Diese Rechnungen führen wir in den drei Funktionen `bit_array_set()`, `bit_array_flip()` und `bit_array_get()` durch. (Diese ist eine eher unelegante Code-Verdopplung – hier sogar eine Verdreifachung. Für den Produktiveinsatz lohnt es sich, darüber nachzudenken, wie man diese vermeiden kann, ohne gleichzeitig an Effizienz einzubüßen. Hierfür käme z. B. ein Präprozessor-Makro in Frage. Für die Lösung der Übungsaufgabe wird dies hingegen nicht verlangt.)

- **Implementieren Sie nun die Überprüfung auf unsinnige Parameterwerte. Damit können Sie sich gleichzeitig von der Bedingung lösen, daß die Länge des Arrays ein Vielfaches von 8 sein muß.**

Siehe: [loesung-1-3.c](#)

Um weitere Code-Verdopplungen zu vermeiden, führen wir eine Funktion `check_index()` ein, die alle Prüfungen durchführt.

Wenn die Länge des Arrays kein Vielfaches von 8 ist, wird das letzte Byte nicht vollständig genutzt. Die einzige Schwierigkeit besteht darin, die korrekte Anzahl von Bytes zu ermitteln, nämlich die Länge dividiert durch 8, aber nicht ab-, sondern aufgerundet. Am elegantesten geht dies durch vorherige Addition von 7: `#define BYTES ((LENGTH + 7) / 8)`. Es ist aber auch zulässig, die Anzahl der Bytes mit Hilfe einer `if`-Anweisung zu ermitteln: Länge durch 8 teilen und abrunden; falls die Division nicht glatt aufging, um 1 erhöhen.

- **Gehen Sie nun von einem statischen zu einem dynamischen Array über, und implementieren sie die Funktionen `bit_array_init()`, `bit_array_done()` und `bit_array_resize()`.**

Siehe: [loesung-1-4.c](#). Damit ist die Aufgabe gelöst.

Aus den Präprozessor-Konstanten `LENGTH` und `BYTES` werden nun globale `int`-Variable. Die Funktion `bit_array_init()` berechnet die korrekten Werte für diese Variablen und legt das Array mittels `malloc()` dynamisch an. Eine Größenänderung des Arrays erfolgt mittels `realloc()`, das Freigeben mittels `free()`.

Das Programm setzt Variable, die aktuell nicht verwendet werden, auf den Wert 0 bzw. `NULL`. Dies ermöglicht es der Funktion `check_index()`, auch zu prüfen, ob das Array vorher korrekt mit `bit_array_init()` erzeugt wurde – oder ob es vielleicht schon wieder mit `bit_array_done()` freigegeben wurde.

Aufgabe 2: Objektorientierte Tier-Datenbank

Das unten dargestellte Programm (Datei: [aufgabe-3a.c](#)) soll Daten von Tieren verwalten.

Beim Compilieren erscheinen die folgende Fehlermeldungen:

```
$ gcc -std=c99 -Wall -O aufgabe-2a.c -o aufgabe-2a
aufgabe-2a.c: In function 'main':
aufgabe-2a.c:31: error: 'animal' has no member named 'wings'
aufgabe-2a.c:37: error: 'animal' has no member named 'legs'
```

Der Programmierer nimmt die in Rot dargestellten Ersetzungen vor

(Datei: [aufgabe-3b.c](#)). Daraufhin gelingt das Compilieren, und die Ausgabe des Programms lautet:

```
$ gcc -std=c99 -Wall -O aufgabe-2b.c -o aufgabe-2b
$ ./aufgabe-2b
A duck has 2 legs.
Error in animal: cow
```

- (a) Erklären Sie die o. a. Compiler-Fehlermeldungen. (2 Punkte)
- (b) Wieso verschwinden die Fehlermeldungen nach den o. a. Ersetzungen? (3 Punkte)
- (c) Erklären Sie die Ausgabe des Programms. (5 Punkte)
- (d) Beschreiben Sie – in Worten und/oder als C-Quelltext – einen Weg, das Programm so zu berichtigen, daß es die Eingabedaten ("A duck has 2 wings. A cow has 4 legs.") korrekt speichert und ausgibt. (4 Punkte)

```
#include <stdio.h>

#define ANIMAL 0
#define WITH_WINGS 1
#define WITH_LEGS 2

typedef struct animal
{
    int type;
    char *name;
} animal;

typedef struct with_wings
{
    int wings;
} with_wings;

typedef struct with_legs
{
    int legs;
} with_legs;

int main (void)
{
    animal *a[2];

    animal duck;
    a[0] = &duck;
    a[0]→type = WITH_WINGS;
    a[0]→name = "duck";
    a[0]→wings = 2;  ← ((with_wings *) a[0])→wings = 2;

    animal cow;
    a[1] = &cow;
    a[1]→type = WITH_LEGS;
    a[1]→name = "cow";
    a[1]→legs = 4;  ← ((with_legs *) a[1])→legs = 4;

    for (int i = 0; i < 2; i++)
        if (a[i]→type == WITH_LEGS)
            printf ("A_%s_has_%d_legs.\n", a[i]→name,
                    ((with_legs *) a[i])→legs);
        else if (a[i]→type == WITH_WINGS)
            printf ("A_%s_has_%d_wings.\n", a[i]→name,
                    ((with_wings *) a[i])→wings);
        else
            printf ("Error_in_animal:_%s\n", a[i]→name);

    return 0;
}
```

Lösung

(a) **Erklären Sie die o. a. Compiler-Fehlermeldungen.**

`a[0]` und `a[1]` sind gemäß der Deklaration `animal *a[2]` Zeiger auf Variablen vom Typ `animal` (ein `struct`). Wenn man diesen Zeiger dereferenziert (`->`), erhält man eine `animal`-Variable. Diese enthält keine Datenfelder `wings` bzw. `legs`.

(b) **Wieso verschwinden die Fehlermeldungen nach den o. a. Ersetzungen?**

Durch die *explizite Typumwandlung des Zeigers* erhalten wir einen Zeiger auf eine `with_wings`- bzw. auf eine `with_legs`-Variable. Diese enthalten die Datenfelder `wings` bzw. `legs`.

(c) **Erklären Sie die Ausgabe des Programms.**

Durch die explizite Typumwandlung des Zeigers zeigt `a[0]` auf eine `with_wings`-Variable. Diese enthält nur ein einziges Datenfeld `wings`, das an genau derselben Stelle im Speicher liegt wie `a[0]->type`, also das Datenfeld `type` der `animal`-Variable, auf die der Zeiger `a[0]` zeigt. Durch die Zuweisung der Zahl 2 an `((with_wings *) a[0])->wings` überschreiben wir also `a[0]->type`, so daß das `if` in der `for`-Schleife `a[0]` als `WITH_LEGS` erkennt.

Bei der Ausgabe `A duck has 2 legs.` wird das Datenfeld `((with_legs *) a[0])->legs` als Zahl ausgegeben. Dieses Datenfeld befindet sich in denselben Speicherzellen wie `a[0]->type` und `((with_wings *) a[0])->wings` und hat daher ebenfalls den Wert 2.

Auf die gleiche Weise überschreiben wir durch die Zuweisung der Zahl 4 an `((with_legs *) a[1])->legs` das Datenfeld `a[0]->type`, so daß das `if` in der `for`-Schleife `a[1]` als unbekanntes Tier (Nr. 4) erkennt und `Error in animal: cow` ausgibt.

(d) **Beschreiben Sie – in Worten und/oder als C-Quelltext – einen Weg, das Programm so zu berichtigen, daß es die Eingabedaten ("A duck has 2 wings. A cow has 4 legs.") korrekt speichert und ausgibt.**

Damit die *Vererbung* zwischen den Objekten `animal`, `with_wings` und `with_legs` funktioniert, müssen die abgeleiteten Klassen `with_wings` und `with_legs` alle Datenfelder der Basisklasse `animal` erben. In C geschieht dies explizit; die Datenfelder müssen in den abgeleiteten Klassen neu angegeben werden (siehe `loesung-2d-1.c`):

```
typedef struct animal
{
    int type;
    char *name;
} animal;

typedef struct with_wings
{
    int type;
    char *name;
    int wings;
} with_wings;

typedef struct with_legs
{
    int type;
    char *name;
    int legs;
} with_legs;
```

Zusätzlich ist es notwendig, die Instanzen `duck` und `cow` der abgeleiteten Klassen `with_wings` und `with_legs` auch als solche zu deklarieren, damit für sie genügend Speicher reserviert wird:

```
animal *a[2];

with_wings duck;
a[0] = (animal *) &duck;
a[0]->type = WITH_WINGS;
a[0]->name = "duck";
((with_wings *) a[0])->wings = 2;
```

```

with_legs cow;
a[1] = (animal *) &cow;
a[1]->type = WITH_LEGS;
a[1]->name = "cow";
((with_legs *) a[1])->legs = 4;

```

Wenn man dies vergißt und sie nur als `animal` deklariert, wird auch nur Speicherplatz für (kleinere) `animal`-Variable angelegt. Dadurch kommt es zu Speicherzugriffen außerhalb der deklarierten Variablen, was letztlich zu einem Absturz führt (siehe [loesung-2d-0f.c](#)).

Für die Zuweisung eines Zeigers auf `duck` an `a[0]`, also an einen Zeiger auf `animal` wird eine weitere explizite Typumwandlung notwendig. Entsprechendes gilt für die Zuweisung eines Zeigers auf `cow` an `a[1]`.

Es ist sinnvoll, explizite Typumwandlungen so weit wie möglich zu vermeiden. Es ist einfacher und gleichzeitig sicherer, direkt in die Variablen `duck` und `cow` zu schreiben, anstatt dies über die Zeiger `a[0]` und `a[1]` zu tun (siehe [loesung-2d-2.c](#)):

```

animal *a[2];

with_wings duck;
a[0] = (animal *) &duck;
duck.type = WITH_WINGS;
duck.name = "duck";
duck.wings = 2;

with_legs cow;
a[1] = (animal *) &cow;
cow.type = WITH_LEGS;
cow.name = "cow";
cow.legs = 4;

```

- (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-2d-2.c](#), da diese bereits weniger explizite Typumwandlungen enthält als [loesung-2d-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](#).

- (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](#).