9
Zeiger (Pointer). Dynamischer Speicher

Jörn Loviscach

Versionsstand: 25. September 2013, 18:07

Die nummerierten Felder sind absichtlich leer, zum Ausfüllen beim Ansehen der Videos: http://www.j3L7h.de/videos.html

Bitte hier notieren, was beim Bearbeiten unklar geblieben ist

1 Idee

Jede Variable muss irgendwo im Speicher abgelegt sein. (Oder in einem Register, aber diese Komplikation ignorieren wir mal.) Der Speicher ist in Bytes unterteilt; man kann zählen, bei dem wievielten Byte des Speichers die Daten einer bestimmten Variable beginnen: Das ist die „Adresse“ der Variable:

[1]

In C und C++ kann man mit dem Operator `&` nach der Adresse einer Variablen fragen und diesen in einer Zeigervariablen ablegen. Deren Typ ist dann „Zeiger auf `int`“ usw., angezeigt durch ein Sternchen:

[2]

Demo mit der Speicheransicht im Debugger.

Das Sternchen `*` ist auch ein Operator, um einen Zeiger zu einer Variablen zu machen, die man lesen und schreiben kann:

[3]


Das ist natürlich wieder riskant, weil man so irgendwo im Speicher lesen und schreiben kann:

[4]


Die Adresse 0 hat eine besondere Bedeutung: Kein gespeichertes Objekt darf hier liegen. Mit dem Wert 0 – gerne mit dem Makro `NULL` aus `<stddef.h>` geschrieben – kann man also kennzeichnen, dass ein Zeiger ungültig ist. Auf größeren Rechnern als dem Mikrocontroller führt jeder Zugriff darauf [dereferencing a null pointer] zum Abbruch des Programms führen. In moderneren Sprachen muss man dafür immer ausdrücklich `null` schreiben, niemals 0. Also lieber auch in C und C++ immer `NULL` schreiben.

2 Zeigerarithmetik

Der Name eines Arrays steht in C und C++ für einen Zeiger auf das erste Element:

[5]


Das geht auch umgekehrt: Ein Zeiger kann wie ein Array verwendet werden. Beispiel:

6

Obendrein kann man auch direkt mit Zeigern rechnen („Zeigerarithmetik“): Zu einem Zeiger etwas zu addieren, heißt, ihn um genau so viele Speicherstellen des entsprechenden Typs weiterzustellen – also im Zweifelsfall *nicht* um diese Zahl an Bytes:

7

Nach `int a[] = {1, 2, 3}; int *p = a;` gibt es damit mindestens vier Möglichkeiten, auf `a[2]` zuzugreifen:

8

Das Suchen aller `'a'` und Ersetzen durch `'e'` in einer Zeichenkette kann man nun so schreiben:

9

3 Dynamischer Speicher

Anders als in C99 und in modereren Sprachen muss man für Arrays in C++ und alten Varianten von C im Programmtext eine feste Größe angeben. Dort darf man zum Beispiel nicht dies schreiben:

10

In der IAR Embedded Workbench ist auch in C99 noch der Schalter „Allow VLA“ in Options > C/C++ Compiler > Language nötig.

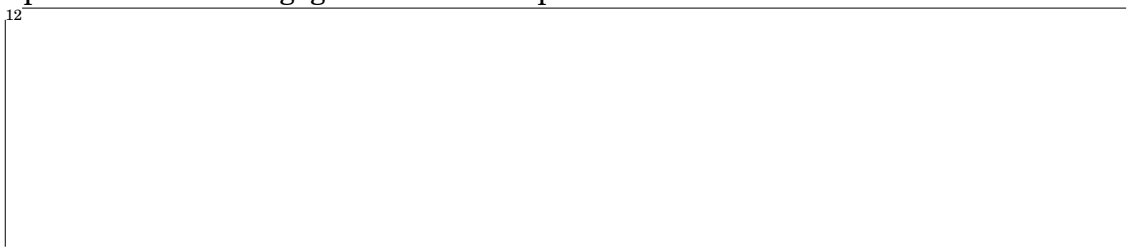
Dass man sich schon vor dem Compilieren auf eine Größe der Arrays festlegen muss, ist natürlich äußerst ungeschickt. Oft weiß man vorher nicht, wie viele Daten man unterbringen muss, und hat aber auch nicht genug Speicher, um einfach auf Verdacht so viel wie nur möglich zu reservieren.

Um trotzdem mit veränderlichen Mengen an Daten umzugehen, kann man C mit der int `<stdlib.h>` deklarierten Funktion `malloc` bitten, einem einen Zeiger auf soundsoviele freie Bytes im Speicher zu geben. Wenn man diese Bytes nicht mehr benötigt, muss man sie mit `free` wieder freigeben, sonst bleiben sie bis zum Programmende blockiert („Speicherleck“). So sieht das aus:

[11]


Der Programmierer ist dafür verantwortlich, dass `free` aufgerufen wird – und zwar nur einmal für denselben Zeiger. Es ist eine gute Praxis, einen Zeiger nach dem Freigeben auf `NULL` zu setzen. Dann führt jeder Zugriff darauf zum Absturz statt zu korrupten Daten. Außerdem garantiert der Standard, dass `free(NULL)` nichts tut.

Bei den üblichen „automatisch“ Variablen sorgt der Compiler selbst dafür, dass ihr Speicher wieder freigegeben wird. Beispiel:

[12]


`malloc` liefert einen Nullzeiger, wenn kein Platz mehr zu haben ist (Demo). Also sollte prüfen, ob das Ergebnis null ist und in diesem Fall zum Beispiel keine neuen Messwerte mehr annehmen o. ä. Auf dem MSP430G2231 gibt es mit 128 Byte derart wenig Platz, dass `malloc` sowieso immer fehlschlägt.

`malloc` gibt den Speicher im Zweifelsfall mit den Werten gefüllt, die vorher mal drinstanden. Die Funktion `calloc` setzt dagegen alle Bits auf null. Mit `realloc` kann man bereits reservierten Speicher vergrößern oder verkleinern.

Mehrdimensionale Arrays, deren Größe beim Compilieren noch nicht feststeht, kann man damit über einen Umweg bauen:

13

Demo in der Speicheransicht des Debuggers.

Für den leichteren Umgang mit Zeigern auf Strukturen kann man statt `(*p).m` übersichtlicher `p->m` schreiben:

14