Programmierkurs: Dynamische Speicherverwaltung

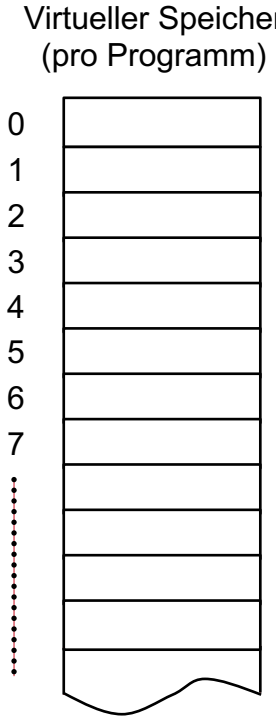
Manfred Hauswirth | Open Distributed Systems | Einführung in die Programmierung, WS 23/24

Rückblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Hello World“: „Lebenswichtiges“, Programablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git

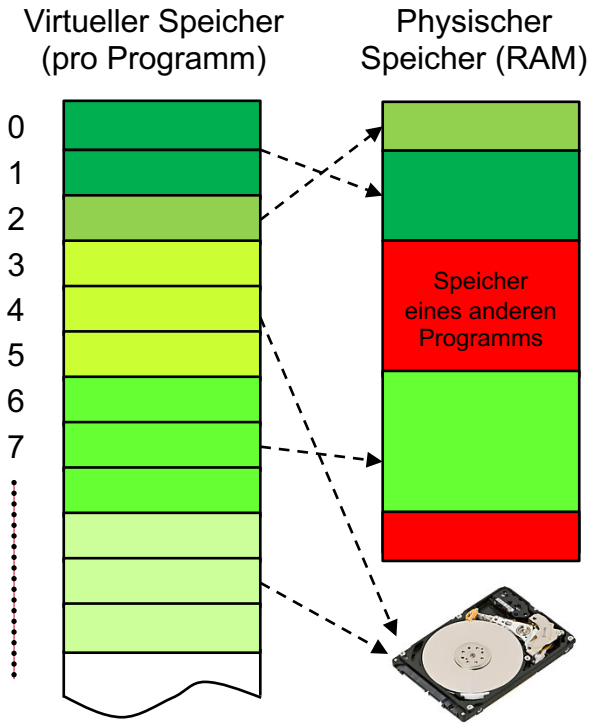
Recap: Speicher – Abstraktion

- Virtueller Speicher: Ein Bytearray
- Programmsicht:
  - Jedes Programm hat seinen eigenen Speicher
  - Es hat eine „unbegrenzte Speichermenge“
  - Der Zugriff auf alle Speicherbereiche ist gleich schnell…



Recap: Speicher – Realität

- Virtueller Speicher: Ein Bytearray
- Realität:
  - Kein unbegrenzter physikalischer Speicher
    - Alle Programme teilen sich den selben physikalischen Speicher
    - Speicher wird durch das Betriebssystem allokiert und verwaltet
    - Viele Anwendungen sind speicherdominiert
  - Es gibt eine Speicherhierarchie
    - Cache, RAM, Festplatte, Netzwerk-Speicher
  - Speicherzugriffsfehler sind besonders schwer zu finden
    - Effekte sind oft weit von der Ursache entfernt



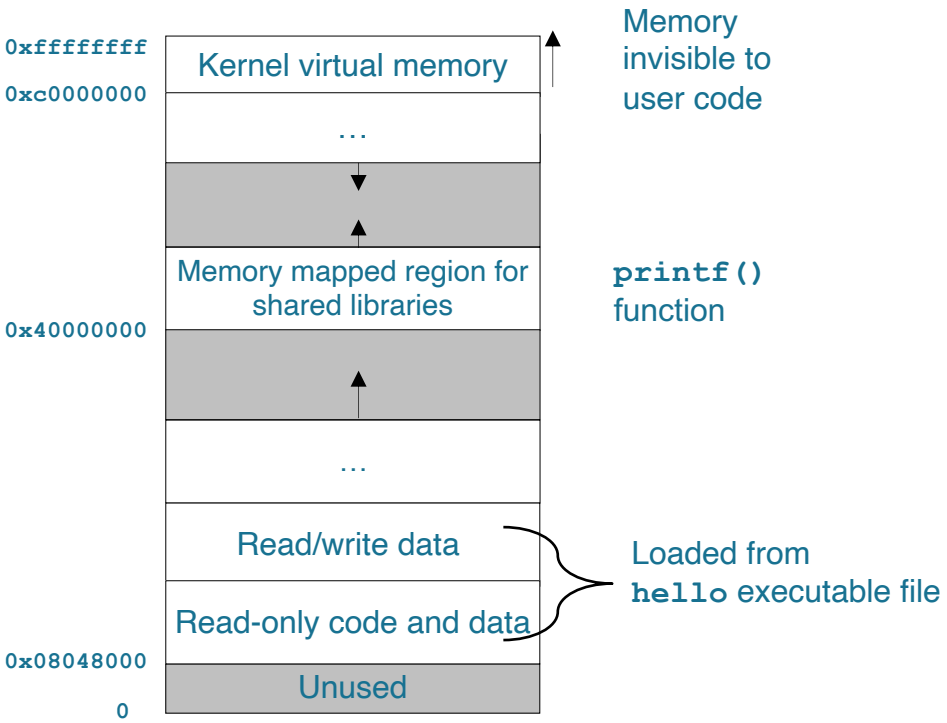
Speicher und C-Programme

Speicher und C-Programme

- Programmkomponenten, die Speicher brauchen
  - Der ausführbare C-Code – das Programm
  - Datenstrukturen innerhalb des Programms
  - C-Bibliotheken und externe Funktionen, z.B. printf
  - …

- Idee: Zuteilung von Speicher nach Bedarf,
  da begrenzte Ressource

Speicher und C-Programme
Physikalischer Speicher



Speicher – implizit

- Speicher nach Bedarf für
  - Variablen
  - Arrays
  - Funktionen

Speicher – implizit

- Speicher nach Bedarf für
  - Variablen
  - Arrays
  - Funktionen

- Problematik
  - Speicher für C-Funktionen unbekannt vor Programmaufruf
  - Warum? Funktionsaufrufreihenfolge unbekannt!

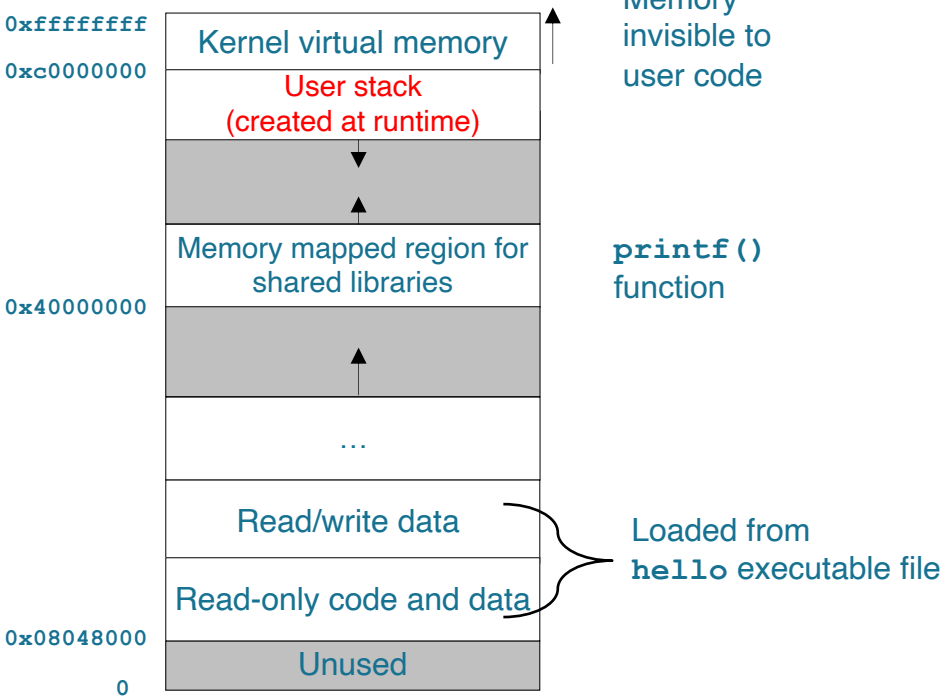
Speicher – implizit

- Speicher nach Bedarf für
  - Variablen
  - Arrays
  - Funktionen

- Problematik
  - Speicher für C-Funktionen unbekannt vor Programmaufruf
  - Warum? Funktionsaufrufreihenfolge unbekannt!

- Idee: Speichern der Variablen, etc. in einer dynamisch wachsenden Datenstruktur: User-Stack (Benutzer-Stapel)
  - mehr zur Datenstruktur Stack später

Speicher und C-Programme
Physikalischer Speicher



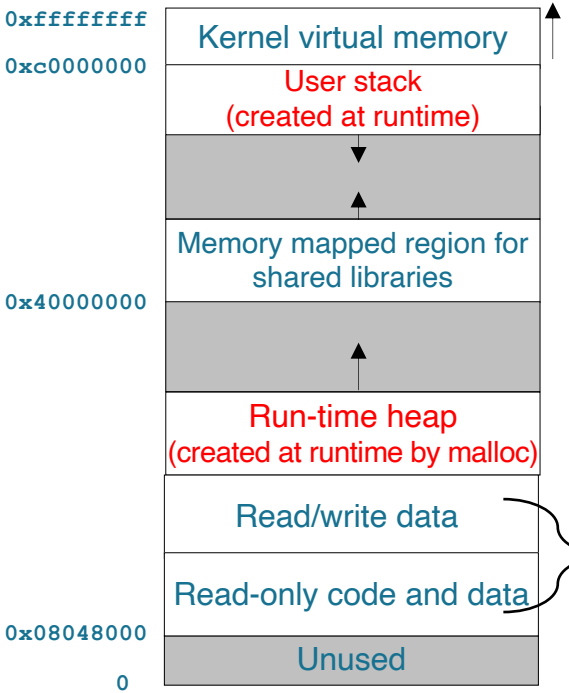
C-Speicher – explizit

- Speicher bei Bedarf
  - explizit für C-Variablen,
    wenn die benötigte Menge Speicher von Parametern abhängig ist

- Problematik
  - Speicherbedarf für C-Variablen unbekannt vor Programaufruf
  - Warum? Anforderungen (Parameter) unbekannt!

- Idee: Speichern der Variablen, etc. in einer weiteren dynamisch wachsenden Datenstruktur: dem Heap (Haufen)
  - mehr zur Datenstruktur Heap später

Speicher und C-Programme
Physikalischer Speicher



Laufzeit vs. Compilezeit

- Compilezeit
  - Während des Compilieren, d.h., Übersetzen des C-Codes in Assemblercode
- Laufzeit
  - Während der Ausführung eines compilierten Programms

- Beispiele:
  - Welche Funktionen existieren: bekannt zur Compilezeit
  - Häufigkeit der Funktionsausführung: in der Regel bekannt zur Laufzeit
  - Mit welchen Funktionsparametern: in der Regel bekannt zur Laufzeit

Dynamische Speicherallokation in C

Dynamische Speicherallokation

- Prinzip: Zuteilung von Speicher nach Bedarf, da begrenzte Ressource
- Zwei Varianten der Speicherverwaltung
  - Implizit: C-Variablen / Arrays

```
int32_t a[100]; /* Die benötigte Menge Speicher
                   steht zur Compilezeit fest */
```

  - Explizit: z.B. für ein Array mit zur Compilezeit unbekannter Länge, welches abhängig von den Eingabedaten ist
    - C-Speicherverwaltung: **`malloc`** und **`free`**

Memory ALLOCation in C

`malloc`

Dynamische Speicherverwaltung: `malloc`

```
#include <stdlib.h>
void* malloc(size_t size)
```

- Rückgabe:
  Zeiger auf Speicherblock der wenigstens die Größe `size` Bytes hat.

- Speicher wird nicht (mit 0) initialisiert
- `void*`: Ein Pointer auf einen nicht festgelegten Typ

Einschub: Fehlerbehandlung

- Motivation:
  - In jeder Funktion können Fehler auftreten
  - Wie werden diese an die aufrufende Funktion zurückgemeldet, um dann behandelt zu werden?
  - z.B.: Speicher ist endlich – `malloc` ist nicht in der Lage, die gewünschte Speichermenge zu allokieren
    ⇒ ein Programm kann einen Fehler haben, obwohl es sich korrekt übersetzen ließ, d.h. es tritt ein **Laufzeitfehler** auf

Einschub: Fehlerbehandlung

- 1. Methode: Nutzen von Rückgabewerten

- Falls ein Fehler in einer Funktion auftritt:
  - Explizite Rückgabe eines bestimmten Wertes
  - Setzen eines Fehlercodes in der globalen Variable `errno`
  - Nutzen einer Hilfsfunktion `perror`. Um diesen Fehlercode und die Fehlermeldung auf der Konsole (via `stderr`) auszugeben

- Gewisse Fehler sollen zum Abbruch des Programms führen

Einschub: Fehlerbehandlung

- 2. Methode: Nutzen der Abbruchfunktion `int exit()`

- Falls ein Fehler in einer Unterfunktion auftritt:
  - Erst Fehleranalyse
  - Dann Abbruch des Programms mittels `exit()`
    - Rückgabewert > 0 => Fehlercode
    - z.B.: `exit(1);`
  - `exit()` bricht das Programm vollständig ab
  - Erfolgreiche Ausführung eines Programms gibt den Wert 0 zurück (Erinnerung: `int main()`)

`malloc`: Fehlerbehandlung

```
int8_t *foo(int32_t n){                    // not to be used!
  int8_t *p;
  // allocate a block of n bytes
  p = (int8_t *) malloc(n);
  return p;
}
```

`malloc`: Fehlerbehandlung

```
int8_t *foo(int32_t n){                    // not to be used!
  int8_t *p;
  // allocate a block of n bytes
  p = (int8_t *) malloc(n);
  return p;
}
```

```
int8_t *foo_with_error_handling(int32_t n){  // to be used
  int8_t *p;
  // allocate a block of n bytes
  p = (int8_t *) malloc(n);
  if (p == NULL){
    perror("malloc failed while allocating n bytes");
    exit(1);
  }
  return p;
}
```

`malloc`-Aufruf erklärt

```
int8_t *foo_with_error_handling(int32_t n){  // to be used
  int8_t *p;
  // allocate a block of n bytes
  p = (int8_t *) malloc(n);
  if (p == NULL){
    perror("malloc failed allocating n bytes");
    exit(1);
  }
  return p;
}
```

- `malloc` liefert einen Pointer auf den angelegten Speicher zurück
- Dieser Speicher muss mit einem Typ versehen werden (casting): hier `(int8_t *)`
- Wenn `malloc` den gewünschten Speicher nicht reservieren konnte, weil z.B. der gesamte Speicher schon verwendet wird, dann wird als Ergebnis ein spezieller Pointer-Wert `NULL` („Nullpointer“) zurückgegeben.

Speicherzugriffsfehler

```
int8_t *foo(int32_t n){                    // not to be used!
  int8_t *p;
  // allocate a block of n bytes
  p = (int8_t *) malloc(n);
  return p;
}
```

- Fehlerbehandlung wichtig, weil sonst Speicherzugriffsfehler möglich sind
- z.B. Zugriff auf eine nicht allokierte Variable über einen Pointer
- Zugriff auf „Nullpointer“ ⇒ „core dump“

malloc

```
#include <stdlib.h>   // notwendig!
void* malloc(size_t size)
```

- Rückgabe (erfolgreich):
  - Zeiger auf Speicherblock, mit wenigstens der Größe `size` Bytes
  - Speicher wird nicht (mit 0) initialisiert
- Rückgabe (nicht erfolgreich): `NULL` und setzen von `errno`.

```
void perror(msg)
```

- Gibt die letzte Systemfehlermeldung auf der Konsole (genauer `stderr`) aus

Rückgabewerte in C (Konvention)

- Funktion hat eigentlich keinen Rückgabewert
  - Status-Code zurückgegeben (d.h. ein „Extra-“ Rückgabewert zur Kontrolle)
  - z.B.: `int32_t add (int32_t *sum, int32_t a, int32_t b)`
  - *Alles OK* => Rückgabe des Wertes 0
  - *Fehler* => Rückgabe eines Wertes != 0

- Funktion hat einen Rückgabewert
  - Rückgabe wird bei Fehlern ein besonderer Wert zugewiesen
  - z.B.: `void *malloc(size_t size)`
  - *Alles OK* => Rückgabe eines Wertes != 0
  - *Fehler* => Rückgabe des Wertes NULL (== 0)

- Zusätzlich: Setzen des Fehlercodes in `errno`.
  - Nutzen einer Hilfsfunktion `perror`. Um diesen Fehlercode und die Fehlermeldung auf der Konsole (via `stderr`) auszugeben

Einschub: Was ist `void` ?

- Motivation:
  - Viele Funktionen geben nichts zurück.

```
void print_hello () {
    printf(„hello world\n“);
}
```

- Problem: C-Syntax verlangt, dass jede Funktion einen Rückgabewert hat
- Idee: Die Funktion gibt einen „Nicht-Wert“ zurück `void`

Einschub: Was ist `void*` ?

- Motivation:
  - Viele Funktionen interessiert es nicht, auf welchen Datentyp ein Pointer zeigt
  - Wichtig ist nur, dass es ein Pointer ist.
- Idee: Benutzen eines Pointer auf „irgendwas“ `void*`

- Beispiel: `void* malloc(size_t size)`

Einschub: Was ist `void*` ?

- Motivation:
  - Viele Funktionen interessiert es nicht, auf welchen Datentyp ein Pointer zeigt
  - Wichtig ist nur, dass es ein Pointer ist.
- Idee: Benutzen eines Pointer auf „irgendwas“ `void*`

- Beispiel: `void* malloc(size_t size)`

- Kann dann in den gewünschten Typ umgewandelt („casting“) werden:
  - `int   *p-int   = (int32_t *) p = malloc(12);`
  - `float *p-float = (float *)   p = malloc(12);`

Dynamische Speicherverwaltung: `free`

- Umkehrfunktion zu `malloc`: Speicher wird nicht mehr benötigt und daher wieder freigegeben

```
#include <stdlib.h>
void free(void *p)
```

- Der Block, auf den `p` zeigt, wird freigegeben
- Parameter `p` muss das Resultat eines vorherigen Aufrufes von `malloc` sein (nur was angelegt wurde, kann wieder freigegeben werden)

- Es gibt in C keine Garbage-Collection („Müllabfuhr“)!
  - Speicher muss explizit freigegeben werden!

Bestimmung der Speichergrößen

- Operator: `sizeof`
- Ermittelt Größe eines Typs oder einer Variablen in Bytes
- Beispiele:

```
int32_t l;
sl = sizeof(l);        // size of variable l
sf = sizeof(float);    // size of type float
```

Malloc / free: Beispiel

```
void foo(int n){
  int32_t i, *p;

  // allocate a block of n integers
  if ((p=(int32_t *) malloc(n*sizeof(int32_t))) == NULL){
    perror("malloc failed when allocating n integers");
    exit(1);
  }
  // assign some values
  for (i = 0; i < n; i++){
    p[i] = i;
  }
  free(p); /* return memory to available memory pool */
}
```

Arrays und Pointer revisited

Arrays und Pointer

- Bekannt:
  Arrays und Pointer werden in C ganz ähnlich behandelt
- Wesentlichster Unterschied:
  - Arrays haben eine feste Dimension
    ⇒ ihnen ist ein fester Speicherort zugeordnet
    ⇒ für die zu speichernden Objekte / Arrayelemente ist Platz reserviert
  - Zeiger/Pointer weisen erst nach Zuweisung oder dynamischer Allokation auf den Speicherort ihrer Objekte
- Arrayvariable ⇒ Adresse des 1. Elements
- Arrayindizes ⇒ Offset im Array

Arrays

- Größe wird bei der Initialisierung festgelegt.
- Zugriff auf Elemente durch Index, z.B. `a[0]`.
- Beispiel:

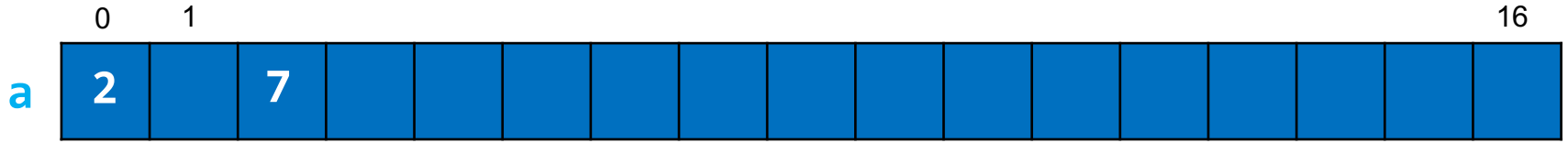
```
uint8_t a[17];
```

Arrays

```
a[0] = 2;
a[2] = 7;
```

Arrays und Pointer

```
a[0] = 2;
a[2] = 7;
```




```
int8_t *b;
b = a;
```

Arrays und Pointer

```
int8_t *b;
b = a;
```


```
int8_t *c;
c = &a[2];  // „&“ bezeichnet die Adresse einer Variablen
```

Arrays und Pointer

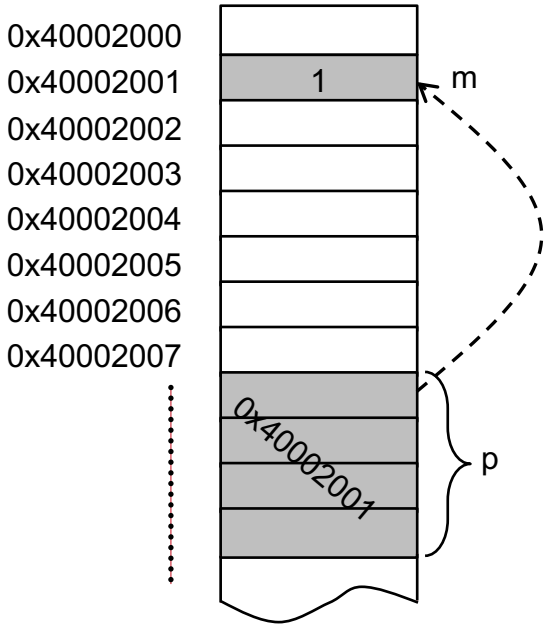
```
*b = 1;    // „*“ Inhalt auf den der Pointer zeigt
```


```
*c = 3;
```

Pointer und Adressen

```
int8_t  m = 1;
int8_t  *p;

p = &m;
```

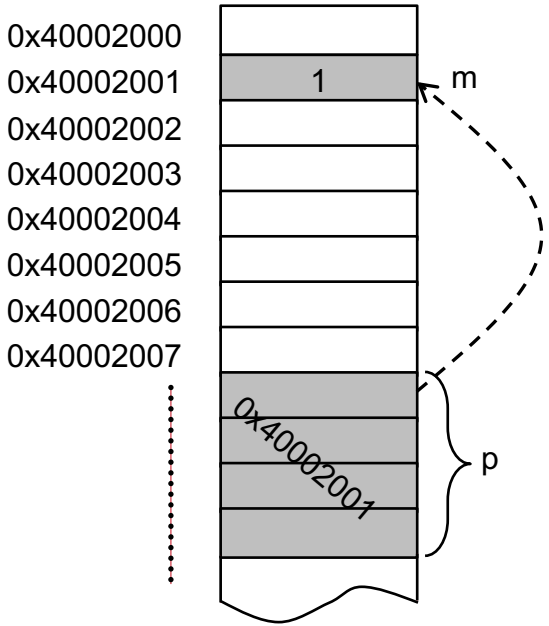


Pointer und Adressen

```
int8_t  m = 1;
int8_t  *p;

p = &m;

*p = 2;
```



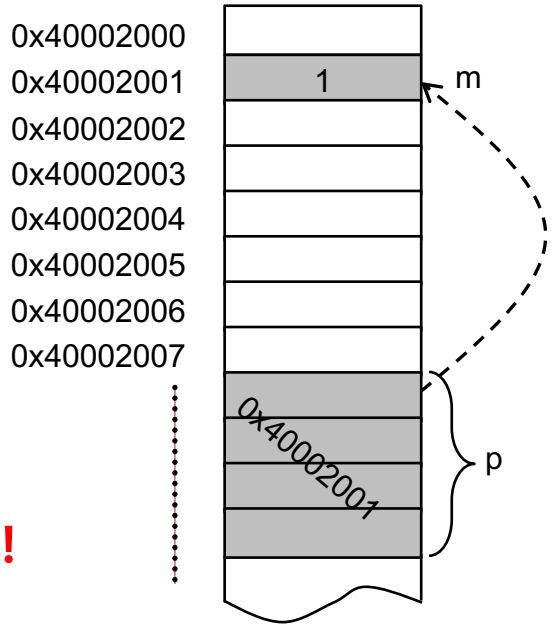
Pointer und Adressen

```
int8_t  m = 1;
int8_t  *p;

p = &m;

*p = 2;

// ACHTUNG falsch!
p = m; // Syntaxfehler!
```



Ausblick

VL 0 „Organisation und Inhalt“: Ablauf der Vorlesung, Termine
VL 1 „Hello World“: „Lebenswichtiges“, Programablauf, Programmierablauf, Kompilierung und Ausführung von Programmen
VL 2 „Die ersten Schritte“: Erstes C-Programm, Elementare C-Strukturen, Datentypen, Operatoren, Schleifen
VL 3 „Kontrollstrukturen & Funktionen“: Syntax, Semantik, bedingte Anweisungen, Blöcke, Sichtbarkeit
VL 4 „Rekursive Funktionen & Bibliotheken“: rekursive Funktionsaufrufe, Modularisierung
VL 5 „Typen“: Einfache und strukturierte Datentypen, Wertebereiche, Typendefinition
VL 6 „Speicher und Adressen“: Speicher, Pointer, Funktionsaufrufe „call by value“ vs. „call by reference“
VL 7 „Speicher und Arrays“: Speicher, Arrays, mehrdimensionale Arrays, Arrays und Pointer
VL 8 „Dynamische Speicherverwaltung“: Speicherallokation, Fehlerbehandlung, Rückgabewerte, Arrays/Pointer/Adressen
VL 9 „Strings, Kanäle, Git“: Strings und Arrays, Zeichensätze, Stringlänge, Ein- und Ausgabe, Arbeiten mit git

Slides für Interessierte

Arrays und Pointer

- Konstante Dimension/Größeangabe von Arrays
```
float f[100];  /* Array mit 100 Elementen */
```
- Variable Dimensionierung von Arrays nur für lokale/automatische Arrayvariable zulässig
```
void fun(int n) {
  float f[n];  /* Array mit n Elementen */
  ...
}
```
- Grund: Arraygröße muss beim Anlegen / bei Speicherzuweisung des Arrays bekannt sein
  - Statisch / global ⇒ Compile-Zeit
  - Automatisch / lokal ⇒ Eintritt in Funktion / Block

Arrays und Pointer

- Variable-Dimensionierung von Arrays wird häufig benötigt
- Lösung ⇒ dynamische Arrayallokation

- Beispiel: `float`-Array dynamisch duplizieren

```
float *fldup(float f[], int n) {
  float *fp;
  int i;

  fp = (float *) malloc(n *sizeof(float));
  for(i = 0; i < n; i++){
    fp[i] = f[i];}
  return(fp);
}
```

Arrays und Pointer

- Arraynamen sind eigentlich Pointer, zeigen auf das erste Element im Array

```
int i, *ip, ia[4] = {11, 22, 33, 44};
ip = ia;
i = *++ip;
```

Arrays und Pointer

- Arraynamen sind eigentlich Pointer, zeigen auf das erste Element im Array

```
int i, *ip, ia[4] = {11, 22, 33, 44};
ip = ia;
i = *++ip;
```

- Ähnlichkeit von Arrays und Zeigern
  $\Rightarrow$ macht die Pointerarithmetik möglich
  $\Rightarrow$ Pointerarithmetik ist mächtig, aber **unübersichtlich!**

Arrays und Pointer

- Arraynamen sind eigentlich Pointer, zeigen auf das erste Element im Array

```
int i, *ip, ia[4] = {11, 22, 33, 44};
ip = ia;
i = *++ip;
```

- Ähnlichkeit von Arrays und Zeigern
  ⇒ macht die Pointerarithmetik möglich
  ⇒ Pointerarithmetik ist mächtig, aber **unübersichtlich!**
- Arrayindizes sind Offsets, == Abstand zum Arrayanfang

```
ia[3]   ⇒   *(&ia[0] + 3)
```

Arrays und Pointer

- Arraynamen sind eigentlich Pointer, zeigen auf das erste Element im Array

```
int i, *ip, ia[4] = {11, 22, 33, 44};
ip = ia;
i = *++ip;
```

- Ähnlichkeit von Arrays und Zeigern
  ⇒ macht die Pointerarithmetik möglich
  ⇒ Pointerarithmetik ist mächtig, aber **unübersichtlich!**
- Arrayindizes sind Offsets, == Abstand zum Arrayanfang

```
ia[3]   ⇒   *(&ia[0] + 3)
```

- Pointerarithmetik ist **mit größter Vorsicht** zu genießen

Pointerarithmetik ☹

```
*(&a[0] + 3) = 4;
```

Pointerarithmetik ist **mit größter Vorsicht** zu genießen !!!