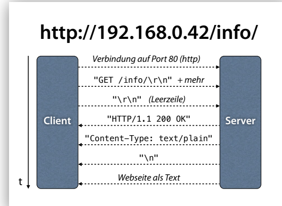
Dynamische Datenstrukturen

Programmieren für Ingenieure
Sommer 2015

Andreas Zeller, Universität des Saarlandes

Server starten

```
// MAC-Adresse
byte mac[] = {
  0xAF, 0xFE, 0xAB, 0xBA, 0xDE, 0xAF
};

// IP-Adresse
IPAddress ip(192, 168, 0, 42);

// Server
EthernetServer server(12345);

void setup() {
  // Serielle Schnittstelle zur Ausgabe der IP nutzen
  Serial.begin(9600);

  // Ethernet-Verbindung und Server starten
  if (Ethernet.begin(mac) == 0)  // DHCP nutzen
  {
     Serial.println("DHCP-Anfrage fehlgeschlagen");
     Ethernet.begin(mac, ip);   // Adresse manuell festlegen
  }

  server.begin();
  Serial.print("server is at ");
  Serial.println(Ethernet.localIP());
}
```



HTML-Tags

```
<!DOCTYPE HTML>
<h1>Ein Titel</h1>
<p>Ein Absatz mit
<em>hervorgehobenem Text</em>
und <br/>
<strong>stark hervorgehobenem
Text</strong>.
</p>
<p>Eine ungeordnete Liste:
<ul>
<li> Ein
Aufz&auml;hlungselement </li>
<li> Noch eins </li>
</ul>
und eine geordnete Liste:
<ol>
<li> Nummer eins </li>
<li> Nummer zwei </li>
</ol>
</p>
```

Ein Titel
Ein Absatz mit *hervorgehobenem Text* und
stark hervorgehobenem Text.
Eine ungeordnete Liste:
- Ein Aufzählungselement
- Noch eins

und eine geordnete Liste:
1. Nummer eins
2. Nummer zwei

LED kontrollieren

Mein Arduino

Analog Eingang 0 hat den Wert 750
Analog Eingang 1 hat den Wert 639
Analog Eingang 2 hat den Wert 527
Analog Eingang 3 hat den Wert 376
Analog Eingang 4 hat den Wert 244
Analog Eingang 5 hat den Wert 225

LED einschalten | ausschalten

Termine

Ping 2015

Heute | Dienstag, 16. Juni | Drucken | Woche | Monat | Terminübersicht

Dienstag, 16. Juni
14:15 Vorlesung "Programmieren für Ingenieure" – Dynamische Datenstrukturen
Dienstag, 23. Juni
14:15 Vorlesung "Programmieren für Ingenieure" – Test und Fehlersuche
Samstag, 27. Juni
10:00 Klausur "Programmieren für Ingenieure"
Mittwoch, 22. Juli
Projektabgabe "Programmieren für Ingenieure"
Samstag, 19. September
10:00 Nachklausur "Programmieren für Ingenieure"

Terminanzeige in der Zeitzone: Berlin

Klausur

Name: ____________________

Matrikelnummer: ____________________

Studiengang: ____________________ seit ____________

Aufgabe		Max. Punkte	Erreichte Punkte
1	Algorithmen	12	
2	Board-Programmierung	20	
3	Datenstrukturen	15	
4	Programmverständnis	8	
5	Wundertüte	5	
Summe		60	

Punkte

Note

Notizen

Ab Donnerstag im Netz (Übungsklausur)

Ihr Projekt

- Teams von 3–4 Teilnehmern bilden
- **Bis 30. Juni**:
  Projektskizze beim Tutor abgeben
- Eine A4-Seite mit
  - Was soll gemacht werden?
  - Warum ist das neu?
  - Warum ist das schwer?
  - Erfolgskriterien

Erfolgskriterien

- **Must-have**: Versprochene Eigenschaften
  "Arduhome ermöglicht die Steuerung einer LED über eine Web-Schnittstelle"
- **May-have**: Mögliche Eigenschaften
  "Ansteuerung mehrerer LEDs; Mobil-Webseite"
- **Must-not-have**: Ausgeschlossene Eigenschaften
  "Sprach- und Gestensteuerung"
- Feedback bis 3. Juli

Abgabe

- **Bis 22. Juli**: *Projekt* einreichen
1. Schaltplan + Quellcode (kommentiert)
2. Aufbauanleitung (für Blöde: 1, 2, 3…)
3. Einsatzanleitung (auch für Blöde)
4. Demo-Video (kommentiert)

Projekt-Beispiele

- Einfache Spiele: Tic-Tac-Toe, …
- Steuerungen: Regelkreis, Tresor, …
- Zeitmessung: Weltzeit, Wecker, …
- Smart Home: Fernsteuerung, Sensoren…
- Und Ihre eigene Idee…
- Bewertung nach *Schwierigkeit* und *Originalität*

Themen heute

- Zeiger
- Freispeicher
- Verbünde
- Suchbäume

Bild: Ohio State

Werte austauschen

- Wir wollen eine Funktion swap(*a*, *b*) schreiben, die die Werte von *a* und *b* vertauscht

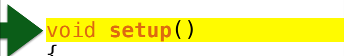
```
int x = 1; int y = 2;
swap(x, y);
// x = 2, y = 1
```

Erster Versuch

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Werte austauschen

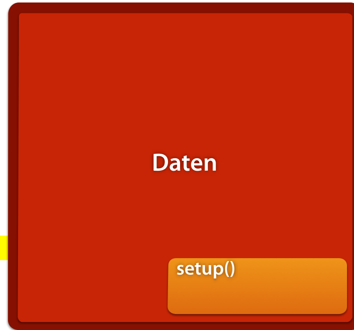
```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```



```
void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```

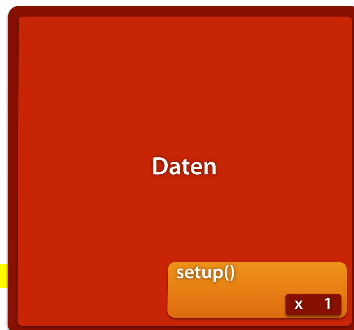
Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```



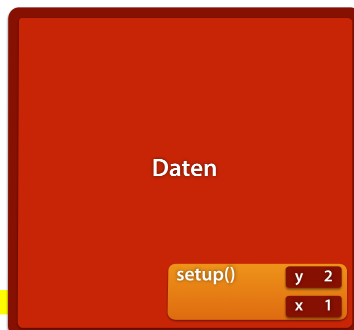
Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```



Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```



Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```

Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```

Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```

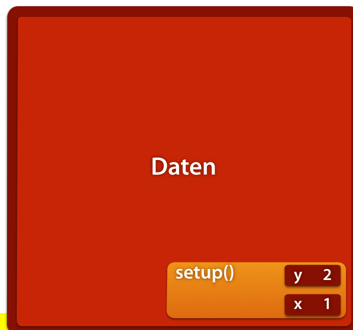
Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```

So weit, so gut – jetzt haben wir a und b vertauscht. Leider wirkt sich das nicht auf den Aufrufer aus.

Werte austauschen

```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(x, y);
}
```



In setup() nämlich bleiben alle Werte so, wie sie waren :–(

Werte austauschen

- Eine Funktion kann die lokalen Variablen einer anderen Funktion nicht verändern

Große Programme

```
void doSomething(EthernetClient c) {
  if (c.available()) { ... }
}

void loop() {
  EthernetClient client = server.available();
  doSomething(client);
}
```

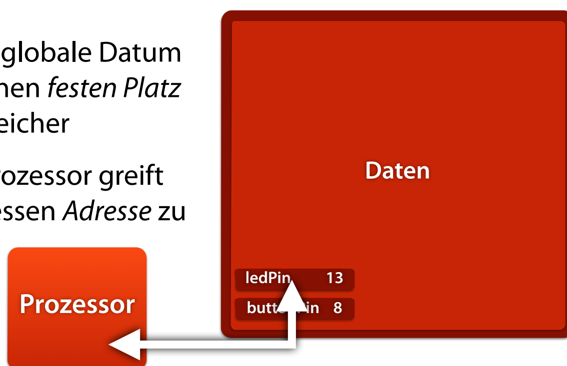
- *c* ist eine *Kopie* von *client*
- Folge: zwei Verbindungen!
- Wie können wir doSomething() auf dem *client* aus loop() arbeiten lassen?

Zeiger

- Eine Funktion kann die lokalen Variablen einer anderen Funktion nicht verändern…
- …es sei denn, sie benutzt einen *Zeiger*

Speicherorte

- Jedes globale Datum hat einen *festen Platz* im Speicher
- Der Prozessor greift auf dessen *Adresse* zu



Speicherorte

- Eine *Adresse* sagt dem Prozessor, wo der Wert zu finden ist
- Eine *Zahl* ähnlich einer Hausnummer
- Die Adresse von *ledPin* könnte etwa **0x0010a024** sein

0x0010a0024

- Hexadezimalzahl = Zahl zur Basis 16
- In C mit Präfix 0x geschrieben
- Ziffern: 0–9 wie bekannt, zudem A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
- 0xA3 ist also $10 \cdot 16^1 + 3 = 163$
- 0xAFFE ist $10 \cdot 16^3 + 15 \cdot 16^2 + 15 \cdot 16^1 + 14 = 45054$

17432612

- Hexadezimalzahl = Zahl zur Basis 16
- In C mit Präfix 0x geschrieben
- Ziffern: 0–9 wie bekannt, zudem A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
- 0xA3 ist also $10 \cdot 16^1 + 3 = 163$
- 0xAFFE ist $10 \cdot 16^3 + 15 \cdot 16^2 + 15 \cdot 16^1 + 14 = 45054$

Adressen

- In C liefert *&x* die *Adresse* von *x:*
- &ledPin = 0x0010a024
- &buttonPin = 0x0010a020

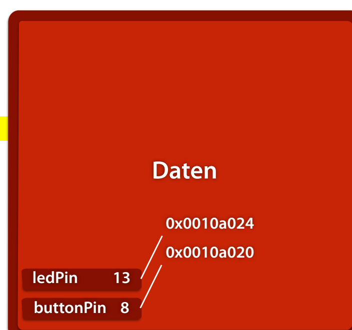
Zeiger

- Ein *Zeiger* ist eine Variable, die die *Adresse* einer Variablen speichert
- Man sagt: Der Zeiger "zeigt" auf die Variable
- Ein Zeiger mit Namen *p*, der auf einen Typ T zeigt, wird als *T *p* deklariert:

```
int *p1 = &ledPin;
```

Zeiger

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
 int *p1 = &ledPin;
}
```

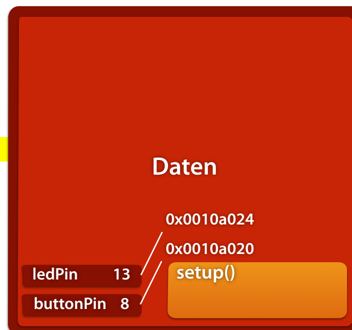


Zeiger

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
  int *p1 = &ledPin;
}
```

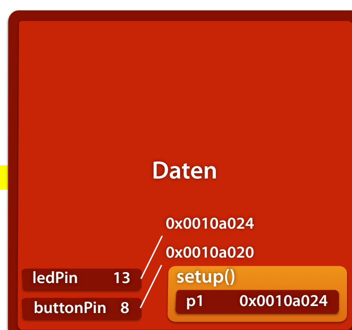
Zeiger

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
  int *p1 = &ledPin;
}
```



Zeiger

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
  int *p1 = &ledPin;
}
```



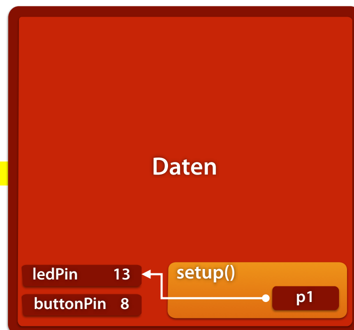
Zeiger

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
  int *p1 = &ledPin;
}
```

In Wahrheit brauchen wir aber die genauen Adressen gar nicht; es genügt zu wissen, dass p1 auf ledPin zeigt – also die Adresse von ledPin enthält.

Zeiger

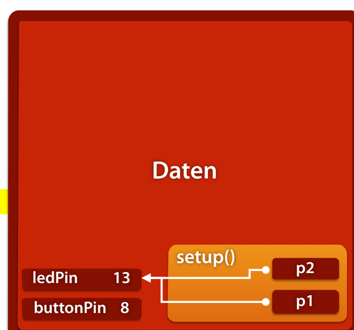
```
int ledPin = 13;
int buttonPin = 8;

void setup() {
  int *p1 = &ledPin;
  int *p2 = p1;
}
```



In Wahrheit brauchen wir aber die genauen Adressen gar nicht; es genügt zu wissen, dass p1 auf ledPin zeigt – also die Adresse von ledPin enthält.

Zeiger

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
  int *p1 = &ledPin;
  int *p2 = p1;
}
```



Fügen wir noch einen weiteren Zeiger hinzu

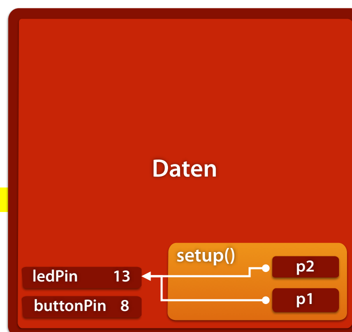
Dereferenzieren

- Der Ausdruck **p* steht für die Variable, auf die *p* zeigt (= die Variable an Adresse *p*)
- Man sagt: Der Zeiger wird *dereferenziert*
- **p* kann wie eine Variable benutzt werden

```
int *p1 = &ledPin;
int x = *p1;   // x = ledPin
*p1 = 25;      // ledPin = 25
```

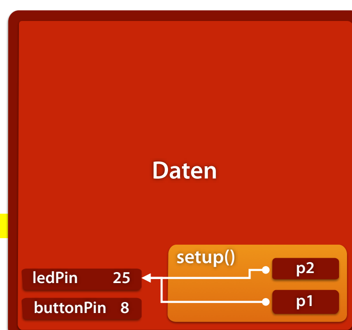
Dereferenzieren

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
 int *p1 = &ledPin;
 int *p2 = p1;
 *p2 = 25;
 p1 = &buttonPin;
 *p1 = *p2;
}
```



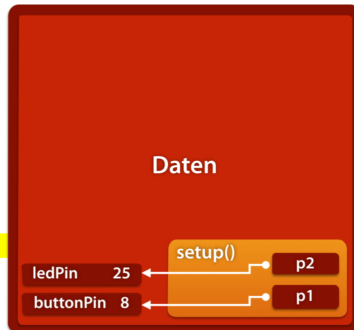
Dereferenzieren

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
 int *p1 = &ledPin;
 int *p2 = p1;
 *p2 = 25;
 p1 = &buttonPin;
 *p1 = *p2;
}
```



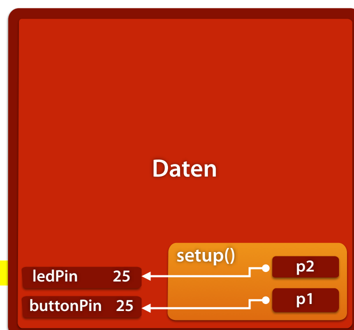
Dereferenzieren

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
 int *p1 = &ledPin;
 int *p2 = p1;
 *p2 = 25;
 p1 = &buttonPin;
 *p1 = *p2;
}
```



Dereferenzieren

```
int ledPin = 13;
int buttonPin = 8;

void setup() {
 int *p1 = &ledPin;
 int *p2 = p1;
 *p2 = 25;
 p1 = &buttonPin;
 *p1 = *p2;
}
```



Werte austauschen

- Wir wollen eine Funktion swap(*a*, *b*) schreiben, die die Werte von *a* und *b* vertauscht
- Wir übergeben die *Adressen* von *a* und *b*

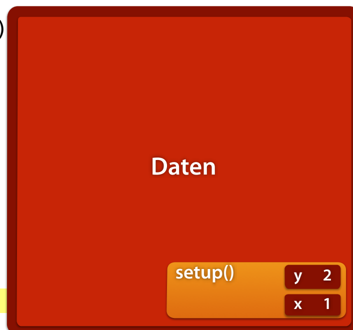
```
int x = 1; int y = 2;
swap(&x, &y);
// x = 2, y = 1
```

Tauschen mit Zeigern

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

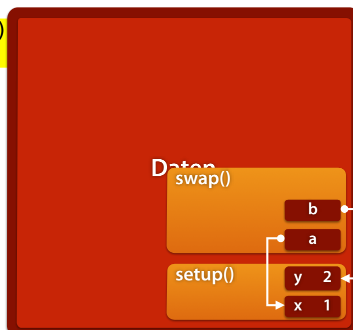
Werte austauschen

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```



Werte austauschen

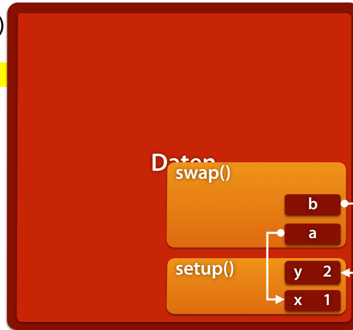
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```



a zeigt nun auf x, b auf y. *a ist der Wert, der an der Adresse von a steht – also der Wert von x.

Werte austauschen

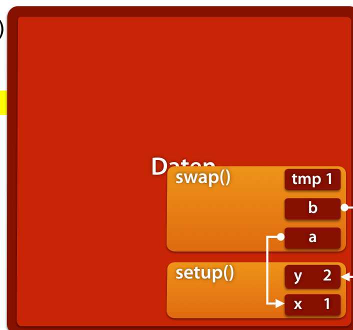
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```



a zeigt nun auf x, b auf y. *a ist der Wert, der an der Adresse von a steht – also der Wert von x.

Werte austauschen

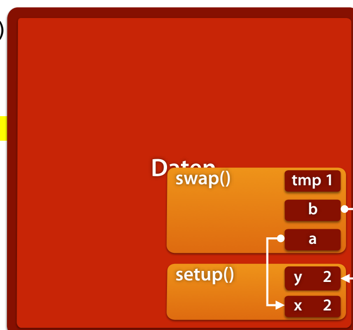
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```



Wir können nun *a einen neuen Wert zuweisen – und verändern damit x

Werte austauschen

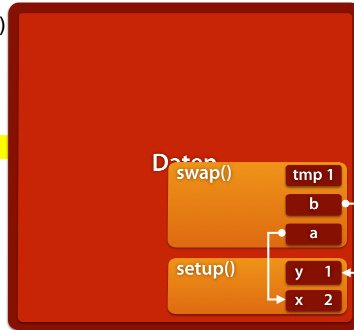
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```



*b verändert analog die Variable y.

Werte austauschen

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```



Werte austauschen

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

void setup()
{
   int x = 1;
   int y = 2;
   swap(&x, &y);
}
```

Fertig!

… und am Ende sind x und y (wie geplant) vertauscht!

Große Programme

```
void doSomething(EthernetClient *client) {
  if ((*client).available()) {
      // Daten verfügbar
  }
}

void loop() {
  // Anfrage bearbeiten
  EthernetClient client = server.available();
  doSomething(&client);
}
```

- Bessere Alternative ohne Kopieren

Spaß mit Zeigern

Mit Zeigern kann man

- eine Funktion Variablen *verändern* lassen
- *Freispeicher* verwalten
- auf *Felder* zugreifen
- *komplexe Datenstrukturen* aufbauen

Dynamische Datenstrukturen

- In C muss ich die Größe eines Feldes bereits *zur Übersetzungszeit* ("statisch") angeben
- Was aber, wenn ich diese Größe erst zur *Laufzeit* ("dynamisch") kenne?

- Ich lade eine Karte (z.B. aus einer Datei)
- Die Karte enthält die Listen der Städte und Straßen
- Die Liste kann je nach Karte unterschiedlich groß sein

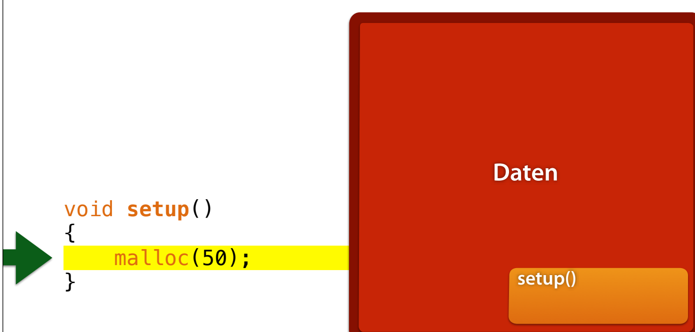
Freispeicher

- *Freispeicher* ist Speicher, der *erst zur Laufzeit* angefordert wird
- Ich kann die Größe frei festlegen
- Ich erhalte einen *Zeiger* auf den Speicher
- Wichtigster Einsatz von Zeigern!

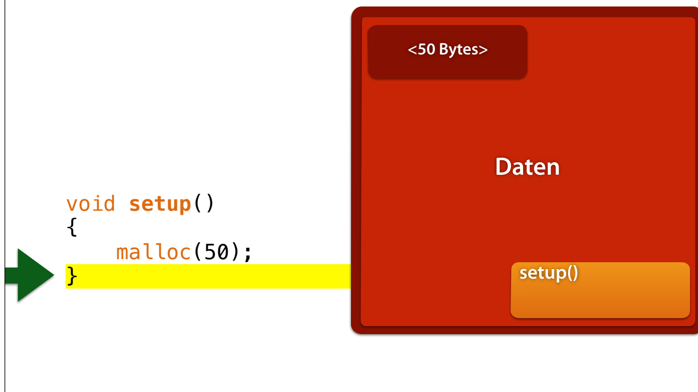
Freispeicher anlegen

- Die C-Funktion malloc(*n*) erzeugt einen Speicherbereich, bestehend aus *n* Bytes

Freispeicher anlegen



Freispeicher anlegen



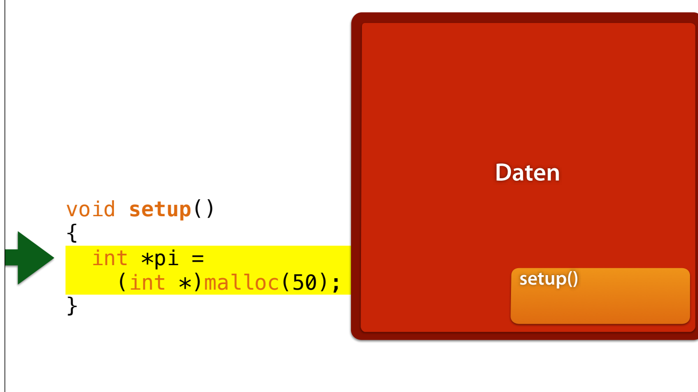
Zugriff Freispeicher

- malloc() gibt einen *Zeiger* auf den Speicherbereich zurück
- Dieser Zeiger muss in den gewünschten Typ *umgewandelt* werden

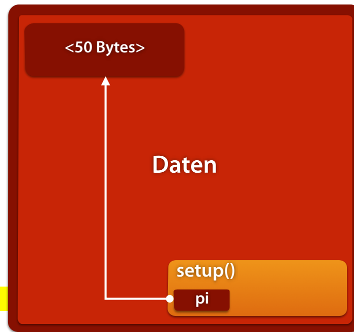
```
int *pi =
  (int *)malloc(50);
```

Typumwandlung

Zugriff Freispeicher



Zugriff Freispeicher




```
void setup()
{
  int *pi =
    (int *)malloc(50);
}
```

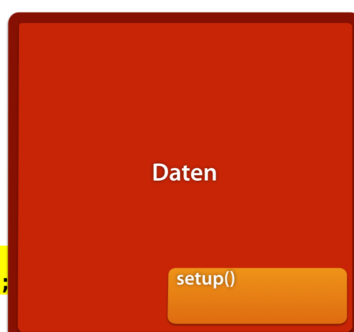
Größe Freispeicher

- Die Funktion sizeof(*x*) gibt die Größe von *x* zurück (in Bytes)
- *x* ist ein Typ oder eine Variable

```
int *pi =
  (int *)malloc(sizeof(int));
```

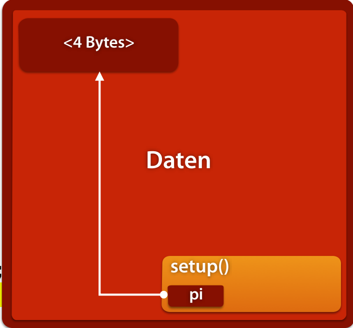
Typumwandlung   Größe

Größe Freispeicher



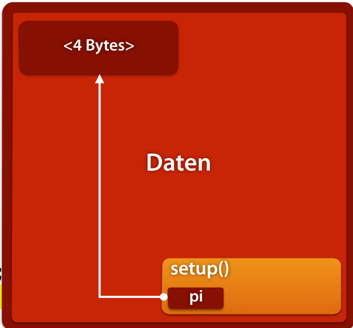

```
void setup()
{
  int *pi = (int *)
   malloc(sizeof(int));
}
```

Größe Freispeicher



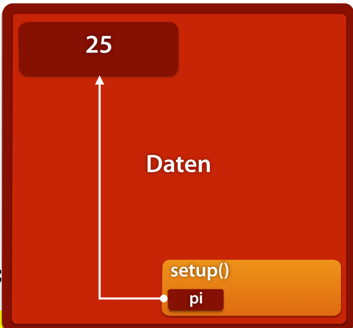

```
void setup()
{
  int *pi = (int *)
   malloc(sizeof(int));
}
```

Größe Freispeicher




```
void setup()
{
  int *pi = (int *)
   malloc(sizeof(int));
  *pi = 25;
}
```

Größe Freispeicher




```
void setup()
{
  int *pi = (int *)
   malloc(sizeof(int));
  *pi = 25;
}
```

Freispeicher für Felder

- Über den Freispeicher kann ich auch genügend Speicher für ein *Feld* anfordern
- Beispiel: 100 int-Elemente

```
int *pi =
  (int *)malloc(sizeof(int) * 100);
```

Freispeicher für Felder

- Indem ich *den Zeiger als Feldname benutze,* kann ich wie gewohnt auf die Feldelemente zugreifen:

```
int *pi =
  (int *)malloc(sizeof(int) * 100);

pi[0] = 2;
pi[1] = 3;
pi[2] = pi[0] * pi[1];
```

Beispiel: Feld einlesen

- Wir lesen erst *n,* und dann *n* Werte ein

```
int n = get_number_of_values();
int *values =
 (int *)malloc(sizeof(int) * n);

for (int i = 0; i < n; i++)
  values[i] = get_value(i + 1, n);
```

Beispiel: Feld einlesen

```
int n = get_number_of_values();
int *values =
 (int *)malloc(sizeof(int) * n);

for (int i = 0; i < n; i++)
  values[i] = get_value(i + 1, n);
```

```
Number of values: 3
Value 1/3: 2
Value 2/3: 8
Value 3/3: 11
```

Demo

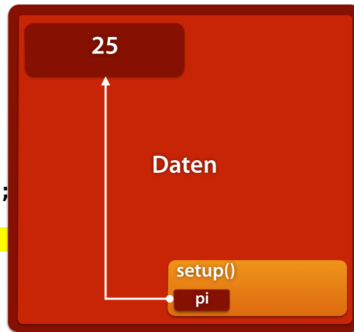
Freispeicher freigeben

- Wenn ich den Speicher nicht mehr benötige, muss ich ihn mit free() freigeben

```
int *pi =
  (int *)malloc(sizeof(int) * 100);

// ...Zugriff auf pi...

free(pi);
```

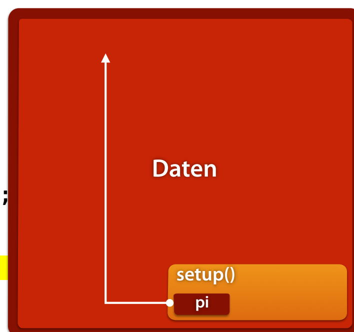
Freispeicher freigeben

```
void setup()
{
  int *pi = (int *)
   malloc(sizeof(int));
  *pi = 25;
  free(pi);
}
```



Freispeicher freigeben

```
void setup()
{
  int *pi = (int *)
   malloc(sizeof(int));
  *pi = 25;
  free(pi);
}
```



Freispeicher freigeben

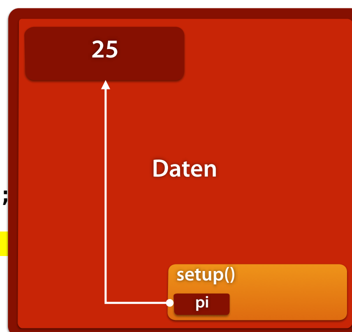
- Freigegebener Freispeicher darf *nicht weiter benutzt werden!*
- Dummerweise merkt man nicht sofort, ob Speicher bereits freigegeben wurde…
- Wird Speicher *2x freigegeben*, kommt es irgendwann zum Absturz (*Zeitbombe*).

Freispeicher freigeben

- Wird Freispeicher nach der Benutzung *nicht* freigegeben, bleibt er erhalten
- Ist aber kein Zugriff mehr möglich, kann der Speicher nicht mehr freigegeben werden (*Speicherleck*)

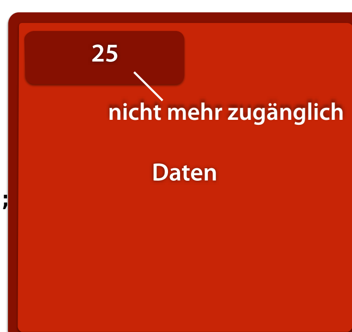
Speicherleck

```
void setup()
{
  int *pi = (int *)
   malloc(sizeof(int));
  *pi = 25;
}
```



Speicherleck

```
void setup()
{
  int *pi = (int *)
   malloc(sizeof(int));
  *pi = 25;
}
```



Speicherleck

- Auf Dauer wird so der verfügbare Speicher immer weniger…
- bis er *voll* ist und kein weiterer Freispeicher mehr zur Verfügung steht.

25	<38 Bytes>
<8 Bytes>	<16 Bytes>
<16 Bytes>	<12 Bytes>
<16 Bytes>	<30 Bytes>
<20 Bytes>	<8 Bytes>

THIS DAY IN TECH | 20th century | Britain | Computers | Health and Medicine

Oct. 26, 1992: Software Glitch Cripples Ambulance Service

BY TONY LONG 10.26.09 | 12:00 AM | PERMALINK

1992: A software error causes London's brand new computer-aided ambulance-dispatch system to fail. The ensuing snafu is blamed for anywhere between 30 and 45 deaths.

CAD, as it was known, replaced the London Ambulance Services human dispatchers who had handled things efficiently, if a bit slowly, for years. The new system was supposed to reduce the average emergency-response time to three minutes, as mandated by a recent government decree. That was beyond the capabilities of the human dispatchers, whose cumbersome methods — involving a lot of paper-

Speicherlecks können zu ernsthaften Problemen führen.

http://www.wired.com/2009/10/1026london-ambulance-computer-meltdown

Three primary flaws hampered things from the start: It didn't function well when given incomplete data, the user interface was problematical and — most damning — there was a memory leak in a portion of the code.

The result in those first hours was complete chaos on the streets. As the system crashed, dispatchers failed to send ambulances to some locations while dispatching multiple units to others.
It got worse as people expecting an ambulance and not getting one began to call back, flooding the already-overwhelmed service. In one case, a person who died while awaiting help had already been removed by the mortician before the ambulance arrived.

Voller Speicher

- Steht kein Speicher mehr zur Verfügung, liefert malloc() einen *besonderen Zeigerwert* namens NULL zurück

```
int *pi = (int *)malloc(10000000000);
if (pi == NULL) {
  Serial.println("Speicher voll");
  abort();
}
```

NULL-Zeiger

- NULL wird in Programmen grundsätzlich als Wert für “ungültige Adresse” benutzt
- Wird NULL dereferenziert, führt dies zum sofortigen Absturz (hoffentlich)

```
int *pi = NULL;
*pi = 25;
```

Freispeicher

1. Du sollst nicht zu viel Speicher anfordern!
2. Du sollst nicht zu wenig Speicher anfordern!
3. Du sollst angeforderten Speicher wieder freigeben!
4. Niemals sollst Du auf freigegebenen Speicher zugreifen!
5. Du sollst Speicher nicht doppelt freigeben!

Und wer es trotzdem tut, der möge in der Hölle schmoren für jetzt und alle Zeit. Oder sein Programm möge ihm verfaulen.

Freispeicher in C++

- In C++ lässt sich der Freispeicher einfacher ansprechen:

C

```
int *pi = (int *)malloc(sizeof(int));
free(pi);
```

C++

```
int *pi = new int;
delete pi;
```

Freispeicher in C++

- In C++ lässt sich der Freispeicher einfacher ansprechen:

C

```
int *pi = (int *)malloc(sizeof(int) * 10);
free(pi);
```

C++

```
int *pi = new int[10];
delete[] pi;
```

Wichtig: zu „malloc“ gehört „free“, zu „new“ „delete“, und zu „new[]“ „delete[]“. Durcheinanderbringen hat schreckliche Folgen.

Zeiger und Felder

- In C und C++ steht jeder *Feldname* für die Adresse, an der das Feld beginnt

```
char s[100] = "Hall";

char *pc = &s[0]; // 1. Element
*pc = 'B';
```

ist dasselbe wie

```
char *pc = s;
*pc = 'B';
```

Zeigerarithmetik

- Ist p ein Zeiger auf ein Feldelement, dann zeigt $p + 1$ auf das *nächste* Element.

```
char s[100] = "Hall";

char *pc = s; // 1. Element
*pc = 'B';
pc = pc + 1;  // 2. Element
*pc = 'i';
```

Zeigerarithmetik

- Ist p ein Zeiger auf ein Feldelement, dann zeigt $p + 1$ auf das *nächste* Element.

```
char s[100] = "Hall";

char *pc = s; // 1. Element
while (*pc++ != '\0');
return pc - s; // Länge von s
```

Zeigerarithmetik

- $p[i]$ kann auch als $*(p + i)$ geschrieben werden

```
char s[100] = "Hall";
char *pc = s; // 1. Element

pc[0] = 'B';  ≡  *pc = 'B';
pc[1] = 'i';     *(pc + 1) = 'i';

              ≡  *(0 + pc) = 'B';
                 *(1 + pc) = 'i';

              ≡  0[pc] = 'B';
                 1[pc] = 'i';
```

…und da + kommutativ ist, ist pc[1] übrigens auch dasselbe wie *(1 + pc) und somit 1[pc]. Wenn Sie die Leser Ihrer Programme abgrundtief verwirren wollen, hätten Sie hier eine Vorlage.

Obfuscated C

```
main(_,l)char**l;{6*putchar(--_
%20?_+_/21&56>_?
strchr(1[l],_^"pt`u}
rxf~c{wk~zyHHOJ]QULGQ[Z"[_/2])?
111:46:32:10)^_&&main(2+_,l);}
```

20th International Obfuscated C Code Contest (2011) http://www.ioccc.org/years.html#2011 (konno)
Das Programm passt in eine Zeile und gibt eine Tastatur aus, bei der die als Argument übergebenen Buchstaben hervorgehoben sind (z.B. “a.out qwerty”).

Übungsklausur

Name: ____________________

Matrikelnummer: ____________________

Studiengang: ____________________ seit __________

Aufgabe		Max. Punkte	Erreichte Punkte
1	Algorithmen	12	
2	Board-Programmierung	20	
3	Datenstrukturen	15	
4	Programmverständnis	8	
5	Wundertüte	5	
Summe		60	

Punkte

Note

Notizen

Kommt das in der Klausur vor? Sie bekommen einige Rätsel, aber so schwere nun auch wieder nicht…

Verbünde

Im wirklichen Leben werden Daten oft *aus anderen Daten* zusammengesetzt:

- *Brüche* bestehen aus *Zähler* und *Nenner*
- *Maße* bestehen aus *Breite, Höhe, Tiefe*
- *Koordinaten* bestehen aus *x, y, z-Werten*

Verbünde

- In C können einzelne Daten zu einem *Verbund* ("struct") zusammengefasst werden
- Beispiel: *Komplexe Zahlen*

Typ-Definition

```
struct Complex {
  double real;
  double imag;
};
```

Variablen-Initialisierung

```
struct Complex c = {
  3.0,  // real
  4.0   // imag
};
```

Verbünde

- Auf die Elemente eines Verbundes kann ich mit *variable.element* zugreifen

Variablen-Initialisierung

```
struct Complex c = {
  3.0,  // real
  4.0   // imag
};
```

Benutzung

```
void print_complex
  (struct Complex c)
{
  Serial.println(c.real);
  Serial.print("+");
  Serial.println(c.imag);
  Serial.print("i");
}
```

Das Schreiben von "struct complex" ist aber eher umständlich auf Dauer; deshalb gibt es eine abkürzende Schreibweise.

Typ-Definitionen

- Mit *typedef typname alias* wird *alias* zu einem alternativen (kürzeren) Namen für *typname*

Typdefinition

```
struct Complex {
  double real;
  double imag;
};

typedef struct Complex
  complex;
```

Benutzung

```
void print_complex
  (complex c)
{
  Serial.println(c.real);
  Serial.print("+");
  Serial.println(c.imag);
  Serial.print("i");
}
```

Verbünde

- Ein Verbund kann wie jede andere Variable als *Parameter* und *Rückgabewert* dienen.

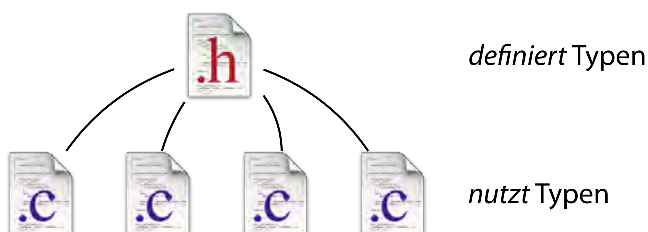
```
complex make_complex(double real, double imag)
{
    complex c;
    c.real = real;
    c.imag = imag;
    return c;
};

complex complex_sum(complex c1, complex c2)
{
    return make_complex(c1.real + c2.real,
                        c1.imag + c2.imag);
};
```

Header-Dateien

- Selbstdefinierte Typen (wie „Complex“) werden in vielen Programmteilen benutzt
- Ziel: Typ 1x definieren, beliebig oft nutzen

Header-Dateien

- Eine *Header-Datei* definiert Typen und Funktionen, die von mehreren Programmteilen genutzt werden



definiert Typen

nutzt Typen

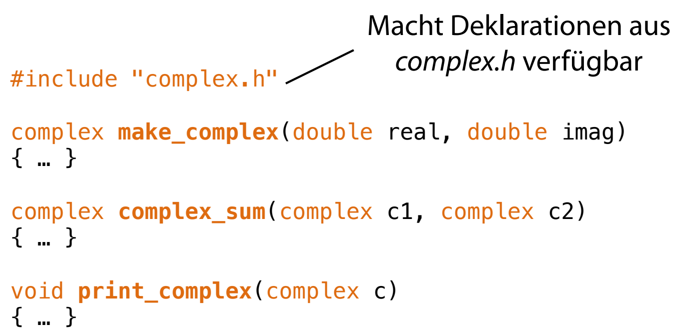
complex.h

```c
struct Complex {
  double real;
  double imag;
};

typedef struct Complex complex;

complex make_complex(double real, double imag);
complex complex_sum(complex c1, complex c2);
void print_complex(complex c);
```

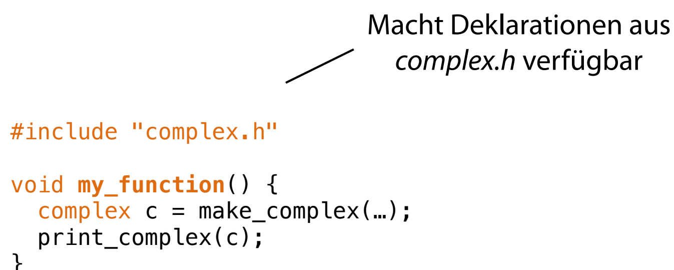
Funktions-Deklarationen

Eine Funktions–Deklaration gibt Namen, Argumente und Rückgabetyp einer Funktion an; der Funktionskörper, die eigentliche Implementierung, entfällt jedoch.

complex.c



user.c



Demo

Datenbanken

- Häufig fassen Verbünde Daten zu einem *Vorgang* oder einer *Person* zusammen.
- Beispiel: Personendaten

Datenbanken

- Häufig fassen Verbünde Daten zu einem *Vorgang* oder einer *Person* zusammen.
- Beispiel: Personendaten

Typ-Definition

```
struct Person {
  int id;
  char name[60];
  char vorname[60];
  char telefon[40];
};
```

Variablen-Initialisierung

```
struct Person az = {
  70970,
  "Zeller",
  "Andreas",
  "+49681410978-0"
};
```

Datenbanken

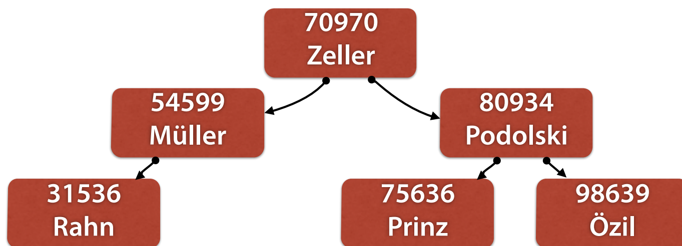
- Um große Mengen von Personen zu speichern, könnte ich ein *Feld* benutzen
- Problem: Wie groß soll das Feld sein?

```
struct Person {
  int id;
  char name[60];
  char vorname[60];
  char telefon[40];
};

struct Person kunden[1000];
```

Ich könnte das Feld dynamisch anlegen, und bei Bedarf vergrößern – aber dann müsste ich bei jedem Vergrößern das Feld umkopieren, und das wäre sehr teuer.

Suchbäume

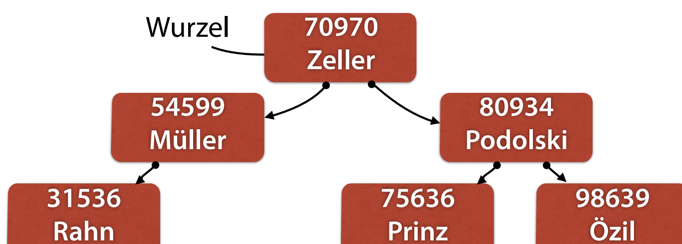
- Ein *Suchbaum* ist eine *dynamische* Datenstruktur zum Speichern und Durchsuchen großer Datenmengen



Suchbäume

- Jeder *Knoten* hat (bis zu zwei) *Kinder:* Im *linken* Teilbaum sind alle kleineren, im *rechten* Teilbaum alle größeren Werte

Wurzel



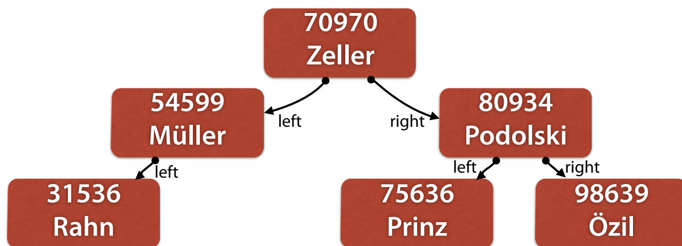
Knoten

- Im Baumknoten lege ich die zu speichernden Werte ab –
- und zwei *Zeiger* auf die Teilbäume

```
struct Node {
  int id;
  char name[60];
  // Mehr Felder…

  struct Node *left;
  struct Node *right;
};
typedef struct Node node;
```

Suchen

- Ich suche den Wert *x* und beginne in *k*
- Ist *x* < *k*.id, suche ich im *linken* Teilbaum
- Ist *x* > *k*.id, suche ich im *rechten* Teilbaum



Knoten suchen

- *p*->elem ist dasselbe wie (**p*).elem

```
node *find_node(node *root, int id)
{
  if (id == root->id)
    return root;

  if (id < root->id && root->left != NULL)
    return find_node(root->left, id);

  if (id > root->id && root->right != NULL)
    return find_node(root->right, id);

  return NULL;
}
```

Einfügen

- Ich suche den Wert x und beginne in k
- Ist $x < k$.id, füge ich im *linken* Teilbaum ein
- Ist $x > k$.id, füge ich im *rechten* Teilbaum ein

Einfügen

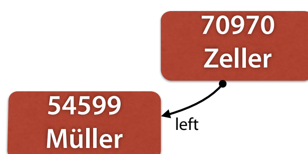
- Ich suche den Wert x und beginne in k
- Ist $x < k$.id, füge ich im *linken* Teilbaum ein
- Ist $x > k$.id, füge ich im *rechten* Teilbaum ein

54599
Müller

70970
Zeller

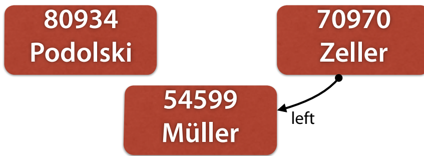
Einfügen

- Ich suche den Wert x und beginne in k
- Ist $x < k$.id, füge ich im *linken* Teilbaum ein
- Ist $x > k$.id, füge ich im *rechten* Teilbaum ein



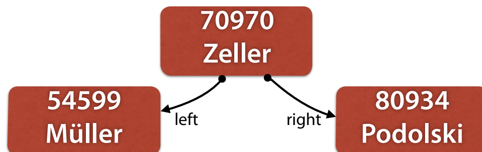
Einfügen

- Ich suche den Wert *x* und beginne in *k*
- Ist *x* < *k*.id, füge ich im *linken* Teilbaum ein
- Ist *x* > *k*.id, füge ich im *rechten* Teilbaum ein



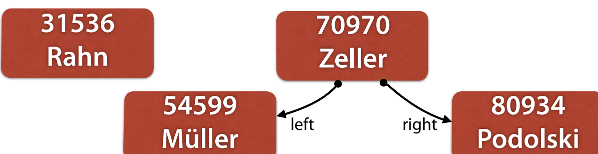
Einfügen

- Ich suche den Wert *x* und beginne in *k*
- Ist *x* < *k*.id, füge ich im *linken* Teilbaum ein
- Ist *x* > *k*.id, füge ich im *rechten* Teilbaum ein



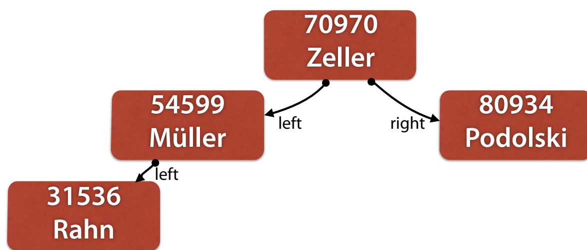
Einfügen

- Ich suche den Wert *x* und beginne in *k*
- Ist *x* < *k*.id, füge ich im *linken* Teilbaum ein
- Ist *x* > *k*.id, füge ich im *rechten* Teilbaum ein



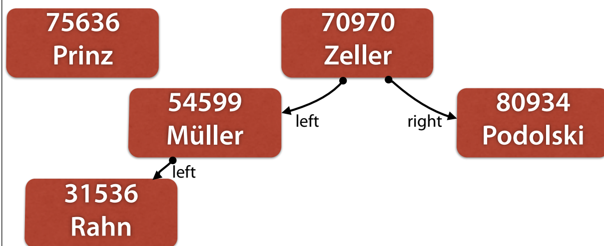
Einfügen

- Ich suche den Wert *x* und beginne in *k*
- Ist $x < k$.id, füge ich im *linken* Teilbaum ein
- Ist $x > k$.id, füge ich im *rechten* Teilbaum ein



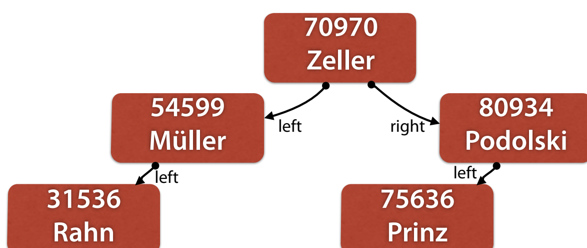
Einfügen

- Ich suche den Wert *x* und beginne in *k*
- Ist $x < k$.id, füge ich im *linken* Teilbaum ein
- Ist $x > k$.id, füge ich im *rechten* Teilbaum ein



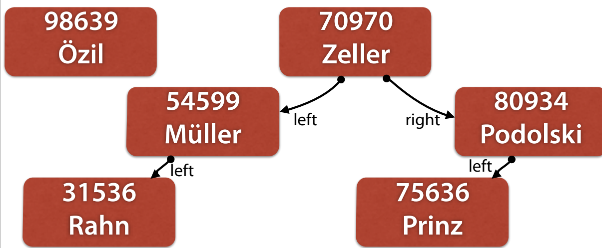
Einfügen

- Ich suche den Wert *x* und beginne in *k*
- Ist $x < k$.id, füge ich im *linken* Teilbaum ein
- Ist $x > k$.id, füge ich im *rechten* Teilbaum ein



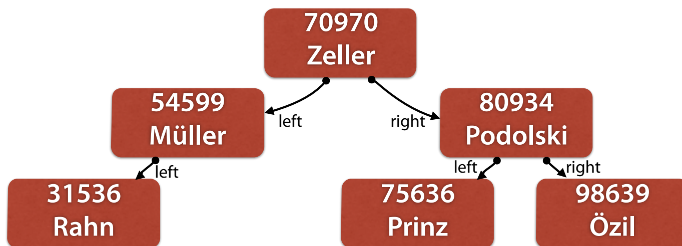
Einfügen

- Ich suche den Wert *x* und beginne in *k*
- Ist *x* < *k*.id, füge ich im *linken* Teilbaum ein
- Ist *x* > *k*.id, füge ich im *rechten* Teilbaum ein



Einfügen

- Ich suche den Wert *x* und beginne in *k*
- Ist *x* < *k*.id, füge ich im *linken* Teilbaum ein
- Ist *x* > *k*.id, füge ich im *rechten* Teilbaum ein



Knoten erzeugen

- Knoten werden im *Freispeicher* angelegt

```
node *make_node(int id, char name[])
{
  node *nd = (node *)malloc(sizeof(node));
  nd->id = id;
  strncpy(nd->name, name, sizeof(nd->name));
  nd->left  = NULL;
  nd->right = NULL;

  return nd;
}
```

strncpy(s, t, n) kopiert bis zu n Zeichen von t nach s. Auf diese Weise vermeiden wir Überläufe.

Knoten einfügen

```
void insert_node(node *root, node *nd)
{
  if (nd->id < root->id)
  {
    if (root->left == NULL)
      root->left = nd;
    else
      insert_node(root->left, nd);
  }
  else if (nd->id > root->id)
  {
     // analog für rechts
  }
}
```

strncpy(s, t, n) kopiert bis zu n Zeichen von t nach s. Auf diese Weise vermeiden wir Überläufe.

Baum füllen

```
node *create_tree()
{
  node *root = make_node(70970, "Zeller");

  insert_node(root, make_node(54599, "Mueller"));
  insert_node(root, make_node(80934, "Podolski"));
  insert_node(root, make_node(31536, "Rahn"));
  insert_node(root, make_node(75636, "Prinz"));
  insert_node(root, make_node(98639, "Oezil"));

  return root;
}
```

Demo

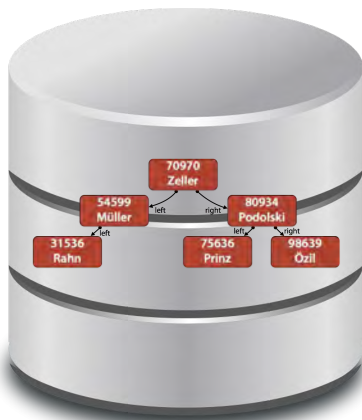
Komplexität

- Einfügen, Suchen, Löschen:
  $\log_2 n$ Vergleiche *(logarithmisch)*
- Baum muss *ausgeglichen* sein

Suchbäume sind äußerst effizient: Alle wichtigen Operationen skalieren beliebig

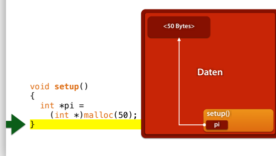
---

Datenbanken



Wenn immer Sie eine Datenbank brauchen, um große Datenmengen zu verwalten – im Innern werkeln überall Suchbäume, wie wir sie hier gesehen haben.

Freispeicher




```
void setup()
{
  int *pi =
    (int *)malloc(50);
}
```

Freispeicher

1. Du sollst nicht zu viel Speicher anfordern!
2. Du sollst nicht zu wenig Speicher anfordern!
3. Du sollst angeforderten Speicher wieder freigeben!
4. Niemals sollst Du auf freigegebenen Speicher zugreifen!
5. Du sollst Speicher nicht doppelt freigeben!

Verbünde

- In C können einzelne Daten zu einem *Verbund* ("struct") zusammengefasst werden
- Beispiel: *Komplexe Zahlen*

Typ-Definition

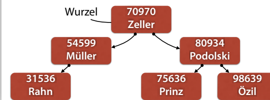
```
struct Complex {
  double real;
  double imag;
};
```

Variablen-Initialisierung

```
struct Complex c = {
    3.0,  // real
    4.0   // imag
};
```

Suchbäume

- Jeder *Knoten* hat (bis zu zwei) *Kinder:*
  Im *linken* Teilbaum sind alle kleineren,
  im *rechten* Teilbaum alle größeren Werte



Handouts

Zeiger

- Ein *Zeiger* ist eine Variable, die die *Adresse* einer Variablen speichert
- Man sagt: Der Zeiger "zeigt" auf die Variable
- Ein Zeiger mit Namen *p*, der auf einen Typ T zeigt, wird als *T *p* deklariert:

```
int *p1 = &ledPin;
```

Dereferenzieren

- Der Ausdruck **p* steht für die Variable, auf die *p* zeigt (= die Variable an Adresse *p*)
- Man sagt: Der Zeiger wird *dereferenziert*
- **p* kann wie eine Variable benutzt werden

```
int *p1 = &ledPin;
int x = *p1;   // x = ledPin
*p1 = 25;      // ledPin = 25
```

Werte austauschen

- Wir wollen eine Funktion swap(*a*, *b*) schreiben, die die Werte von *a* und *b* vertauscht
- Wir übergeben die *Adressen* von *a* und *b*

```
int x = 1; int y = 2;
swap(&x, &y);
// x = 2, y = 1
```

Tauschen mit Zeigern

```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Freispeicher

- Die C-Funktion malloc(*n*) erzeugt einen Speicherbereich, bestehend aus *n* Bytes
- Beispiel: 100 int-Elemente

```
int *pi =
  (int *)malloc(sizeof(int) * 100);
```

Freispeicher freigeben

- Wenn ich den Speicher nicht mehr benötige, muss ich ihn mit free() freigeben

```
int *pi =
   (int *)malloc(sizeof(int) * 100);

// ...Zugriff auf pi...

free(pi);
```

NULL-Zeiger

- NULL wird in Programmen grundsätzlich als Wert für "ungültige Adresse" benutzt
- Wird NULL dereferenziert, führt dies zum sofortigen Absturz (hoffentlich)

```
int *pi = NULL;
*pi = 25;
```

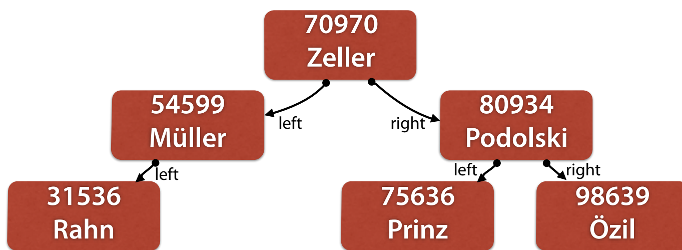
Zeigerarithmetik

- Ist p ein Zeiger auf ein Feldelement, dann zeigt $p + 1$ auf das *nächste* Element.

```
char s[100] = "Hall";

char *pc = s; // 1. Element
*pc = 'B';
pc = pc + 1;  // 2. Element
*pc = 'i';
```

Suchen in Bäumen

- Ich suche den Wert *x* und beginne in *k*
- Ist *x* < *k*.id, suche ich im *linken* Teilbaum
- Ist *x* > *k*.id, suche ich im *rechten* Teilbaum



Suchbäume

- Im Baumknoten lege ich die zu speichernden Werte ab –
- und zwei *Zeiger* auf die Teilbäume

```
struct Node {
  int id;
  char name[60];
  // Mehr Felder…

  struct Node *left;
  struct Node *right;
};
typedef struct Node node;
```

Knoten suchen

```
node *find_node(node *root, int id)
{
  if (id == root->id)
    return root;

  if (id < root->id && root->left != NULL)
    return find_node(root->left, id);

  if (id > root->id && root->right != NULL)
    return find_node(root->right, id);

  return NULL;
}
```

Knoten einfügen

```
void insert_node(node *root, node *nd)
{
  if (nd->id < root->id)
  {
    if (root->left == NULL)
      root->left = nd;
    else
      insert_node(root->left, nd);
  }
  else if (nd->id > root->id)
  {
     // analog für rechts
  }
}
```

strncpy(s, t, n) kopiert bis zu n Zeichen von t nach s. Auf diese Weise vermeiden wir Überläufe.