Dynamische Datenstrukturen

Programmieren für Ingenieure
Sommer 2014

Andreas Zeller, Universität des Saarlandes

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

Ihr Projekt

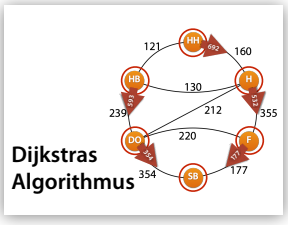
- **Bis 27. Juni**: *Projektskizze* abgeben
- Eine A4-Seite mit
  - Was soll gemacht werden?
  - Warum ist das originell?
  - Warum ist das schwer?
- Feedback bis 1. Juli

Abgabe

- **Bis 25. Juli**: *Projekt* einreichen
  - Schaltplan, Quellcode
  - Aufbauanleitung
  - Einsatzanleitung
  - Demo-Video

Projekt-Beispiele

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



Tabellen in C

- Eine *Tabelle* in C wird so deklariert:

0	1
2	3
4	5

```
int a[2][3] = {
    {0, 1},  // a[0]
    {2, 3},  // a[1]
    {4, 5}   // a[2]
};
```

- Eigentlich ein *Feld aus 3 Feldern* mit je 2 Elementen

Konstanten

- Problem: Feldgrößen müssen *konstant* sein
- Lösung: Variable als *Konstante* markieren

```
const int COLS = 2;
const int ROWS = 3;

int a[COLS][ROWS] = {
    {0, 1},  // a[0]
    {2, 3},  // a[1]
    {4, 5}   // a[2]
};
```

- Ältere Alternative: `#define COLS 2`

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;
```

Flashback: 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
```

Tauschen mit Zeigern

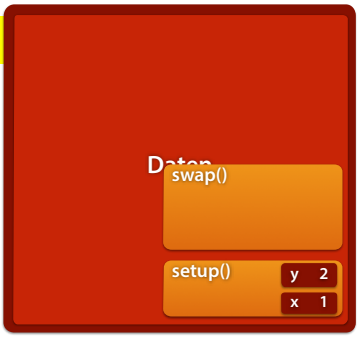
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

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

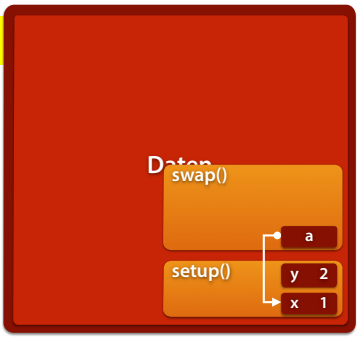
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

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.

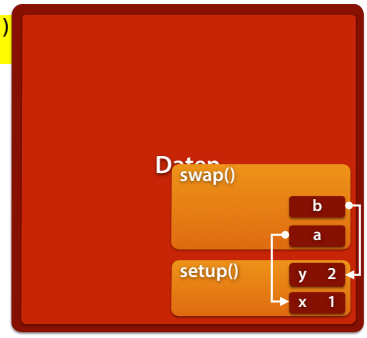
Werte austauschen

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

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



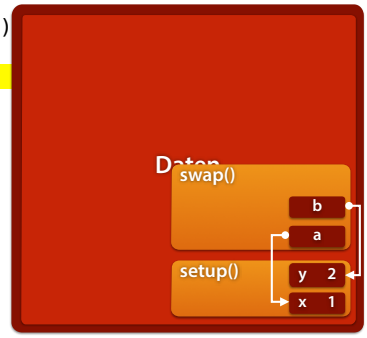
Werte austauschen

```
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

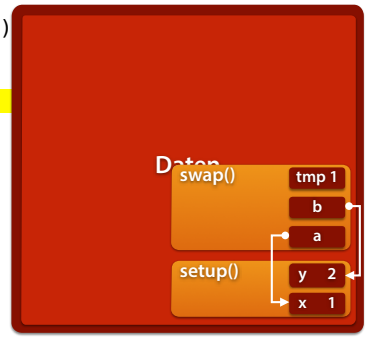
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

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

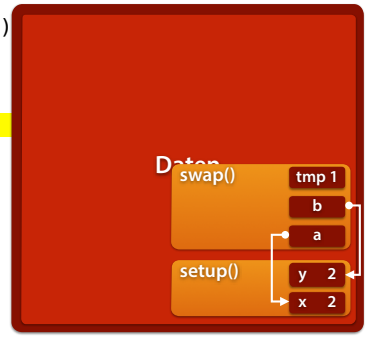
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

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



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

Werte austauschen

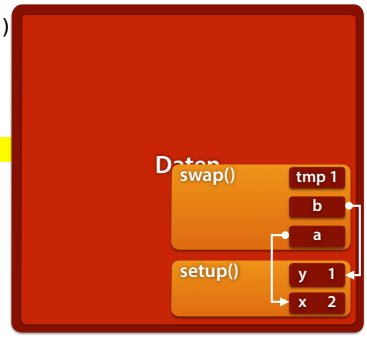
```
void swap(int *a, int *b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

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);
}
```

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

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!

Themen heute

Bild: Ohio State

Themen heute

- Freispeicher

Themen heute

- Freispeicher
- Verbünde

Themen heute

- Freispeicher
- Verbünde
- Suchbäume

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

```
void setup()
{
    malloc(50);
}
```

Freispeicher anlegen

```
void setup()
{
    malloc(50);
}
```

Zugriff Freispeicher

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

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

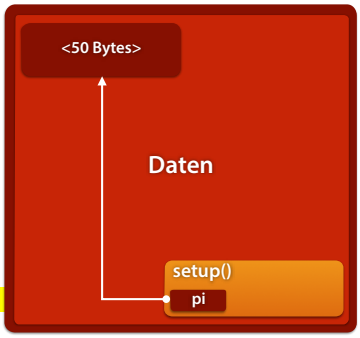
Typumwandlung

Zugriff Freispeicher

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

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

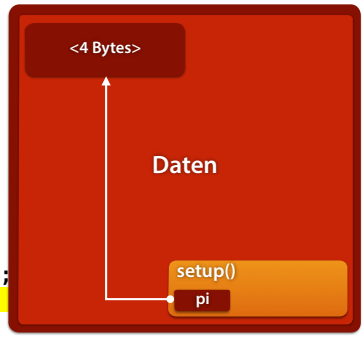
Größe

Größe Freispeicher

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

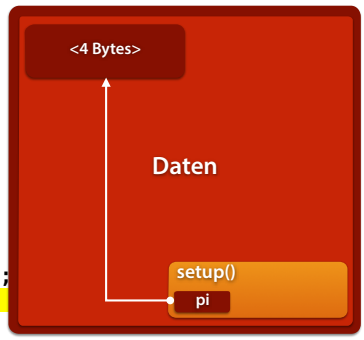
Größe Freispeicher

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



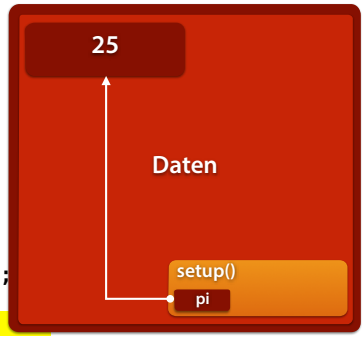
Größe Freispeicher

```
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);
```

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);
```

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);
```

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);
```

Beispiel: Feld einlesen

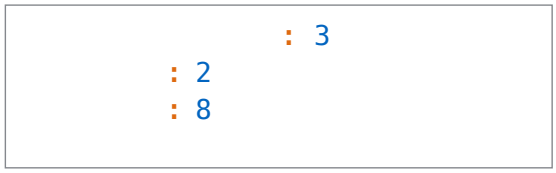
```
int n = get_number_of_values();
int *values =
 (int *)malloc(sizeof(int) * n);

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

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);
```

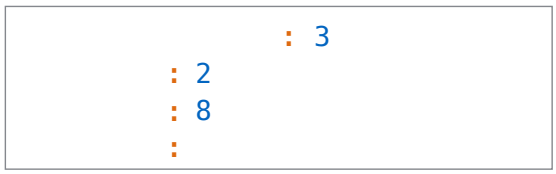
Beispiel: Feld einlesen

```
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);
```



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);
```



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);
```

```
                : 3
        : 2
        : 8
        :  1
```

Demo

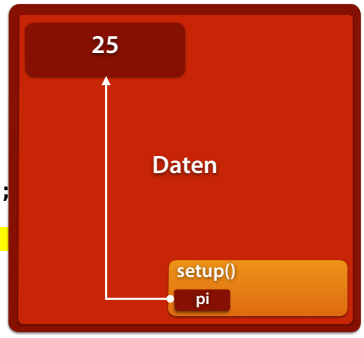
Freispeicher freigeben

- 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);
```

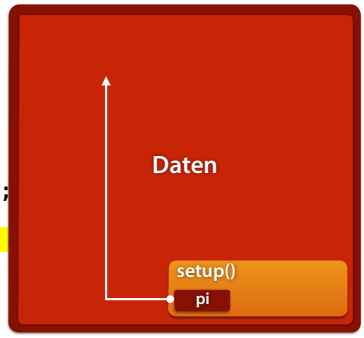
Freispeicher freigeben

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



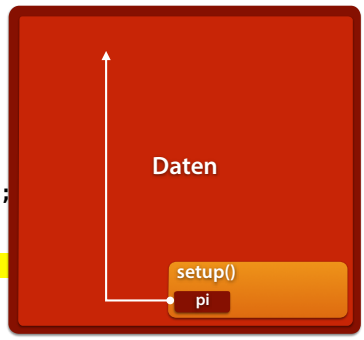
Freispeicher freigeben

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



Freispeicher freigeben

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



Freispeicher freigeben

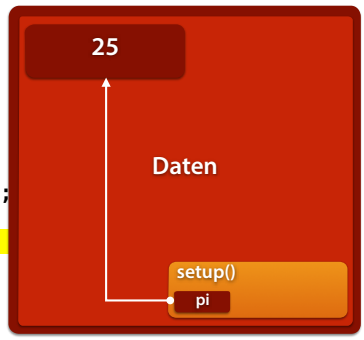
- Freigegebener Freispeicher darf *nicht weiter benutzt werden!*
- Dummerweise merkt man nicht sofort, ob Speicher bereits freigegeben wurde…
- Wird Speicher *2x freigegeben*, kommt es irgendwann zum Absturz (*Zeitbombe*).

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;
}
```



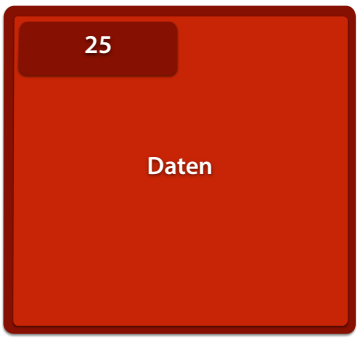
Speicherleck

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

Speicherleck

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

Speicherleck

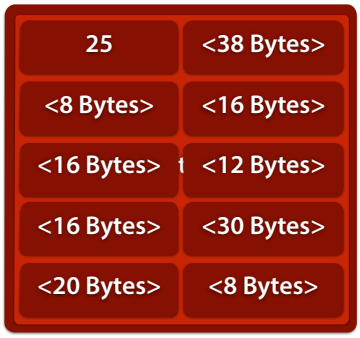


Speicherleck

- Auf Dauer wird so der verfügbare Speicher immer weniger…

Speicherleck

- Auf Dauer wird so der verfügbare Speicher immer weniger…



Speicherleck

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

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

Share 0 Tweet 0 +1 0 Share Pin it

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

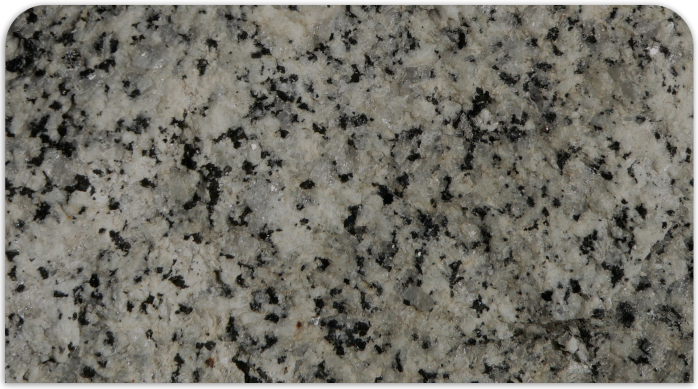
```
int *pi = (int *)malloc(10000000000);
if (pi == NULL) {
  Serial.println("Speicher voll");
  abort();
}
```

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



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

Freispeicher

Freispeicher

1. Du sollst nicht zu viel Speicher anfordern!
2. Du sollst nicht zu wenig Speicher anfordern!

Freispeicher

1. Du sollst nicht zu viel Speicher anfordern!
2. Du sollst nicht zu wenig Speicher anfordern!
3. Du sollst angeforderten Speicher wieder freigeben!

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. Niemal$ sollst Du auf freigegebenen Speicher

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. Niemal$ sollst Du auf freigegebenen Speicher

Freispeicher

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[1] = 'i';
```

ist dasselbe wie

```
*pc = 'B';
*(pc + 1) = '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

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

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);}
```

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

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.

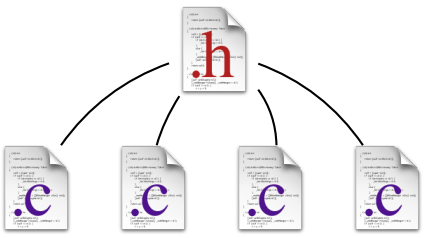
```
complex make_complex(double real, double imag)
{
    complex c;
    c.real = real;
    c.imag = imag;
    return c;
};

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

```
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

Macht Deklarationen aus *complex.h* verfügbar

```
#include "complex.h"

complex make_complex(double real, double imag)
{ … }

complex complex_sum(complex c1, complex c2)
{ … }

void print_complex(complex c)
{ … }
```

---

user.c

Macht Deklarationen aus *complex.h* verfügbar

```
#include "complex.h"

void my_function() {
  complex c = make_complex(…);
  print_complex(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

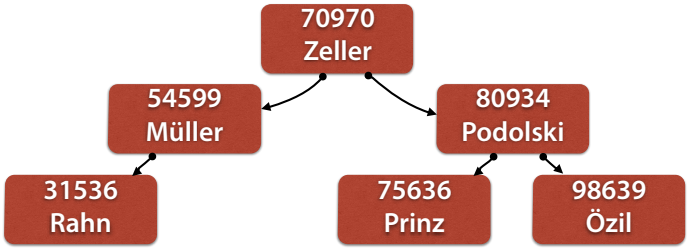
- Um große Mengen von Personen zu speichern, könnte ich ein *Feld* benutzen
- Problem: Wie groß soll das Feld sein?

```
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.

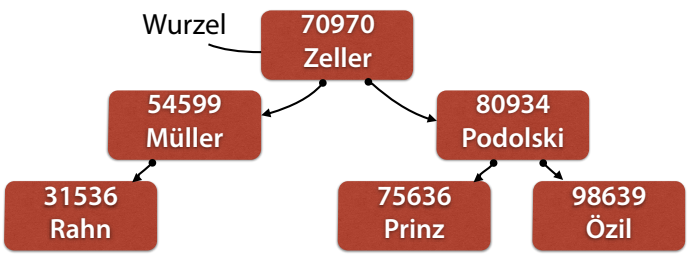
Suchbäume

- 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



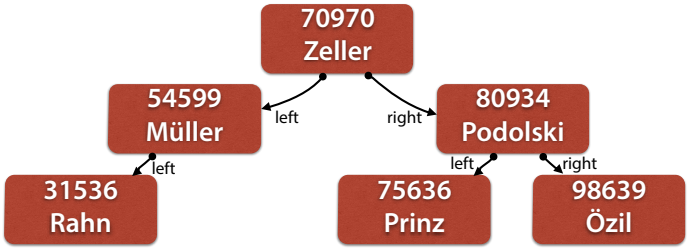
Knoten

- 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



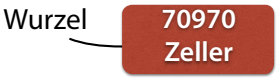
Knoten suchen

- *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

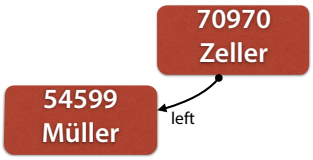


Einfügen

- 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

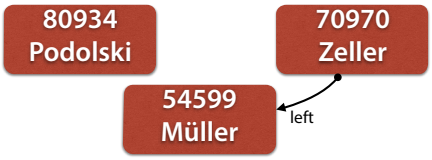
Einfügen

- 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



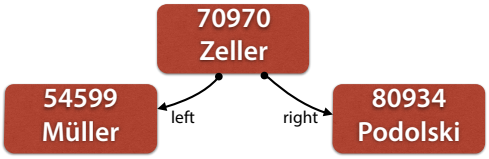
Einfügen

- 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



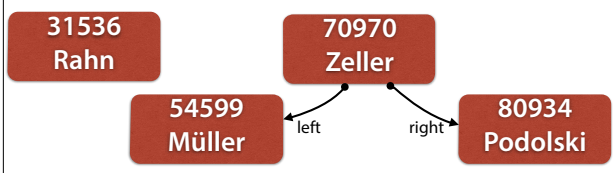
Einfügen

- 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



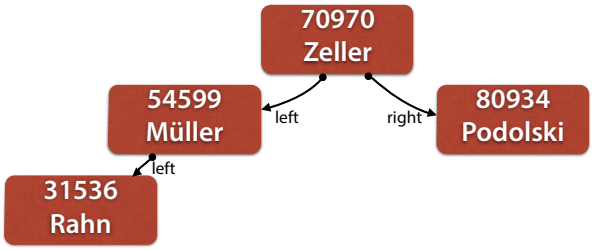
Einfügen

- 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



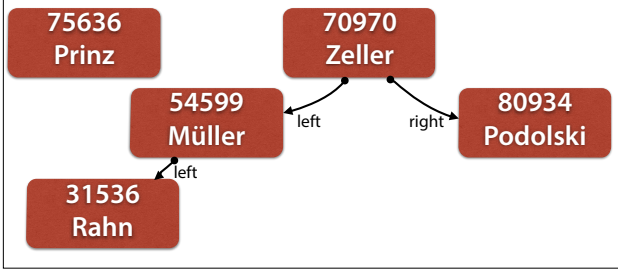
Einfügen

- 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



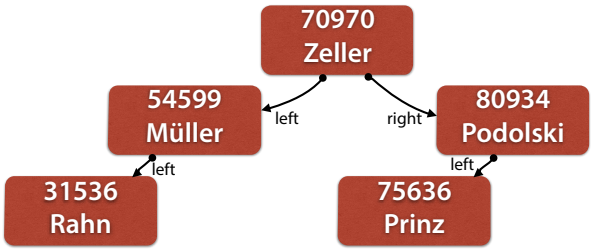
Einfügen

- 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



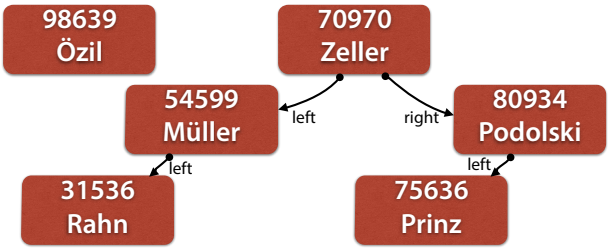
Einfügen

- 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



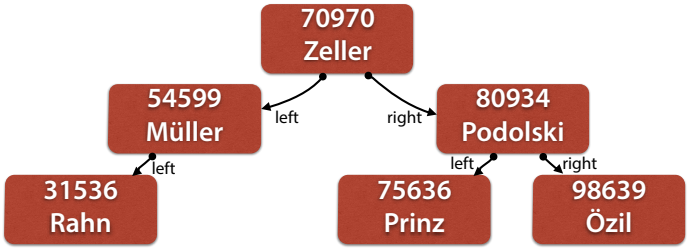
Einfügen

- 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

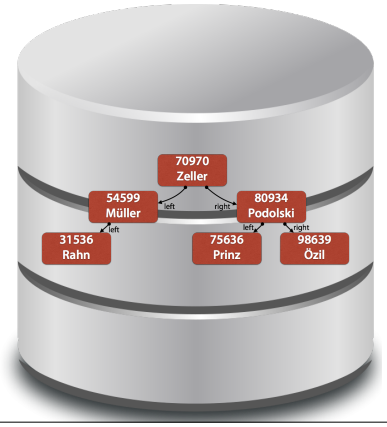
Komplexität

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

Suchbäume sind äußerst effizient: Alle wichtigen Operationen skalieren beliebig

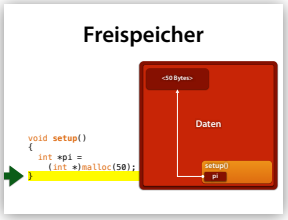
---

Datenbanken

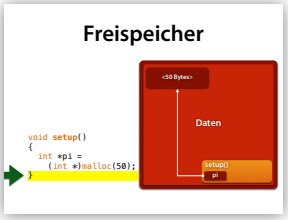


Wenn immer Sie eine Datenbank brauchen, um große Datenmengen zu verwalten – im Innern werkeln überall Suchbäume, wie wir sie hier gesehen haben.

Freispeicher



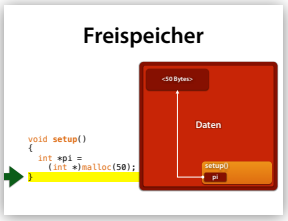
Freispeicher



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!

Freispeicher



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

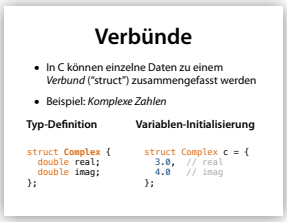
- In C können einzelne Daten zu einem *Verbund* ("struct") zusammengefasst werden
- Beispiel: *Komplexe Zahlen*

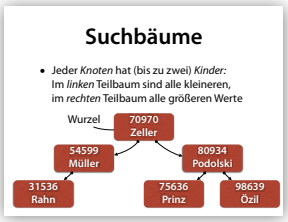
Typ-Definition

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

Variablen-Initialisierung

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





Handouts

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;
```

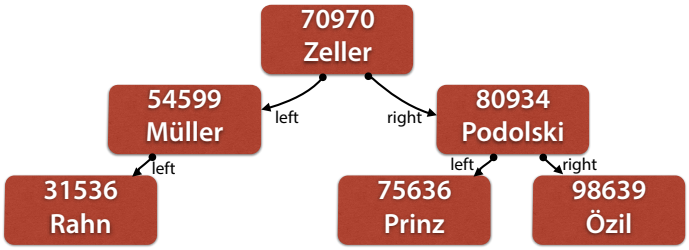
Zeigerarithmetik

- 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.