Dynamic Data Structures

Programming for Engineers
Winter 2015

Andreas Zeller, Saarland University

An Algorithm

- unambiguous instruction to solve a problem
- consists of finitely many, well defined steps
- typically implemented as computer programs

Algorithms

Calculate	Search	Sort
• Fibonacci • GCD • Collatz	• Linear • Binary	• Insertion • Merge

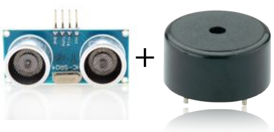
Sort In-Place

a

0	1	2	3	4	5	6	7	8	9	10
10	-10	7	2	2	-4	-7	-10	1	4	5

- We want to sort inside of an array
- Assume: array a[0…i−1] is already sorted
- We inspect the element a[i]…
- …and insert it into the sorted array

Theremin =



Dates

Exam

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

Demo exam available next week

Today's Topics

- Pointer
- Dynamic memory
- Structs
- Search trees

Bild: Ohio State

Swapping Values

- We want to write a function swap(*a*, *b*) that swaps the values of a and b

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

First Attempt

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

Swapping Values

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

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

Swapping Values

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

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

Swapping Values

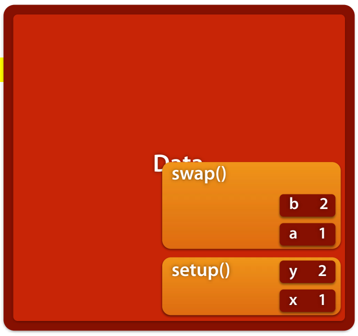
```
void swap(int a, int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}

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

Swapping Values

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

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

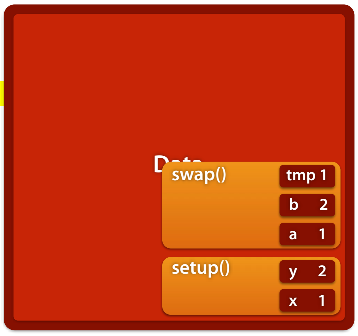
Swapping Values

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

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



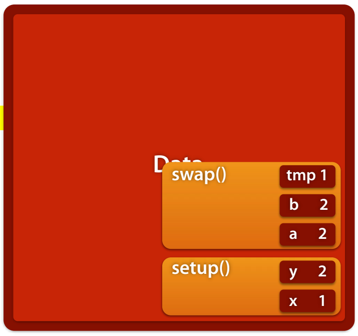
Swapping Values

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

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



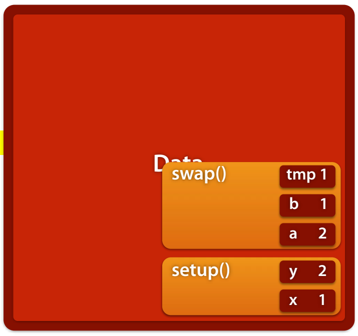
Swapping Values

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

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



Swapping Values

```
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 far, so good. We swapped a and b. However, this does not propagate to the caller.

Swapping Values

```
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(), all values remain as they were :-(

Swapping Values

- A function cannot alter the local variables of another function

Large Programs

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

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

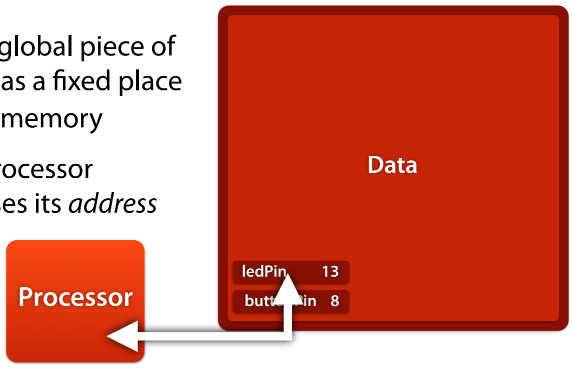
- *c* is a copy of *client*
- Consequence: *two* connections!
- How can we make doSomething() work on the *client* from loop()?

Pointers

- A function cannot alter the local variables of another function…
- …unless it uses a *pointer*

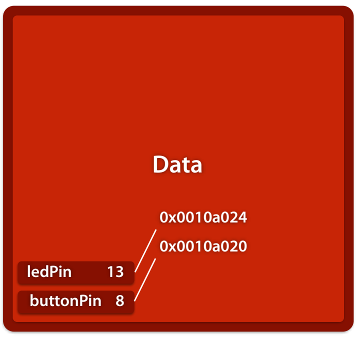
Memory Locations

- Every global piece of data has a fixed place in the memory
- The processor accesses its *address*



Memory Locations

- An address tells the processor where the value can be found
- A number *similar to a house number*
- The address of *ledPin* could be **0x0010a024**



0x0010a0024

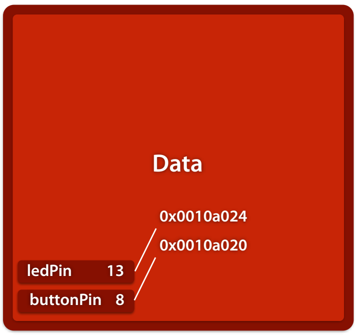
- Hexadecimal number = number in base 16
- Written in C with prefix 0x
- Digits: 0–9 as usual, additionally
  A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
- 0xA3 is $10 \cdot 16^1 + 3 = 163$
- 0xAFFE is
  $10 \cdot 16^3 + 15 \cdot 16^2 + 15 \cdot 16^1 + 14 = 45054$

17432612

- Hexadecimal number = number in base 16
- Written in C with prefix 0x
- Digits: 0–9 as usual, additionally
  A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
- 0xA3 is $10 \cdot 16^1 + 3 = 163$
- 0xAFFE is
  $10 \cdot 16^3 + 15 \cdot 16^2 + 15 \cdot 16^1 + 14 = 45054$

Addresses

- In C, ***&x*** returns the address *of x:*
- &ledPin = 0x0010a024
- &buttonPin = 0x0010a020



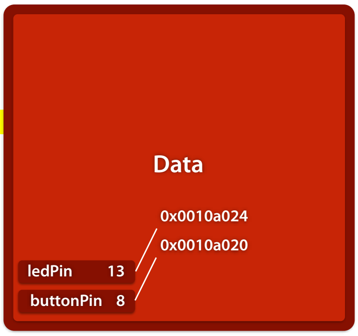
Pointers

- A *pointer is a* variable that stores the address of another variable
- We say: The pointer "points to" a variable
- A pointer named *p that points to variable of type* T is declared as *T *p*:

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

Pointers

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

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



Pointers

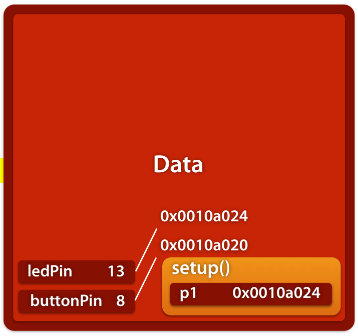
```
int ledPin = 13;
int buttonPin = 8;

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

Pointers

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

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

Pointers

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

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



Pointers

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

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

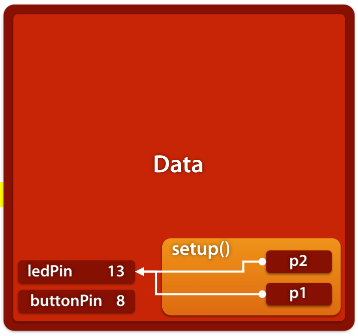
Note that we do not need the actual address of ledPin – it suffices to know that p1 _points to_ ledPin – that is, its value is the address of ledPin.

Pointers

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

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

Pointers

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

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



Let's add another pointer, will we?

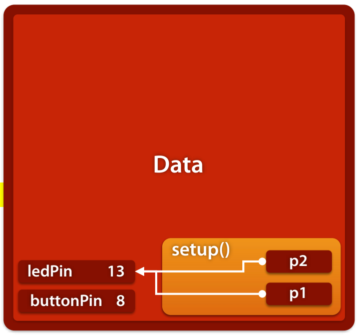
Dereferencing

- The expression **p* represents the variable, that p points to (= the variable at address *p*)
- We say: The pointer gets *dereferenced*
- **p* can be used like a variable

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

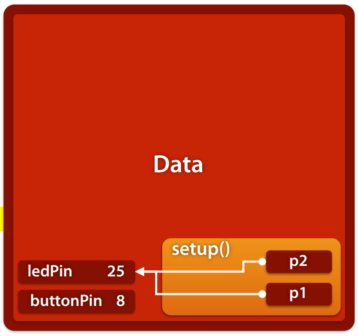
Dereferencing

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

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



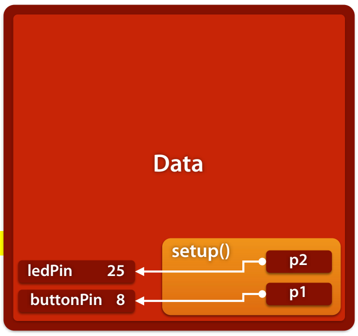
Dereferencing

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

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



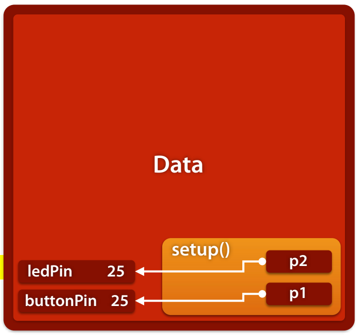
Dereferencing

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

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



Dereferencing

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

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



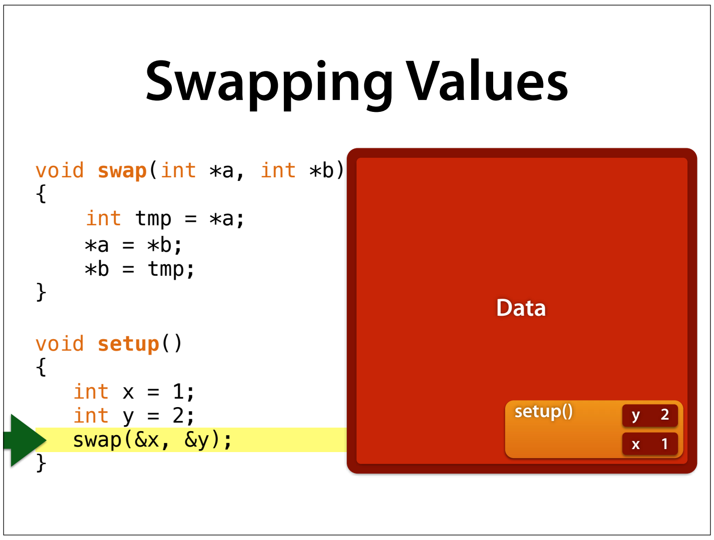
Swapping Values

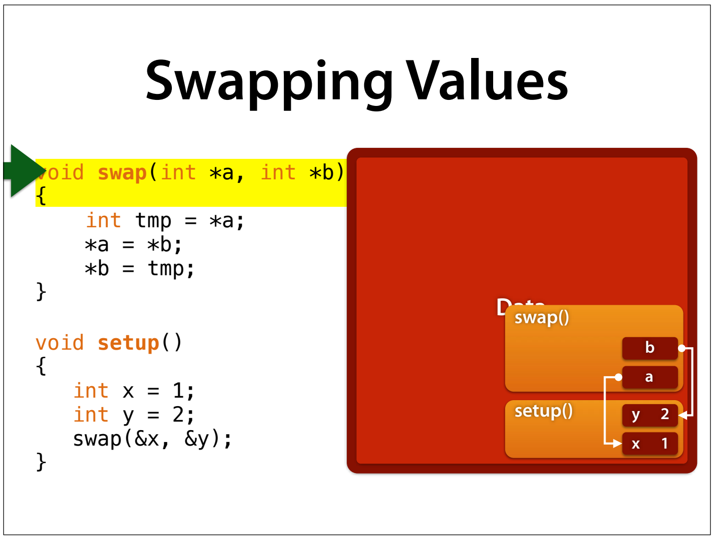
- We want to write a function swap(*a*, *b*) that swaps the values of a and b
- We pass the *addresses* of *a* and *b*

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

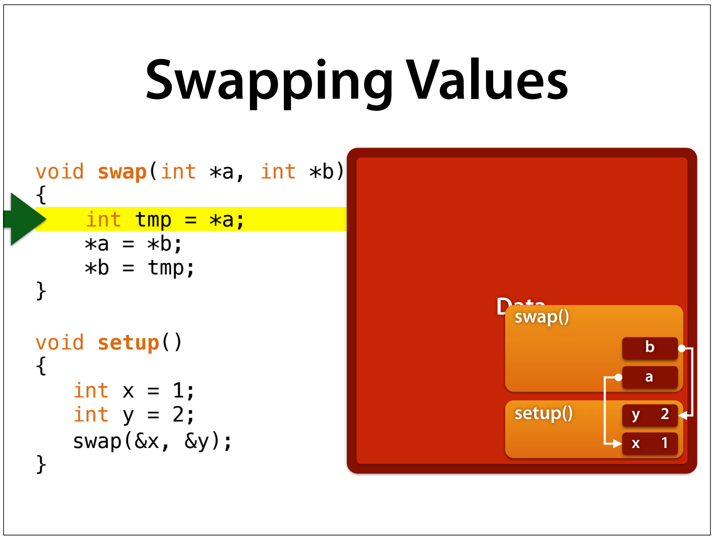
Swapping with Pointers

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

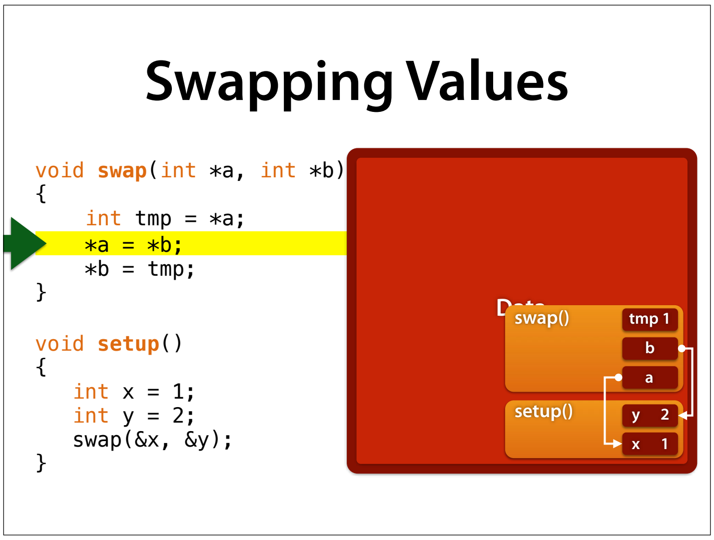




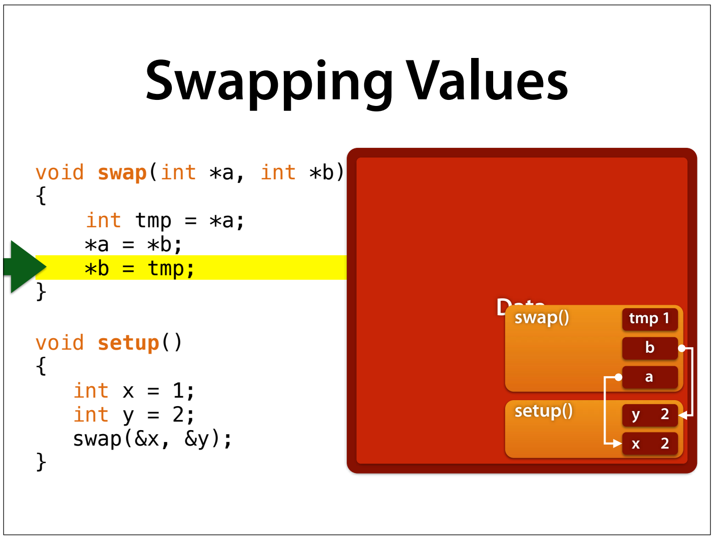
a now points to x, b to y. *a is the value at the address of a – that is, the value of x.



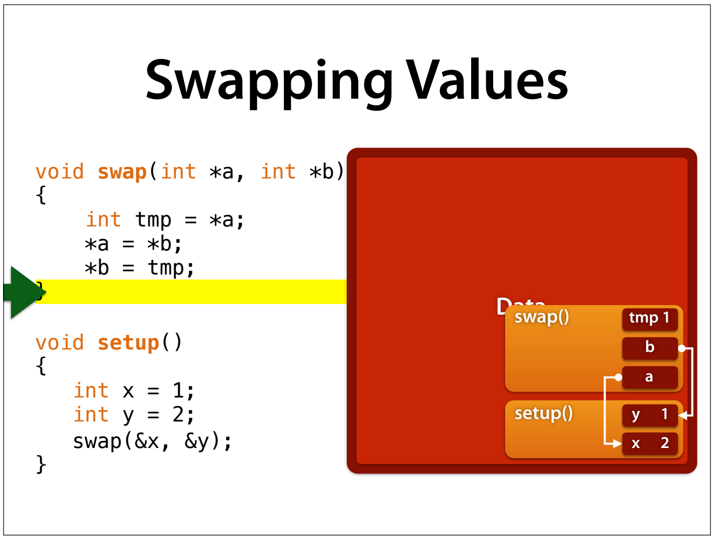
a now points to x, b to y. *a is the value at the address of a – that is, the value of x.



Assigning *a a new value changes the value of x



Likewise, *b changes y



Swapping Values

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

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

Done!

and at the end, x and y are swapped (as intended!)

Large Programs

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

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

- Better alternative without copying

Fun with Pointers

With pointers you can

- make a function change variables
- *manage dynamic memory*
- access arrays
- *build complex data structures*

Dynamic Data Structures

- In C the size of an array must be already known at compile time (“statically”)
- But what if we only find out the size at runtime (“dynamically”)?

- We load a map (e.g. from a file)
- The map contains a list of cities and roads
- The list may be of different sizes depending on the map

Dynamic Memory

- *Dynamic memory* is memory which is requested at runtime
- We can set the size
- We get a pointer to the memory
- Most important use case of pointers!

Allocating Dynamic Memory

- The C function malloc(*n*) creates a new memory region consisting of *n* bytes

Allocating Dynamic Memory

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

Allocating Dynamic Memory

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

Accessing Dynamic Memory

- malloc() returns a pointer to the allocated memory
- This pointer must be converted ("cast") to the required type.

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

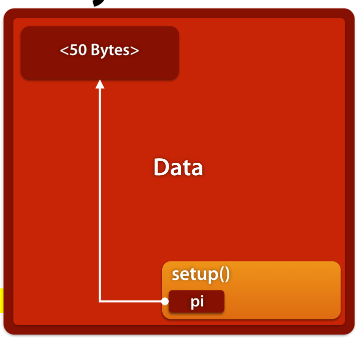
typecast

Accessing Dynamic Memory

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

Accessing Dynamic Memory

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



Dynamic Memory Size

- The function sizeof(*x*) *returns* the size of *x* (in bytes)
- *x* is a type or a variable

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

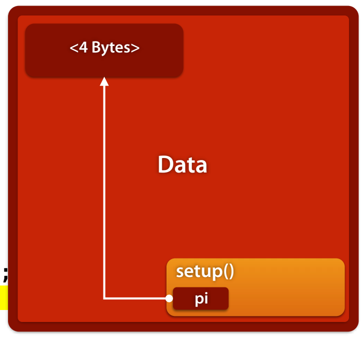
typecast   size

Dynamic Memory Size

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

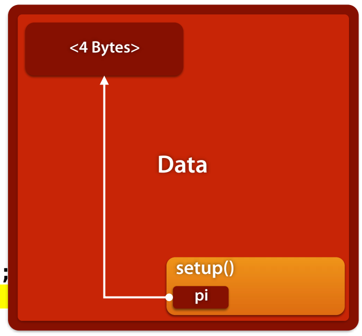
Dynamic Memory Size

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



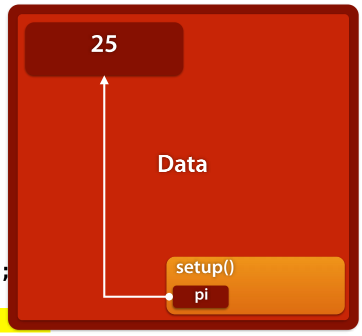
Dynamic Memory Size

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



Dynamic Memory Size

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



Dynamic Memory for Arrays

- Using dynamic memory we can also request enough memory for an array
- Example: 100 int elements

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

Dynamic Memory for Arrays

- By using the pointer as *the name of the array we can access the elements as usual*:

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

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

Example: Read Array

- First we read n and then n values

```
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);
```

Example: Read Array

```
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

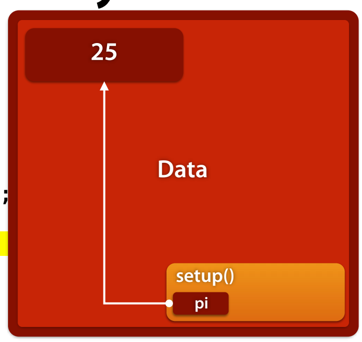
Freeing Dynamic Memory

- When we no longer need the memory, we must dispose of it with free()

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

// ...accessing pi...

free(pi);
```

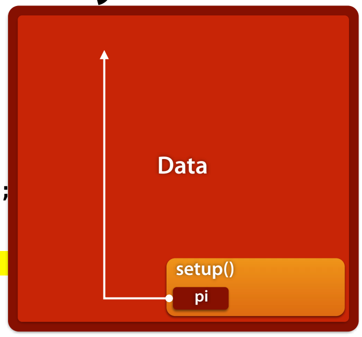
Freeing Dynamic Memory

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



Freeing Dynamic Memory

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



Freeing Dynamic Memory

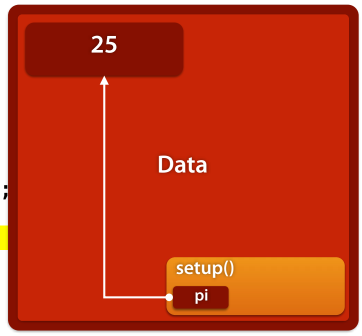
- Freed dynamic memory must *no longer be used!*
- Unfortunately one can't tell immediately whether memory was already freed…
- If memory is freed twice, the program eventually crashes (*time bomb*).

Freeing Dynamic Memory

- If dynamic memory is not freed after use, it simply remains
- If it is not possible to access, then the memory can no longer be freed (*memory leak*)

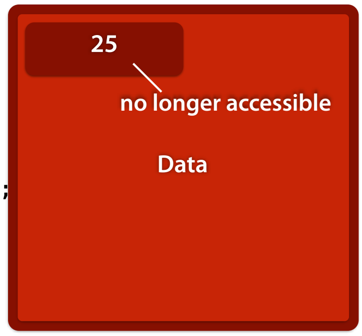
Memory Leak

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



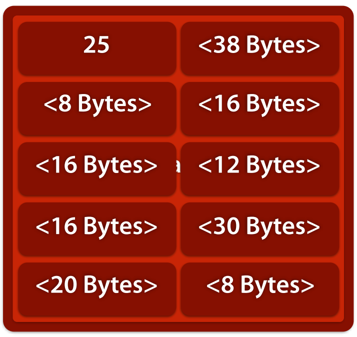
Memory Leak

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



Memory Leak

- In the long run the available memory get less and less…
- until no more dynamic memory is available.



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-

Memory Leaks can lead to real problems.

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.

Out of Memory

- When no more memory is available, malloc() returns the special address *NULL*

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

NULL Pointer

- NULL is generally used in programs for the value of "invalid address"
- Dereferencing NULL leads to an immediate crash (hopefully)

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

Dynamic Memory

1. Thou shalt not request too much memory!
2. Thou shalt not request too little memory!
3. Thou shalt free the requested memory!
4. Thou shalt never access freed memory!
5. Thou shalt not free the memory twice!

And whoever violates these rules may burn to hell for eternity, and his programs may rot away.

Dynamic Memory in C++

- In C++ dynamic memory can be accessed more easily:

C

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

C++

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

Dynamic Memory in C++

- In C++ dynamic memory can be accessed more easily:

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

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

Note: Memory from „malloc“ must be freed with „free“, memory from „new“ must be freed with „delete“, und memory from „new[]“ must be freed with „delete[]“. Mixing up things will have terrible consequences.

Pointers and Arrays

- In C and C++ every array name represents the address at which the array begins

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

char *pc = &s[0]; // first element
*pc = 'B';
```

is the same as

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

Pointer Arithmetic

- If *p* is a pointer to an array element, then *p* + 1 points to the next element.

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

char *pc = s; // first element
*pc = 'B';
pc = pc + 1;  // second element
*pc = 'i';
```

Pointer Arithmetic

- If *p* is a pointer to an array element, then *p* + 1 points to the next element.

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


char *pc = s; // first element
while (*pc++ != '\0');
return pc - s; // length of s
```

Pointer Arithmetic

- *p*[*i*] can also be written as *(*p* + *i*)

```
char s[100] = "Hall";
char *pc = s; //first element

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

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

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

…and since + is commutative, pc[1] is the same as *(pc + 1), the same as *(1 + pc), and therefore 1[pc]. If you want to utterly confuse the readers of your program, here's some ideas.

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)
The program fits into a single line. It displays a keyboard with the keys present in the argument highlighted (“a.out qwerty”).

Demo Exam

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

You will get puzzles in the exam, but not as tough as this one.

Structs

In real life data is often *composed of other data:*

- *Fractions* consist of *numerator* and *denominator*
- *Dimensions consist of width, height, depth*
- *Coordinates* consist of *x, y, z values*

Structs

- In C we can combine data into a struct (also called record)
- Example: Complex *Numbers*

Type definition

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

Variable initialisation

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

Structs

- We can access the elements of a struct with *variable.element*

Variable initialisation

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

Usage

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

Since writing "struct complex" all the time is a bit bothersome, there is a way to have shortcuts

Type Definitions

- By using *typedef typname alias we make alias an alternative* (shorter) name for *typname*

Type definition

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

typedef struct Complex
  complex;
```

Usage

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

Structs

- A struct can serve as a parameter or return value just like any other variable.

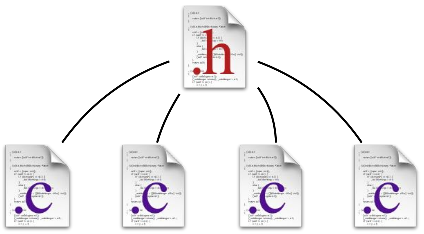
```
complex make_complex(double real, double imag)
{
    complex c;
    c.real = real;
    c.imag = imag;
    return c;
};

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

Header Files

- User-defined types (like „Complex“) are used in many parts of the program
- Goal: Define type once, use as often as desired

Header Files

- A *header file* defines types and functions that are used by multiple parts of the program



defines types

uses types

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);
```

function declarations

A function declaration provides everything except for the function body.

complex.c

makes available the declarations from *complex.h*

```c
#include "complex.h"

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

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

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

user.c

makes available the declarations from *complex.h*

```c
#include "complex.h"

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

Demo

Databases

- Structs often hold data about a process or person.
- Example: personal data

Databases

- Structs often hold data about a process or person.
- Example: personal data

Type definition

```c
struct Person {
  int id;
  char name[60];
  char firstname[60];
  char telephone[40];
};
```

Variable initialisation

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

Databases

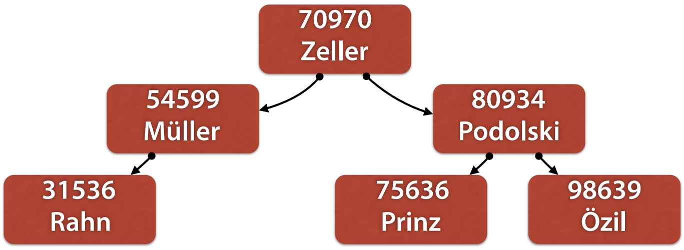
- We could use an array to store large quantities of data
- How many customers will we have?

```c
struct Person {
  int id;
  char name[60];
  char firstname[60];
  char telephone[40];
};

struct Person customers[1000];
```

We could dynamically allocate the array and resize as needed – but then every resize would require an (expensive) copying.

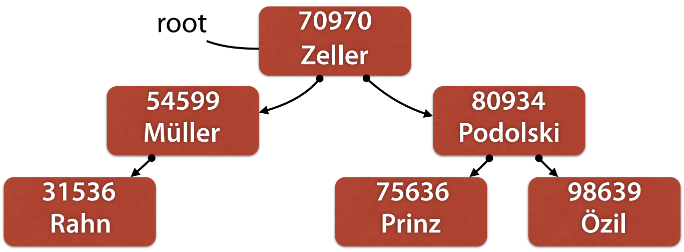
Search Trees

- A search tree is a dynamic data structure for storing and searching of large quantities of data



Search Trees

- Every node has (up to two) children: *in the left subtree* are all smaller values, in the right subtree are all larger values



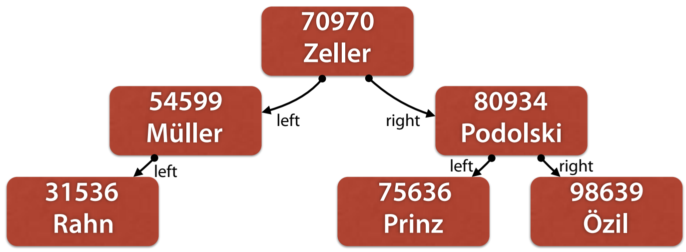
Nodes

- Inside a tree node we store the values –
- and two pointers *to the subtrees*

```
struct Node {
  int id;
  char name[60];
  // more arrays…

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

Searching

- We look for the value x and start in k
- *If* $x < k$.id, search in the left subtree
- If $x > k$.id, search in the right subtree



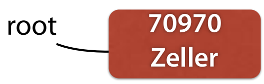
Searching for a Node

- p->elem is the same as (*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;
}
```

Inserting

- We look for the value x and start in k
- If $x < k$.id, insert into the left subtree
- If $x > k$.id, insert into the right subtree

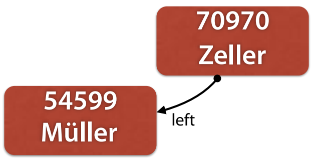


Inserting

- We look for the value x and start in k
- If $x < k$.id, insert into the left subtree
- If $x > k$.id, insert into the right subtree

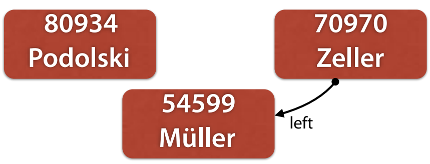
Inserting

- We look for the value x and start in k
- If $x < k$.id, insert into the left subtree
- If $x > k$.id, insert into the right subtree



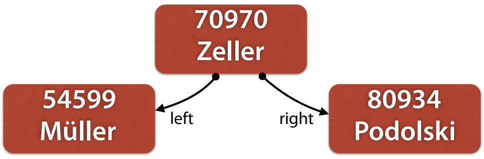
Inserting

- We look for the value x and start in k
- If $x < k$.id, insert into the left subtree
- If $x > k$.id, insert into the right subtree



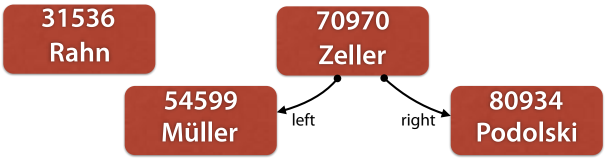
Inserting

- We look for the value *x* and start in *k*
- If *x* < *k*.id, insert into the left subtree
- If *x* > *k*.id, insert into the right subtree



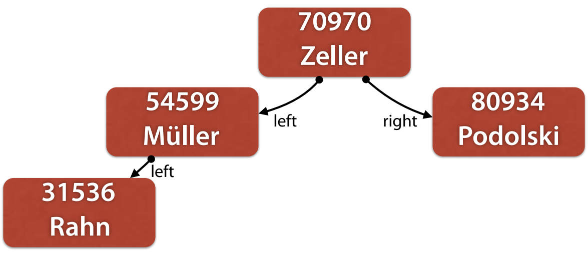
Inserting

- We look for the value *x* and start in *k*
- If *x* < *k*.id, insert into the left subtree
- If *x* > *k*.id, insert into the right subtree



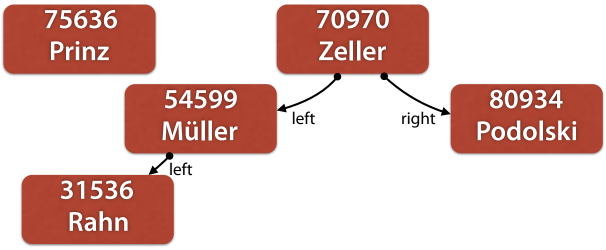
Inserting

- We look for the value *x* and start in *k*
- If *x* < *k*.id, insert into the left subtree
- If *x* > *k*.id, insert into the right subtree



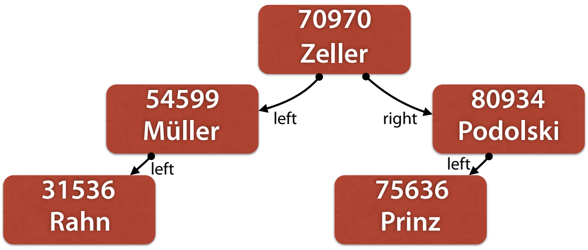
Inserting

- We look for the value *x* and start in *k*
- If *x* < *k*.id, insert into the left subtree
- If *x* > *k*.id, insert into the right subtree



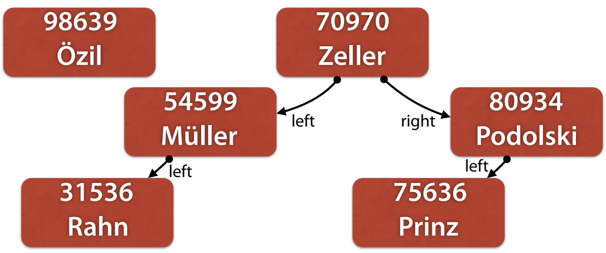
Inserting

- We look for the value *x* and start in *k*
- If *x* < *k*.id, insert into the left subtree
- If *x* > *k*.id, insert into the right subtree



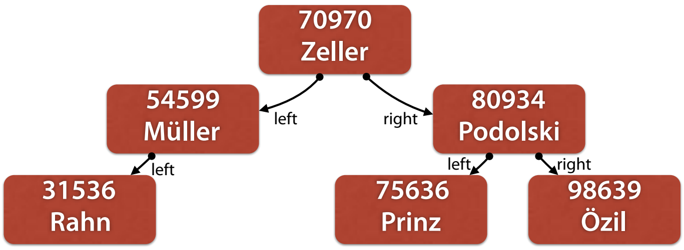
Inserting

- We look for the value *x* and start in *k*
- If *x* < *k*.id, insert into the left subtree
- If *x* > *k*.id, insert into the right subtree



Inserting

- We look for the value *x* and start in *k*
- If *x* < *k*.id, insert into the left subtree
- If *x* > *k*.id, insert into the right subtree



Creating Nodes

- Nodes are allocated in *dynamic memory*

```
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) copies up to n characters from t to s. This way, we avoid overflows.

Inserting Nodes

```
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)
  {
     // analogously for right
  }
}
```

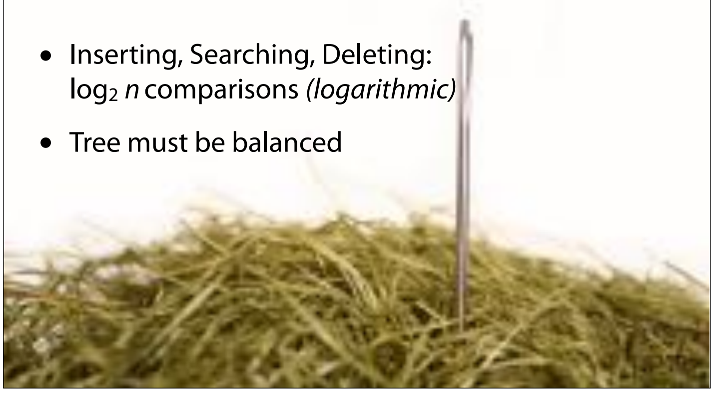
Filling the Tree

```
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;
}
```

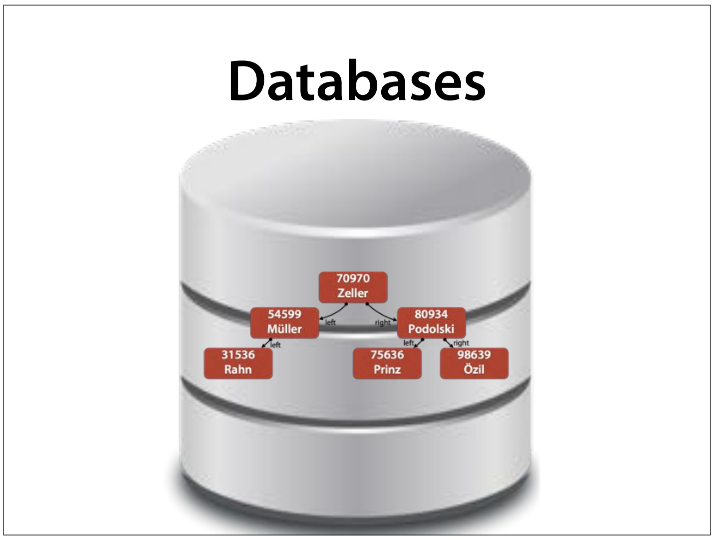
Demo

Complexity

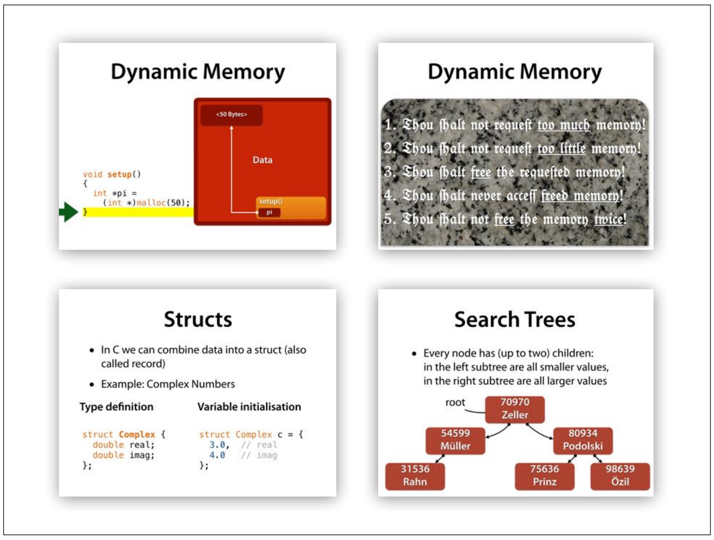
- Inserting, Searching, Deleting: $\log_2 n$ comparisons *(logarithmic)*
- Tree must be balanced



Search Trees are very efficient: All important operations scale.



Whenever you use a database – internally, it uses search trees for indexing.



And we're done :-)