KUAS
KYOTO UNIVERSITY OF ADVANCED SCIENCE
京都発世界人財

京都先端科学大学

Information Processing 2 12

`sizeof`, `typedef`, bit fields, unions
data structures: arrays vs. lists and trees
recursive structures: singly linked list, ordered binary tree

Ian Piumarta

Faculty of Engineering, KUAS

last week: structures

`point` structure
`rect` structure
`point` and `rect` functions
pointers to structures and `(*x).y` vs. `x->y`
structure for dynamic strings
arrays of structures
structure for counting words
selection sort

Chapter 6. Structures
6.1 Basics of Structures
6.2 Structures and Functions
6.3 Arrays of Structures
6.4 Pointers to Structures

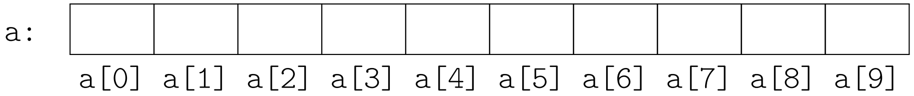
review: arrays

arrays are sequences of values stored contiguously in memory

consider an array of values of some given *type*

    type `a[10];`

`a` is a *constant* equal to the location in memory where the array begins



each element is a value of the given *type*

the total size of the array is: `sizeof(`*type*`) * 10`

review: arrays of arrays

the element *type* can be another array

```
type a[10][3];
```

to decode the type of a, start at the identifier and move outward to the *type*

a is an array of size 10, where each element
a[i] is an array of size 3, where each element
a[i][j] is a value of the given *type*

or imagine where the parentheses would be placed when using a in an expression

```
int a[10][3];     ≡     int (a[10])[3];
```

therefore…

a is an array of 10 arrays of 3 `int`s
a[i] has indexes i from 0 to 9 and elements of type ‘array of 3 `int`’
a[i][j] has indexes j 0 to 2 and elements of type ‘`int`’

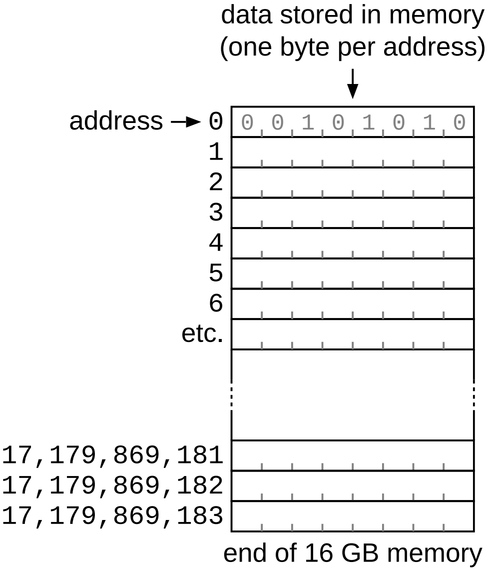
review: computer memory

memory is a (huge!) array of bytes

each byte in memory has a unique *address*, which works exactly like an array index

the addresses are consecutive

adding 1 to any address gives the address of the next byte in the memory



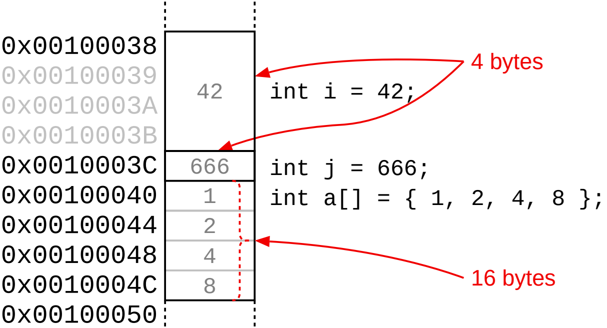
review: computer memory

a 32-bit `int` occupies four bytes of memory

consecutive `int`s have addresses that differ by 4

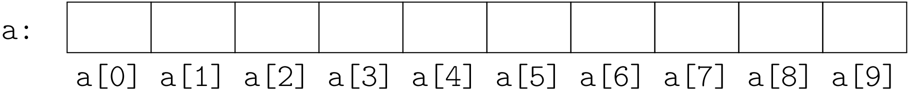
an array of `int` occupies 4 times as many bytes of memory as it has elements

similarly for:	`short`	2 bytes per value
	`long`	8 bytes
	`float`	4 bytes
	`double`	8 bytes



review: arrays and pointers

a sequence of 10 `int` values starting at address `a`



a pointer is an address in memory where a sequence of values begins

an array is the same thing as a pointer to its first element

```
int a[10];        int *p = a; // p is a pointer to int(s)
```

```
a  ==  p  ==  &a[0]
```

adding n to the address of an array gives the address of the n^{th} element

```
a + 3  ==  p + 3  ==  &a[3]
```

review: addresses, indexing, and indirection

'&x' is the address of the variable x

it behaves exactly like an array of size 1 containing x

```
int    x = 42;     // x is 42
int *px = &x;      // a pointer to x (a sequence of 1 int)
*px      = 123;    // x is now 123
px[0]    = 666;    // x is now 666

int i = ++*px;     // increments x and sets i to the result
int j = (*px)++;   // sets j to x and then increments x

int k = *px++;     // sets k to x then increments px (!!!)
                   // px now points to the integer after x

int l = *(px++);   // equivalent to the previous expression
```

review: pointers and auto increment/decrement

consider a string: `char s[32], *ps = s, *t = "hello, world";`

when copying a string, incrementing the pointer is the desired behaviour

```
while ((*ps = *t)) { // copy one character
  ++t;               // t points to the next character in the 'input'
  ++ps;              // s points to the next character in the 'output'
}
```

the precedence rules help you write this concisely:

```
while ((*ps++ = *t++)) ; // copy one character, advance to next
```

review: structures

a `struct` is a type that collects several related values together

individual members are accessed using the ‘`.`’ operator

```
struct Student {
  char  name[50];  // e.g: "Fred Flintstone"
  int   year, id;  // e.g: 2019, 123
  float gpa;       // e.g: 4.9 (yay!)
};

struct Student fred, wilma, barney, betty;

{
  strcpy(barney.name, "Barney Rubble"); // copy string into array
  barney.year = 2018;
  ...
}
```

review: pointers to structures

pointers to structures are used often, especially when the structure is big

```
printStudent(struct Student *ps) // pointer to student
{
  printf("%04dm%03d %s\n", (*ps).year, (*ps).id, (*ps).name);
}
```

the parentheses are needed because the precedence of '`.`' is higher than '`*`'

the pattern `(*`*pointerToStruct*`).`*member* is so common that it has its own syntax:

`ps->member` ≡ `(*ps).member`

```
printStudent(struct Student *ps) // pointer to student
{
  printf("%04dm%03d %s\n", ps->year, ps->id, ps->name);
}
```

review: allocating memory to hold arrays or structures

the `sizeof` operator tells you how many bytes are in a value or type

```
int x, a[10];
struct Student s, *ps;

sizeof(int);            // usually 4 bytes
sizeof(x);              // same as previous line

sizeof(a);              // size of 10 ints, usually 40 bytes

sizeof(struct Student); // number of bytes in the entire structure
sizeof(s);              // same as previous line

sizeof(*ps);            // same as previous line
sizeof(ps);             // size of a pointer, usually 8 bytes

int *pi = malloc(sizeof(int) * 10);  // space for (an array of) 10 ints

ps = malloc(sizeof(struct Student)); // space for one Student
ps = malloc(sizeof(*ps));            // same as previous line
```

test

this week: recursive structures, bit fields, unions

Chapter 6. Structures
6.5 Self-referential Structures
6.6 Table Lookup
6.7 Typedef
6.8 Unions
6.9 Bit-fields

`sizeof()` and counting number of array elements
pointer version of `findWord()`
data structure: binary tree or linked list
using `typedef` for nodes
unions and bit fields

sizeof

`sizeof(x)` tells you the size of `x`, in bytes

`x` can be a constant expression or a type

```
sizeof(char);   // usually 1
sizeof(int);    // usually 4

char c[100];
int  i[100];

sizeof(c);      // 100        == sizeof(char) * 100
sizeof(i);      // 400        == sizeof(int)  * 100
sizeof(*i);     // 4          == sizeof(int)
```

size of an array vs. number of elements

given a declaration

```
type array[10];
```

then

```
sizeof(*a)                // =      sizeof(type)
sizeof(a)                 // = 10 x sizeof(type)
sizeof(a) / sizeof(*a)    // = 10
sizeof(a) / sizeof(a[0])  // = 10
```

a very useful macro is therefore

```
#define indexableSize(ARRAY)  (sizeof(ARRAY) / sizeof(*(ARRAY)))

indexableSize(a)          // = 10
```

type of `sizeof`

the result of `sizeof` is an integer

it *must* be large enough to represent the largest possible object/array

the type of the result of `sizeof` is therefore machine dependent

small microcontroller	16 bits	`short`
powerful microcontroller	32 bits	`int`
laptop, desktop	64 bits	`long`

`<stddef.h>` includes a definition `size_t` that is correct for the local machine

- the suffix `_t` means “type”
- this naming convention is widely used by C programmers

to print a value of type `size_t` use the `printf` format `"%zu"` (unsigned decimal)
or `"%zx"` (unsigned hexadecimal)

note that `size_t` is *unsigned*

- a `size_t` value can never be negative (less than zero)

using `size_t` as the type of an array index

`size_t` is often used as the type of an array index

- guaranteed never to overflow, no matter how large the array
- but the index can *never* be negative

some algorithms might require an array index to become negative

- e.g., to terminate a loop counting down

```
while (--index >= 0) { ... }
```

for these algorithms there is a *signed* version called `ssize_t`

additional `s` means *signed* size type; print it using '`%zd`'
↓

```
ssize_t index = indexableSize(array);
while (--index >= 0) // index must be signed!
  printf("%3zd: %d\n", index, array[index]);
```

alignment size of a structure

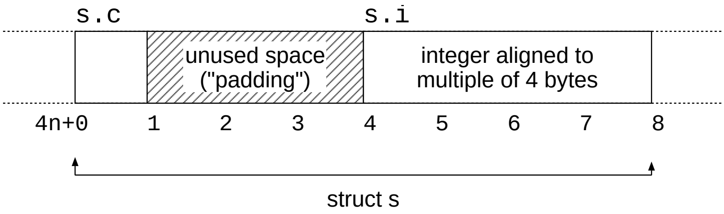
usually every value has to be *aligned* when stored in memory

- it can only be stored at an address that is **a multiple of its size**

this can lead to gaps between successive variables in memory

- particularly between members in a `struct`

```
struct {
  char c;   ← alignment 1
  int  i;   ← alignment 4
} s;        ← alignment 4
```




```
printf("%zd\n", sizeof(struct s));     // => 8
```

the alignment of an entire structure is equal to the largest alignment of its members

unions

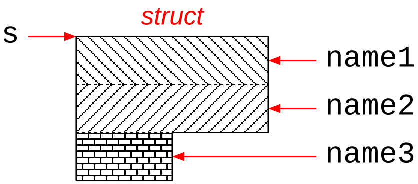
`struct`: members are located at consecutive (aligned) memory addresses

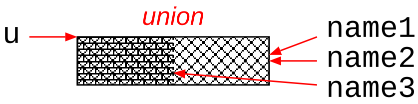
`union`: members are located at **the same memory** address

unions save space, allow changes of representation, or implement dynamically typed objects

```
struct {
  int   name1;
  int   name2;
  short name3;
} s;
```

```
union {
  int   name1;
  int   name2;
  short name3;
} u;
```





bit fields

one use of a `struct` is to describe a hardware device control register

- e.g., the communication speed of a serial interface

these registers typically have many parameters packed into a single `int` or `short`

RXC	TXC	DRE	ERR	-	rate (11 bit clock divisor)

- receive complete, transmit complete, data register empty, error, unused
- 11-bit divisor to set communication rate by dividing processor clock rate

```
struct {
  unsigned      rxc : 1, txc : 1, dre : 1, err : 1; // single-bit flags
  unsigned     _pad : 1;                             // unused bit
  unsigned  divisor : 11;                            // clock divisor 1..2047
};
```

note: almost every aspect of bit fields (order, alignment, etc.) is *machine-dependent*

typedef

`typedef` lets you give a name to an existing type

- it *does not* create a new type, just defines a name for an existing type

if `typedef` appears in a declaration then all the names in the declaration name types

- instead of naming variables

```
typedef int Integer;

Integer i, j;                  // i, j have type 'int'

typedef char *String;

String s = "hello, world"; // s has type 'char *'

typedef struct Pt { int x, y; } Point, *PointPtr; // *two* new type names

Point    p = { 3, 4 };         // type 'struct Pt'
PointPtr q = &p;               // type 'struct Pt *'
```

simplest data structure: the array

arrays are a fundamental building block for many data structures

data	data	data	data	data	data	data	data	data	data
0			$i-1$	i	$i+1$				$n-1$
			previous	element	next				

data values are

- stored in consecutive memory locations
- identified by a numerical *index* starting at `0`

lookup operations are fast

order of items is *implied* by their adjacency in the array

operations that rearrange data or insert new data are slow

- might have to copy many or even all elements

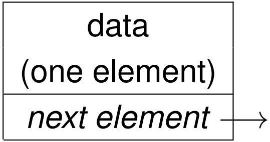
nodes are individual data structure elements

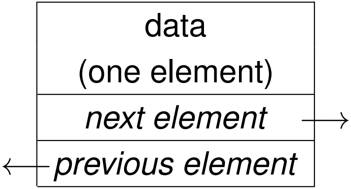
a *node* is a small block of memory identified by its address

one node contains data for one element of the structure

- sometimes called the *payload* of the node

the order of nodes in the data structure is made *explicit* using pointers to other nodes

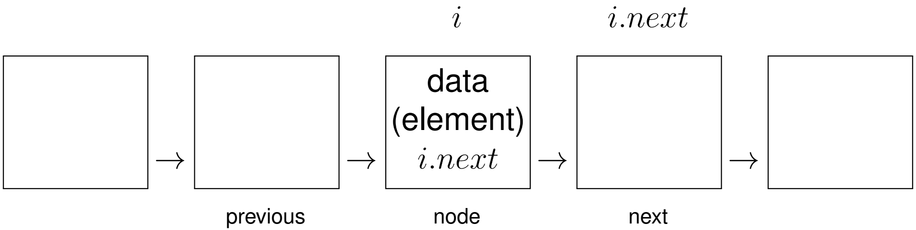




node-based data structure: linked list

a linked list is a linear structure, like an array

- elements are nodes, stored as *separate objects* in memory
- the order of elements is made *explicit* using a pointer to the next element
- the actual order in memory is unknown (and irrelevant)

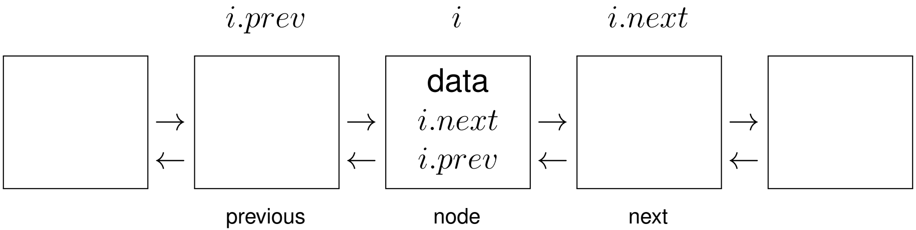


a linked list can be traversed (easily) in **one direction only**

node-based data structure: doubly linked list

a doubly linked list adds another pointer, back to the previous element

- the order of elements is *explicit* using pointers to previous and next elements
- the ‘next’ pointers are often called *forward links*
- the ‘previous’ pointers are often called *backward links*



a doubly linked list can be traversed (easily) in **both directions**

self-referential structures: linked list

each node in a list looks the same and contains

- the **data** for the node
- a pointer to the **next** node, or NULL (zero)

```
typedef struct Node Node;

struct Node {
  int data;
  Node *next;
};
```

the structure is therefore *recursive*

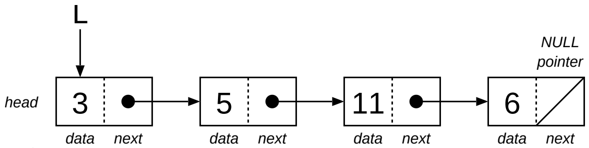
- the **next** pointer refers to another list
- (often recursive functions are best for processing recursive structures)

the end of the list is marked with a NULL **next** pointer

- its numerical value is zero

an entire list L is represented by a pointer to the first element in the list

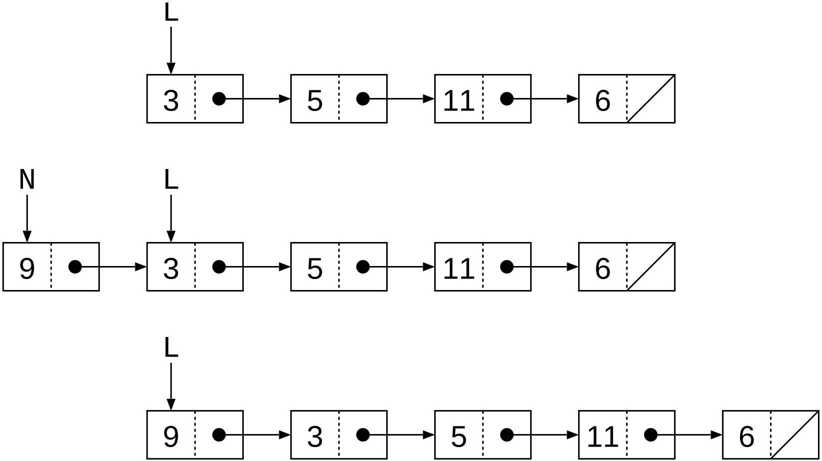
- this is often called the 'head' of the list (think: 'line of people', or 'snake')



operations on a linked list: add

add a data element 9 to the head of a list L:

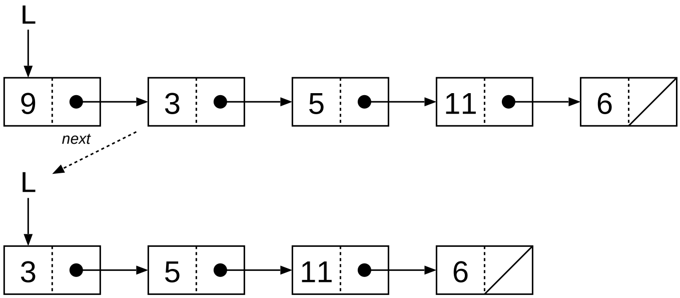
- create a new node N with **data** 9 and **next** pointer L
- set L to N (the new node becomes the new head of the list)



operations on a linked list: remove

remove the first element in the list:

- set L to L.*next* (the second node becomes the new head of the list)



C `struct` for a linked list

```
typedef struct Node Node;

struct Node {
  int   data;
  Node *next;
};

Node *newNode(int data, Node *next)
{
  Node *node = malloc(sizeof(Node));  // new memory for node
  node->data = data;
  node->next = next;
  return node;
}
```

note: `malloc(sizeof(*node))` would be safer

operations on lists of `Nodes`

```
Node *addFirst(Node *list, int data)
{
  Node *node= newNode(data, list);
  return node;
}

Node *removeFirst(Node *list)
{
  return list ? list->next : 0;
}
```

note the use of the ‘null’ pointer `0` to represent an ‘illegal’ or ‘undefined’ pointer

- C guarantees there is never valid memory at address `0`
- attempts to access it cause run-time errors (segmentation fault or bus error)

using the list operations

```c
Node *list = 0;

for (int i = 0;  i < 10;  ++i)
  list = listAddFirst(list, i);

while (list) {
  printf("%2d", list->data);
  list = listRemoveFirst(list);
}
printf("\n");
```

having to assign to `list` is troublesome

- forgetting to assign to `list` could cause catastrophic run-time errors

improving the list operations

solution: pass a *pointer* to the list so the operations can assign to it

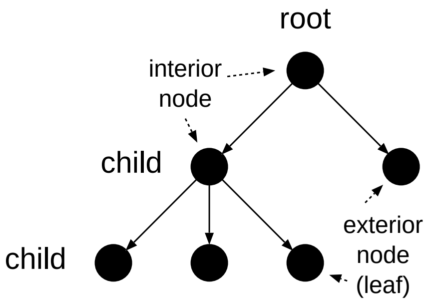
```
void listAddFirst(Node **list, int data)
{
  *list = newNode(data, *list);
}

int listRemoveFirst(Node **list)
{
  if (0 == *list) return 0;
  int data = (*list)->data;
  Node *next = (*list)->next;
  free(*list);
  *list = next;
  return data;
}

int listEmpty(Node **list)
{
  return 0 == *list;
}
```

```
int main()
{
  Node *list = 0;

  for (int i = 0;  i < 10;  ++i)
    listAddFirst(&list, i);

  while (!listEmpty(&list))
    printf("%2d", listRemoveFirst(&list));
  printf("\n");

  return 0;
}
```

node-based data structure: tree

a *tree* structure looks like a tree

- there is a *root* that is the “start” node of the tree
- each node can have pointers to “child” nodes

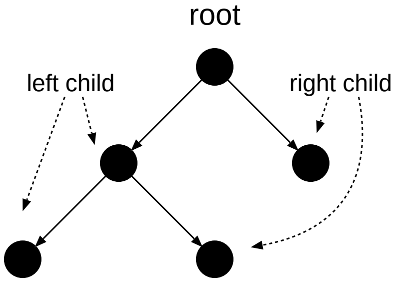


a node’s left and right sub-trees are usually related to that node in specific ways

node-based data structure: binary tree

```
typedef struct Node Node;

struct Node {
  int   data;
  Node *left;  // child
  Node *right; // child
}
```

a *binary tree* is a tree structure where each node has at most two *children*



each node has at most one *parent*

two nodes with a common parent are called *siblings*

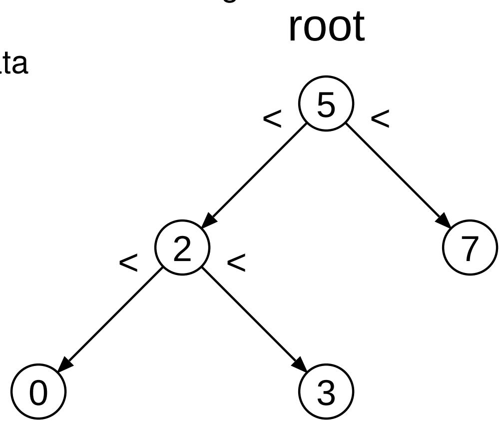
a child can be an entire *sub-tree* of any complexity

the left and right sub-trees are often related to the parent in a specific way

node-based data structure: ordered binary tree

in an *ordered binary tree*

- each parent node contains some data, e.g., a number or a string
- all data in the left sub-tree is $<$ the parent's data
- all data in the right sub-tree is $>$ the parent's data



to insert new data into an ordered binary tree:

- start at the root
- branch left if new data $<$ current node
- branch right if new data $>$ current node
- when 'missing' node encountered:
  - insert node with new data

to process data in ascending order:

- process entire left sub-tree
- process data in current node (e.g., print it)
- process entire right sub-tree

C `struct` for binary tree node

```
struct Node
{
  int   number;          // node data
  Node *left, *right;   // two child nodes
};

Node *newNode(char *word)
{
  Node *node= malloc(sizeof(Node));  // new memory for node
  node->word  = strdup(word);        // private copy of string
  node->left  = node->right = 0;     // no children
  return node;
}
```

note: `malloc(sizeof(*node))` would be safer

find node

```
Node *findNode(Node *nodep, int data)
{
  if (!nodep) // no node stored at this location
    return 0;

  if (data < nodep->data) // search left sub-tree
    return findNode(nodep->left,  data);

  if (data > nodep->data) // search right sub-tree
    return findNode(nodep->right, data);

  return nodep;           // found the data
}
```

same issue we had earlier:

- to insert a new node, have to assign to the location where NULL pointer is stored
- solution: pass a *pointer* to the location where the node pointer is stored

improved find node

```
Node *findNode(Node **nodepp, int data)
{
  Node *nodep = *nodepp;                  // get node pointer
  if (!nodep)                             // node missing
    return *nodepp = newNode(word);       // insert node into tree
  if (data < nodep->data)                 // search left sub-tree
    return findNode(&nodep->left,  data);
  if (data > nodep->data)                 // search right sub-tree
    return findNode(&nodep->right, data);
  return nodep;
}
```

printing the tree in ascending order

at a particular node

- everything in the left sub-tree is smaller
- everything in the right sub-tree is larger

```
void printTree(Node *nodep)
{
  if (!nodep) return;            // bottom of tree
  printTree(nodep->left);        // print all smaller values
  printf("%d\n", nodep->data);   // print the value
  printTree(nodep->right);       // print all larger values
}
```

next week: input and output

`getchar()`, input redirection, pipes
`putchar()`, output redirection, pipes
`lower()`
`printf()`
variadic functions
home-made `minprintf()`
`scanf()`
`FILE` access, `stdin`, `stdout`
writing the `cat` program
`stderr` and `exit()`
line input and output
string, char, memory functions

assignment

please download assignment from

MS Team “Information Processing 2”, “General” channel

exercises

create a node for storing words in a binary tree

create a constructor function for a node

write findNode(Node **nodepp, char *word)

modify findNode(Node **nodepp, char *word) to create the tree

write printTree(Node *nodep)

add getword() from last week and read words from input into tree

12:40	5	`sizeof`
12:45	5	size of an array vs. number of elements
12:50	5	type of `sizeof`
12:55	5	alignment size of a structure
13:00	5	bit fields
13:05	5	`typedef`
13:10	5	simplest data structure: the array
13:15	5	node-based data structure: linked list
13:20	5	node-based data structure: doubly linked list
13:25	5	self-referential structures: linked list
13:30	5	operations on a linked list: add, remove
13:35	5	C code for a linked list
13:40	5	improving the list operations
13:45	5	node-based data structure: tree, binary tree, ordered binary tree
13:50	5	C `struct` for binary tree node
13:55	5	find node
14:00	5	improved find node
14:05	5	printing the tree in ascending order
14:10	10	*break*
14:20	90	exercises
15:50	00	end