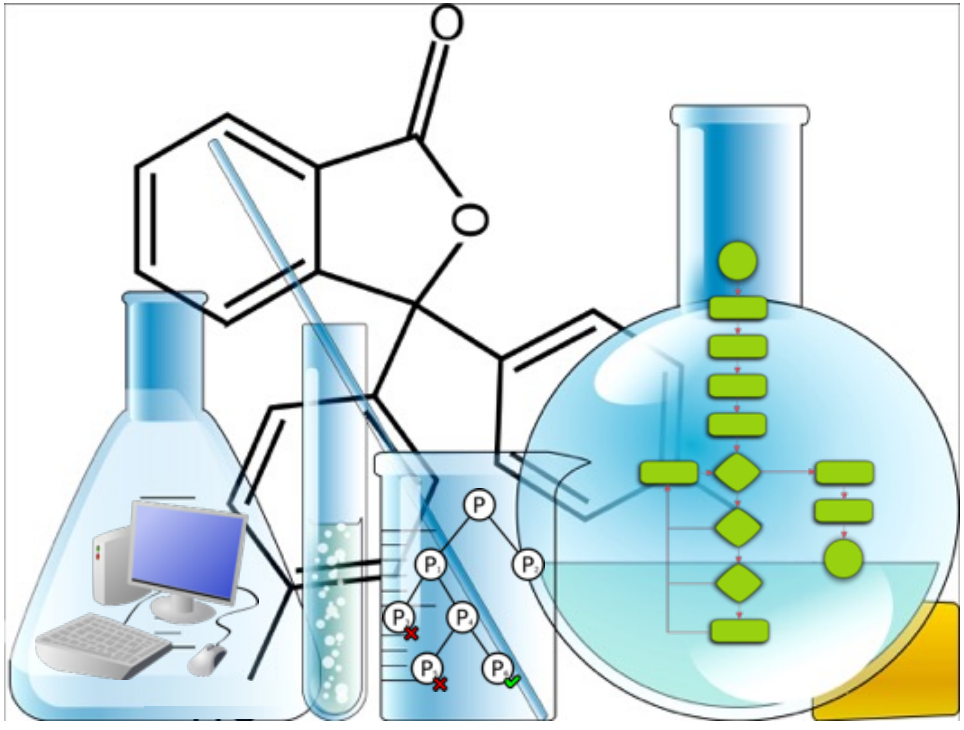
# Labor Angewandte Informatik

Stephan Schulz
stephan.schulz@dhbw-stuttgart.de

Jan Hladik
jan.hladik@dhbw-stuttgart.de




DHBW Stuttgart
Duale Hochschule
Baden-Württemberg

# Inhaltsverzeichnis

# Ziele für Woche 1

- Übersicht über das Labor
- Erarbeiten einer gemeinsamen Basis
  - Praktischer Umgang mit dem C-Compiler und Toolchain
  - Kommandozeilen und Ein-/Ausgabe
- Drei einfache Programme

# Rechnerausstattung

- Für die praktischen Übungen brauchen Sie eine Umgebung, mit der Sie C-Programme entwickeln und ausführen können. Beispiele werde für UNIX (Linux/MacOS-X) angeboten. Sie können eine aktuelle Linux-Installation unter einer virtuellen Maschine installieren.
  - Z.B. VirtualBox (`https://www.virtualbox.org`)
  - Betriebssystem: Z.B.. Ubuntu (`http://www.ubuntu.com`)
  - Paketmanager für OS-X: Fink (`http://fink.sourceforge.net`) oder MacPorts (`https://www.macports.org`)
  - unter Windows:
  - Cygwin (`http://www.cygwin.com`)
- Compiler:
  - `gcc` (Unter Linux in der Regel bereits installiert)
  - Unter OS-X: X-Code und die *Command Line Tools* installieren LLVM mit `gcc` Front-End
- Text-Editor:
  - Klassisch: `vi/vim`
  - Unschlagbar: `emacs`
  - Einfach/modern: `gedit`
  - Windows: Notepad++
  - Auf OS-X installiert: `Textedit`

# Aufgaben Woche 1

Erstellen Sie Programme für die folgenden Aufgaben:

- `rand_seq` soll eine Sequenz von zufällig zwischen 0 (einschließlich) und `MAXNUM` (ausschließlich) verteilten Zahlen generieren und ausgeben.
- `double_num` soll Ganzzahlen einlesen, verdoppeln, und wieder ausgeben. Sie können davon ausgehen, dass eine Zahl pro Zeile steht.
- `reverse_str` sollte eine Liste von bis zu `MAXITEMS` Strings einlesen und in umgekehrter Reihenfolge wieder ausgeben (also den letzten String zuerst). Jeder String steht in einer einzelnen Zeile. Strings haben maximal 255 Zeichen.

Verwenden Sie `MAXNUM` gleich 1000 und `MAXITEMS` gleich 10000. Die Anzahl der Zufallszahlen für `rand_seq` sollte auf der Kommandozeile (d.h. als Parameter beim Aufruf) übergeben werden. Für `double_num` und `reverse_str` sollten die Eingabedaten aus einer auf der Kommandozeile spezifizierten Datei ausgelesen werden. Falls keine Name angegeben wird, soll `stdin` gelesen werden.

# Tipps und Hinweise

- Linux/UNIX/OS-X dokumentieren große Teile des Systems im Manual
- `man 3` greift auf die Manualseiten zu den Funktionalitäten der System- und C-Libraries zu
  - Z.B. `man 3 random`, `man 3 perror`, `man 3 fopen`
  - Z.B. `man 3 atoi`, `man 3 exit`, `man 3 strdup`
  - Z.B. `man 3 free`, `man 3 fgets`
- `man` ohne Sektionsauswahl sucht im ganzen Manual
  - Z.B. `man make`
  - Z.B. `man gcc`
  - Siehe auch `man man`

*man-pages* sind von Experte für Experten geschrieben, und oft sehr dicht. Bei Seiten zu UNIX-Befehlen hilft oft der Abschnitt `EXAMPLES` weiter.

# Feedback/Abschluss

- Erfolg?
- Schwierigkeiten?
- Sonstiges?

# Ziele für Woche 2

- Umgang mit Pointern und dynamischem Speicher
- Implementierung eines Datentyps “Lineare Liste”
- Kleines Testprogram dazu

# Dynamischer Speicher

- C Programme verwenden 3 Arten von Speicher:
  - Statischen Speicher (im Data Segment, typischerweise globale Variablen)
  - Automatischen Speicher (auf dem Stack, lokale Variablen)
  - Dynamischen Speicher (auf dem Heap, dynamisch verwaltet)
- Verwendung von dynamischem Speicher:
  - Langlebige Objekte
  - Große Objekte
  - Objekte unvorhersehbarer Größe

# Dynamischer Speicher

- C Programme verwenden 3 Arten von Speicher:
  - Statischen Speicher (im Data Segment, typischerweise globale Variablen)
  - Automatischen Speicher (auf dem Stack, lokale Variablen)
  - Dynamischen Speicher (auf dem Heap, dynamisch verwaltet)
- Verwendung von dynamischem Speicher:
  - Langlebige Objekte
  - Große Objekte
  - Objekte unvorhersehbarer Größe

**Speziell: Dynamische Datenstrukturen**

# Dynamischer Speicher

- Dynamischer Speicher
  - Auf dem Heap von der libc-Speicherverwaltung verwaltet
  - Zugriff über Pointer
  - Mit `malloc()` beschafft
  - Mit `free()` zur Wiederverwendung zurückgegeben

# Dynamischer Speicher

- Dynamischer Speicher
  - Auf dem Heap von der libc-Speicherverwaltung verwaltet
  - Zugriff über Pointer
  - Mit `malloc()` beschafft
  - Mit `free()` zur Wiederverwendung zurückgegeben

**Anwendung heute: Dynamische Datenstrukturen**

# Datenstrukturen/Datentypen

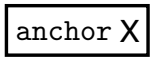
- Die Implementierung eines Datentyps besteht aus
  - Den notwendigen Strukturobjekten (Records oder Structs)
    - Beschreiben, welche Daten von welchem Typ verarbeitet werden
  - Einer Reihe von Funktionen auf diesen Strukturen
  - Typische Beispiele:
    - Anlegen eines neuen Elementes
    - Freigeben eines Elementes
    - Einfügen eines neuen Elements
    - Löschen eines Elements
    - Finden eines Elements
    - Sortieren
    - ...

# Lineare Listen

- Lineare Listen sind ein Beispiel für einfache dynamische Datenstrukturen
- Heute: Einfach verkettete Listen
  - Eine Liste besteht aus Listenzellen
  - Jede Listenzelle hat zwei Felder:
    - Den gespeicherten Wert (`payload`)
    - Einen Pointer auf den Nachfolger (`next`)
- Listen werden durch Pointer repräsentiert und sind rekursiv:
  - Der NULL-Pointer repräsentiert eine Liste (die *leere Liste*)
  - Ein Pointer auf eine Listenzelle repräsentiert eine Liste, wenn deren `next` Pointer eine Liste repräsentiert

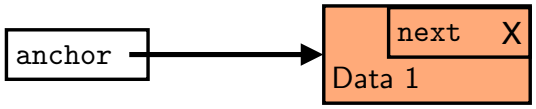
# 1000 Worte

## Leere Liste (Länge 0)

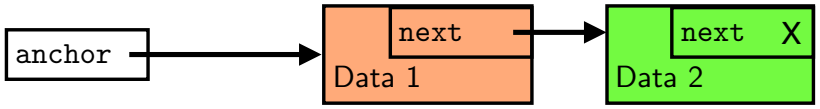



Für die Implementierung in C:
X symbolisiert den NULL-Pointer

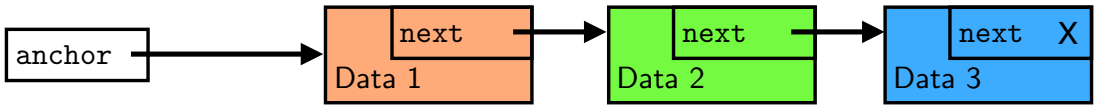
## Liste mit einem Eintrag




## Liste mit zwei Einträgen




## Liste mit drei Einträgen




o o o

# Konkreter Listentyp

```
/* Data type: Linear list cell */

typedef struct lin_list
{
    char            *payload; /* User data (in this case
                                 just strings) */
    struct lin_list *next;    /* Pointer to rest of list
                                 (or NULL) */
}LinListCell, *LinList_p;
```

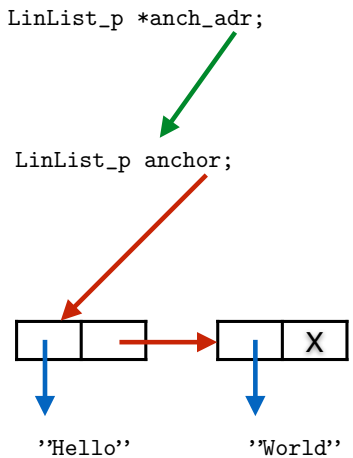
# Konkreter Listentyp

```
/* Data type: Linear list cell */

typedef struct lin_list
{
   char           *payload; /* User data (in this case
                               just strings) */
   struct lin_list *next;   /* Pointer to rest of list
                               (or NULL) */
}LinListCell, *LinList_p;
```

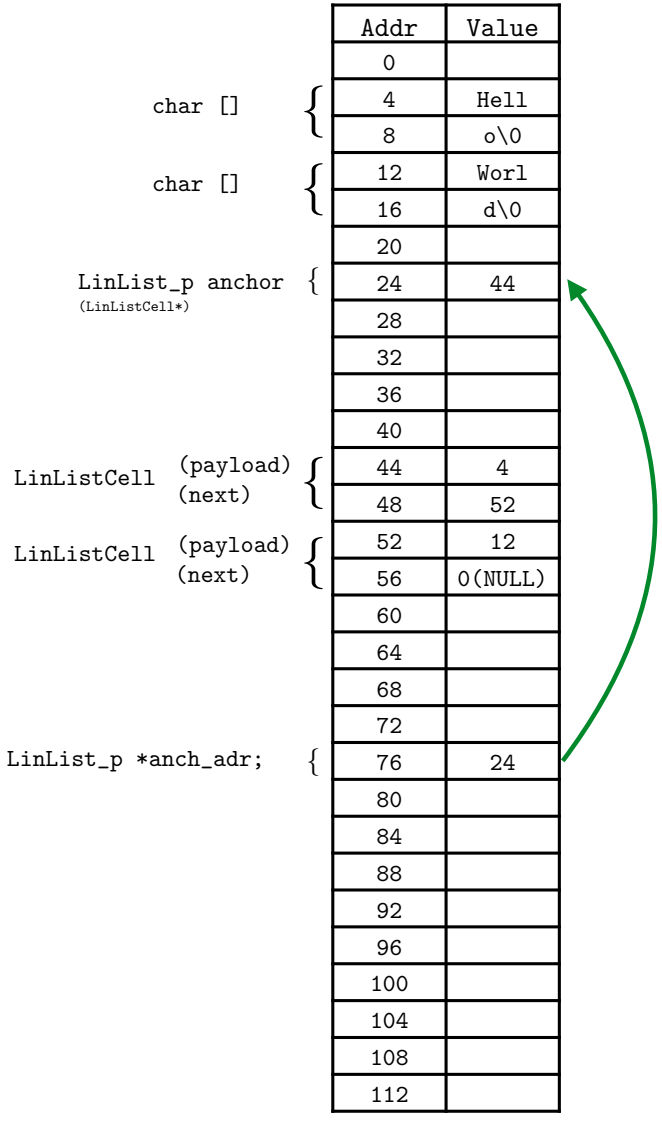
- Tipp: Listen brauchen praktisch einen Anker
  - Variable vom Typ `LinListCell*` oder `LinList_p`
  - Für Operationen, die die Liste verändern wird ein Pointer auf den Anker übergeben (Z.B. `&anker` vom Typ `*LinList_p`)

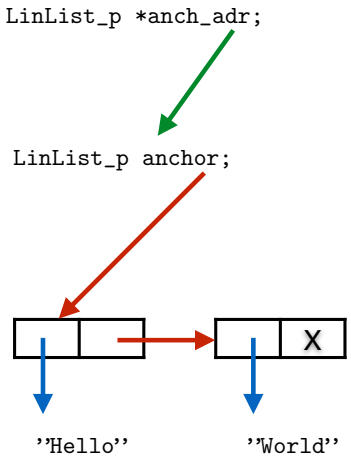
# Listen im Speicher

| | | Addr | Value |
|---|---|---|---|
| | | 0 | |
| char [] | { | 4 | Hell |
| | | 8 | o\0 |
| char [] | { | 12 | Worl |
| | | 16 | d\0 |
| | | 20 | |
| LinList_p anchor (LinListCell*) | { | 24 | 44 |
| | | 28 | |
| | | 32 | |
| | | 36 | |
| | | 40 | |
| LinListCell (payload) | { | 44 | 4 |
| LinListCell (next) | | 48 | 52 |
| LinListCell (payload) | { | 52 | 12 |
| LinListCell (next) | | 56 | 0(NULL) |
| | | 60 | |
| | | 64 | |
| | | 68 | |
| | | 72 | |
| LinList_p *anch_adr; | { | 76 | 24 |
| | | 80 | |
| | | 84 | |
| | | 88 | |
| | | 92 | |
| | | 96 | |
| | | 100 | |
| | | 104 | |
| | | 108 | |
| | | 112 | |

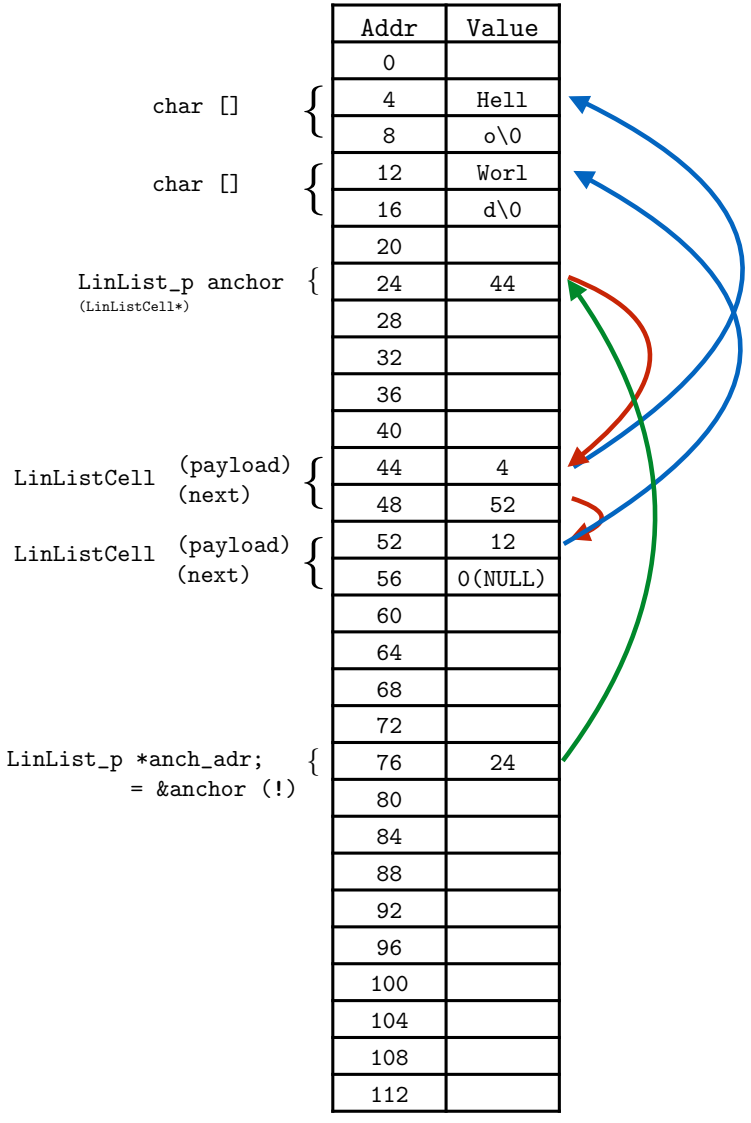


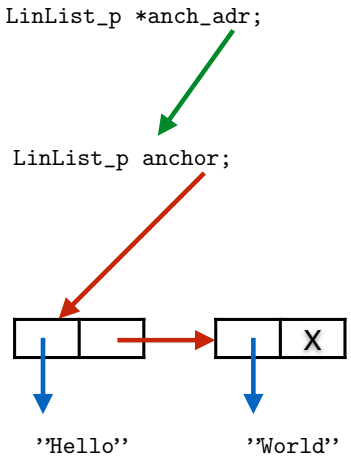
# Listen im Speicher



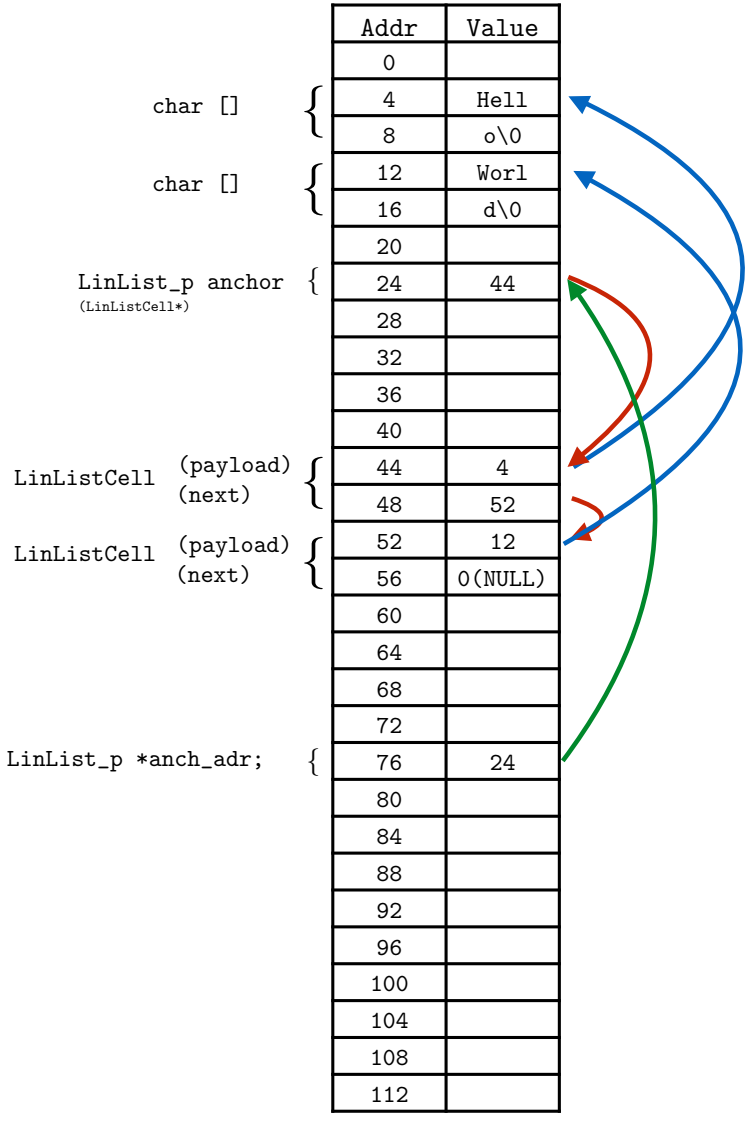



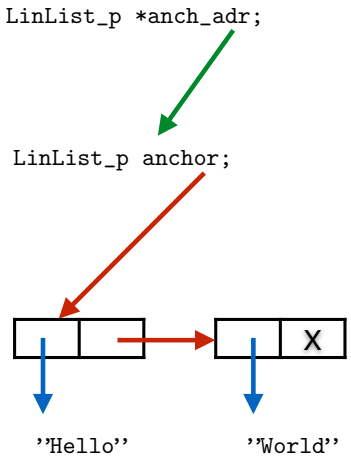
# Listen im Speicher



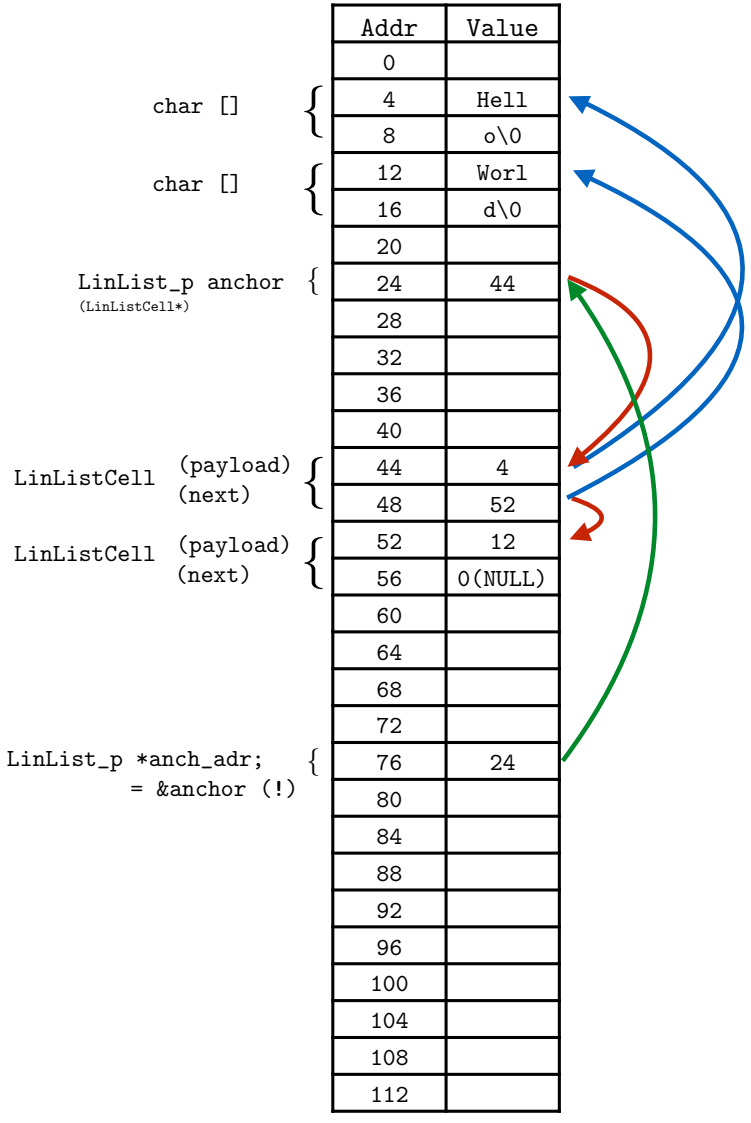



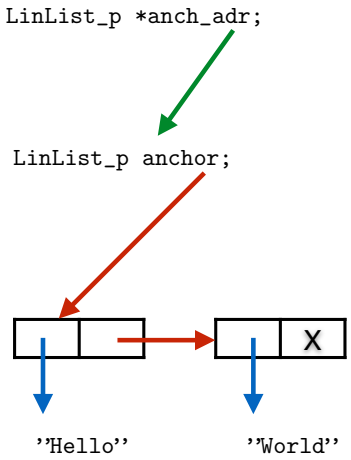
# Listen im Speicher



# Listen im Speicher



# Listenfunktionen (Speicherverwaltung)

`LinList_p LinListAllocCell(char* payload);`

- Erzeugt eine Listenzelle, deren Payload eine Kopie von `payload` ist und gibt einen Pointer auf diese zurück (Hinweis: `malloc()`, `strdup()`) (Hinweis 2: Es ist sinnvoll, `next` auf `NULL` zu setzen)

`void LinListFreeCell(LinList_p junk);`

- Gibt eine einzelne Listenzelle und den gespeicherten String an die Speicherverwaltung zurück (Hinweis: `free()`)

`void LinListFree(LinList_p *junk);`

- Gibt die gesamte Liste mit allen Zellen an die Speicherverwaltung zurück und setzt den Listenanker auf NULL

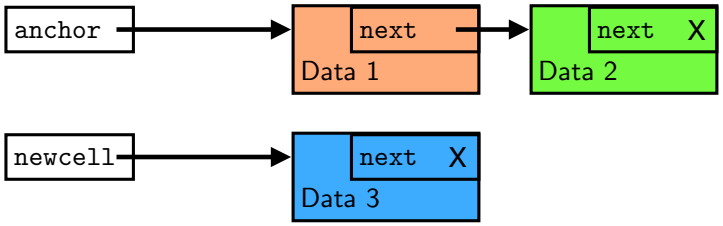
# Einfügen (Vorne)

```
LinList_p LinListInsertFirst(LinList_p *anchor, LinList_p
newcell);
```

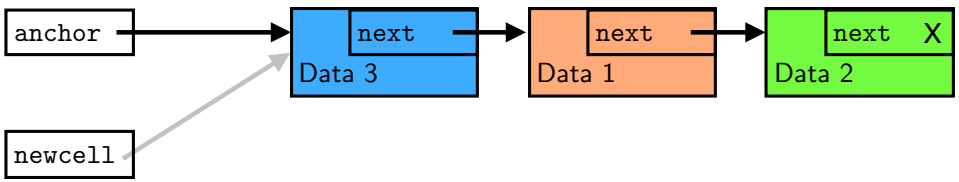
- Erweitert die Liste, die bei `*anchor` steht um das neue Element `newcell`
  - `*anchor` zeigt danach auf `newcell`
  - `newcell->next` zeigt auf den alten Wert von `*anchor`
  - Rückgabewert ist Pointer auf die gesamte Liste

Für lineare Listen ist das Einfügen und Ausfügen am Anfang der Liste typisch - vergleiche `cons` und `car/cdr` in Scheme

# Einfügen (Vorne)




```
LinListInsertFirst(&anchor, newcell);
```

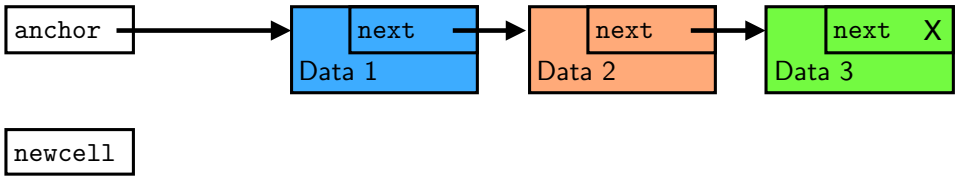


# Ausfügen

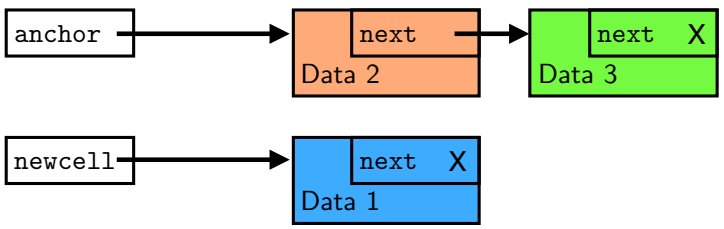
```
Linlist_p LinListExtractFirst(LinList_p *anchor);
```

- Entferne das erste Element der Liste, die bei `*anchor` steht
  - Voraussetzung: Die Liste ist nicht leer (sonst: Rückgabewert `NULL`)
  - `*anchor` zeigt danach auf das vormals zweite Element der Liste
  - Rückgabewert ist Pointer auf das vormals erste Element

# Ausfügen




```
newcell = LinListExtractFirst(&anchor);
```



# Aufgaben Woche 2

- Implementieren Sie einen Datentyp lineare Liste zum Speichern von Listen von Strings
- Implementieren Sie dazu mindestens folgende Funktionen:

```
LinList_p LinListAllocCell(char* payload);
void LinListFreeCell(LinList_p junk);
void LinListFree(LinList_p *junk);
LinList_p LinListInsertFirst(LinList_p *anchor,
                             LinList_p newcell);
Linlist_p LinListExtractFirst(LinList_p *anchor);
```

- Verwenden Sie lineare Listen, um ein Programm zu schreiben, das:
  - Beliebig viele Strings (bis 255 Zeichen Länge) einliest
  - Diese in umgekehrter Reihenfolge wieder ausgibt
  - Diese danach in Originalreihenfolge wieder ausgibt
- Stellen Sie sicher, dass der gesamte allokierte Speicher auch wieder frei gegeben wird.

# Beispielausgabe

```
> ./lin_list
ab
ac
bc
def
Reverse order
=============
def
bc
ac
ab
Original order
==============
ab
ac
bc
def
```

# Feedback/Abschluss

- Erfolg?
- Schwierigkeiten?
- Sonstiges?

# Ziele für Woche 3

- Erweiterung des Datentyps “Lineare Liste”
  - Einfügen/Löschen im allgemeinen
  - Finden

# Erinnerung: Lineare Listen

- Lineare Listen sind eine einfache Standard-Datenstruktur
  - Aufgebaut aus gleichartigen Zellen
  - Einfach verkettet (jede Listenzelle zeigt auf ihren Nachfolger, falls es einen gibt)
  - In C: NULL repräsentiert die leere Liste
  - Ein Pointer auf die erste Zelle repräsentiert die gesamte Liste

# Erinnerung: Konkreter Listentyp

```
/* Data type: Linear list cell */

typedef struct lin_list
{
    char            *payload; /* User data (in this case
                                 just strings) */
    struct lin_list *next;    /* Pointer to rest of list
                                 (or NULL) */
}LinListCell, *LinList_p;
```

# Erinnerung: Konkreter Listentyp

```
/* Data type: Linear list cell */

typedef struct lin_list
{
   char            *payload; /* User data (in this case
                                just strings) */
   struct lin_list *next;    /* Pointer to rest of list
                                (or NULL) */
}LinListCell, *LinList_p;
```

- Tipp: Listen brauchen praktisch einen Anker
  - Variable vom Typ `LinListCell*` oder `LinList_p`
  - Für Operationen, die die Liste verändern wird ein Pointer auf den Anker übergeben (Z.B. `&anker` vom Typ `*LinList_p`)

# Weitere Ein-/Ausfügeoperationen

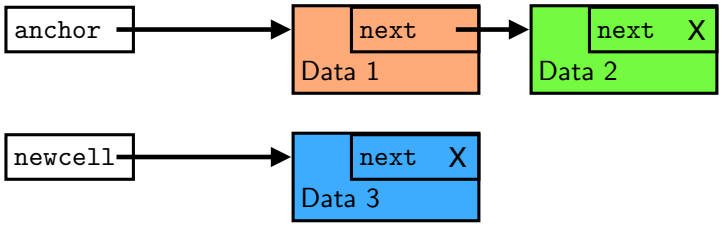
`LinList_p LinListInsertLast(LinList_p *anchor, LinList_p newcell);`

- Hänge `newcell` als letztes Element an die Liste bei `*anchor` an
- Ansonsten wie `LinListInsertFirst()`
- Anmerkug: Das implementiert "nebenbei" append!

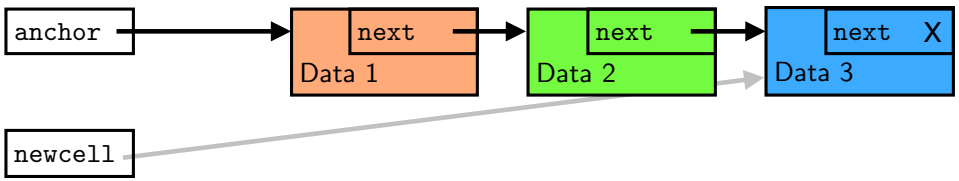
`LinList_p LinListExtractLast(LinList_p *anchor);`

- Entferne das letzte Element der Liste, die bei `*anchor` steht
  - Voraussetzung: Die Liste ist nicht leer (sonst: Rückgabewert `NULL`)
  - `*anchor` bleibt unverändert, es sei denn die Liste hat genau ein Element (dann wird `*anchor` auf `NULL` gesetzt)
  - Rückgabewert ist Pointer auf das vormals letzte Element

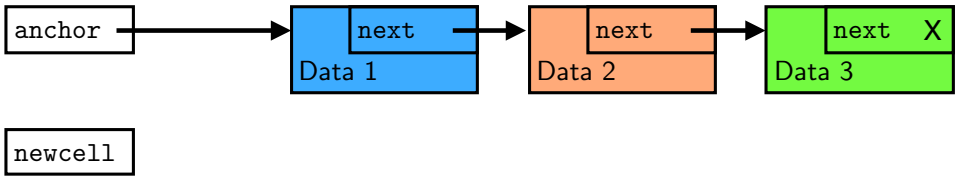
# Einfügen (hinten)



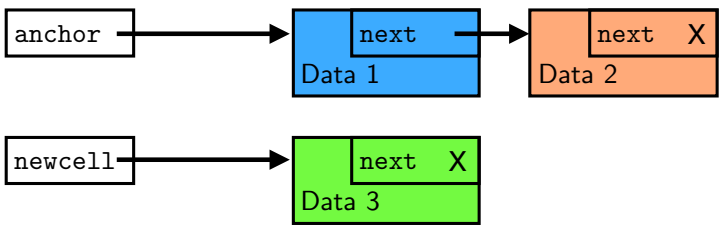

```
LinListInsertLast(&anchor, newcell);
```



# Ausfügen (hinten)




```
newcell = LinListExtractLast(&anchor);
```



# Weitere Operationen

- `LinList_p LinListFind(LinList_p anchor, char* payload);`
  - Finde eine Listenzelle anhand der `payload`
  - Gibt Pointer auf gefundenen Zelle oder NULL zurück
- `LinList_p LinListExtract(LinList_p *anchor, LinList_p cell);`
  - Entferne eine beliebige Zelle aus der Liste und gib sie zurück
- `LinList_p LinListRevert(LinList_p *anchor);`
  - Kehre die Liste um
- `LinList_p LinListSort(LinList_p *anchor);`
  - Sortiere die Liste (ASCIIbetisch, Hinweis: `strcmp()`)

# Aufgaben Woche 3

- Erweitern Sie den Datentyp “Lineare Liste” um einige (oder alle) der vorgestellten Funktionen.
- Verwenden Sie diese, um ein Programm zu schreiben, dass
  - Beliebig viele Strings (bis 255 Zeichen Länge) einliest
  - Alle Duplikate verwirft oder löscht (d.h. nur die erste Kopie behält)
  - Die Liste danach in Originalreihenfolge wieder ausgibt
- Stellen Sie sicher, dass der gesamte allokierte Speicher auch wieder frei gegeben wird.

# Beispielausgabe

```
> ./myuniq
ab
ac
ab
bc
bc
def
Output
=============
ab
ac
bc
def
```

# Feedback/Abschluss

- Erfolg?
- Schwierigkeiten?
- Sonstiges?

# Ziele für Woche 4

- Komplexere Listenfunktionen
  - Suchen/Finden
  - Sortieren

# Weitere Operationen

- `LinList_p LinListFind(LinList_p anchor, char* payload);`
  - Finde eine Listenzelle anhand der `payload`
  - Gibt Pointer auf gefundenen Zelle oder NULL zurück
- `LinList_p LinListSort(LinList_p *anchor);`
  - Sortiere die Liste (ASCIIbetisch, Hinweis: `strcmp()`)

# Aufgaben Woche 4

- Schreiben Sie ein Program `mysort`, welches beliebig viele Strings (einen pro Zeile) einliest, die Liste ASCIIbetisch sortiert, und dann in sortierter Reihenfolge wieder aus gibt
- Schreiben Sie ein Program `mystat`, welches beliebig viele Strings (einen pro Zeile) einliest, von jedem String ermittelt, wie oft dieser in der Eingabe vorkommt, und die Liste der verschiedenen Strings mit Zähler sortiert wieder ausgibt.

# Beispielausgabe

```
> ./mysort
ab
ac
ab
bc
bc
def
^D (Windows: ^Z<ret>)
Output
==============
ab
ab
ac
bc
bc
def
```

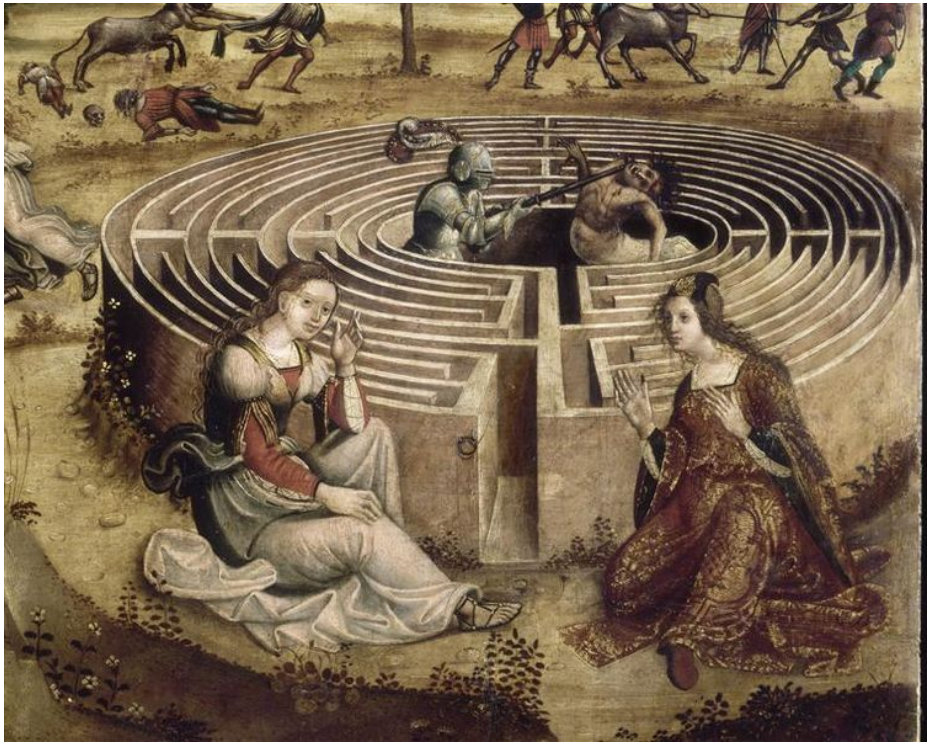
```
> ./mystat
ab
ac
ab
bc
bc
def
^D (Windows: ^Z<ret>)
Output
==============
2   ab
1   ac
2   bc
1   def
```

# Feedback/Abschluss

- Erfolg?
- Schwierigkeiten?
- Sonstiges?

# Ziele für Woche 5

- Aufgabe: Navigation im Labyrinth
  - Dynamische Programmierung
  - Etwas I/O
  - Fast keine Pointer!
  - Statt dessen 2-D Arrays



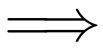
# Unsere Labyrinthe

```
########################
#S           #          #
# #####  #######  #   #
# #         #       #   #
# #  #######  #    #   #
# ## #        #    #   #
#       #  #  #    #   #
# ##########  #####   #
#      #      #     X#
########################
```

- 100% pure ASCII art!
  - `#`: Unpassierbare Wand
  - `S`: Startpunkt
  - `X`: Ziel
    - Schatz
    - Minotaurus
    - Ausgang
  - Alles andere (aber insbesondere `' '` und `'.'`) sind passierbar
  - Bewegung: Manhattan
    - Rechts, Links, Oben, Unten
    - Nicht diagonal (zumindest in Version 1.0)

# Aufgabe im Beispiel

```
##########################
#S           #           #
# #####  #######  #   #
# #          #       #   #
# #  #######  #    #   #
# ## #         #    #   #
#       #  #   #    #   #
# ##########  #####   #
#      #       #     X#
##########################
```



```
##########################
#S           #   .... #
#.#####  #######. #. #
#.#         # .... #. #
#.#  ####### .#   #. #
#.## #.......#   #. #
#..... #  #  #   #. #
# ##########  #####. #
#      #       #    .X#
##########################
Cost: 36
```

# Konkret: IO und die Kommandozeile

```
int main(int argc, char *argv[])
{
    FILE* in = stdin;
    Lab_p lab;
    if(argc > 2)
    {
        fprintf(stderr, "Usage: %s "
                "[<file >]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if(argc == 2)
    {
        in = fopen(argv[1], "r");
        if(!in)
        {
            perror(argv[0]);
            exit(EXIT_FAILURE);
        }
    }
    lab = LabRead(in);
```

- Definition von `main()`
  - Rückgabe `int` (z.B. `EXIT_SUCCESS` == 0 oder `EXIT_FAILURE`)
  - `int argc`: Anzahl der Argumente (einschließlich Programmname)
  - `char *argv[]`: Array der Argumente
    - `argv[0]`: Programmname
    - `argv[1]`: 1. Argument
  - Beispiel: `prog file.txt 7`
    - `argc = 3`
    - `argv[0] = "prog"`
    - `argv[1] = "file.txt"`
    - `argv[2] = "7"`

# Konkret: IO und die Kommandozeile

```
int main(int argc, char *argv[])
{
    FILE* in = stdin;
    Lab_p lab;
    if(argc > 2)
    {
        fprintf(stderr, "Usage: %s "
                "[<file>]\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    if(argc == 2)
    {
        in = fopen(argv[1], "r");
        if(!in)
        {
            perror(argv[0]);
            exit(EXIT_FAILURE);
        }
    }
    lab = LabRead(in);
```

- `FILE* in`: File pointer
- Default: `stdin`, *Standard Input*, per Default an das Terminal gebunden
- `fopen(name, "r")`: Öffne Datei zum Lesen (*reading*)
  - Erfolg: File pointer
  - Misserfolg: `NULL`
- `perror()`: Gibt Systemfehler aus, der zum aktuellen Wert von `errno` gehört
- `exit()`: Beendet das Programm. Rückgabewert signalisiert Status an die Umgebung!

# Aufgaben Woche 5

- Schreiben Sie ein Programm `labsolve` mit folgenden Funktionen:
  - Es liest ein ASCII-Labyrinth aus einer Datei ein
  - Es überprüft das Labyrinth
    - Genau ein Startpunkt `S`
    - Genau ein Ziel `X`
  - Es gibt das Labyrinth wieder aus
  - Es löst das Labyrinth
    - Es findet einen Weg von `S` nach `X`
    - Es markiert diesen mit `'.'`
    - Es gibt das Labyrinth mit dem markierten Weg wieder aus
    - Bonus: Es findet den/einen besten Weg von `S` nach `X`
- Einige Beispiellabyrinthe finden Sie unter `http://wwwlehre.dhbw-stuttgart.de/~sschulz/algo2016.html`
- Bonusaufgabe: Erlauben Sie auch diagonale Bewegung
- Bonus-Bonus: Berücksichtigen Sie verschiedene Terrains (Ziffern von 1-9 in der Karte sind sie Kosten, auf das entsprechende Feld zu treten, andere Felder haben Kosten 1)

# Torvalds: *Good programmers worry about data structures*

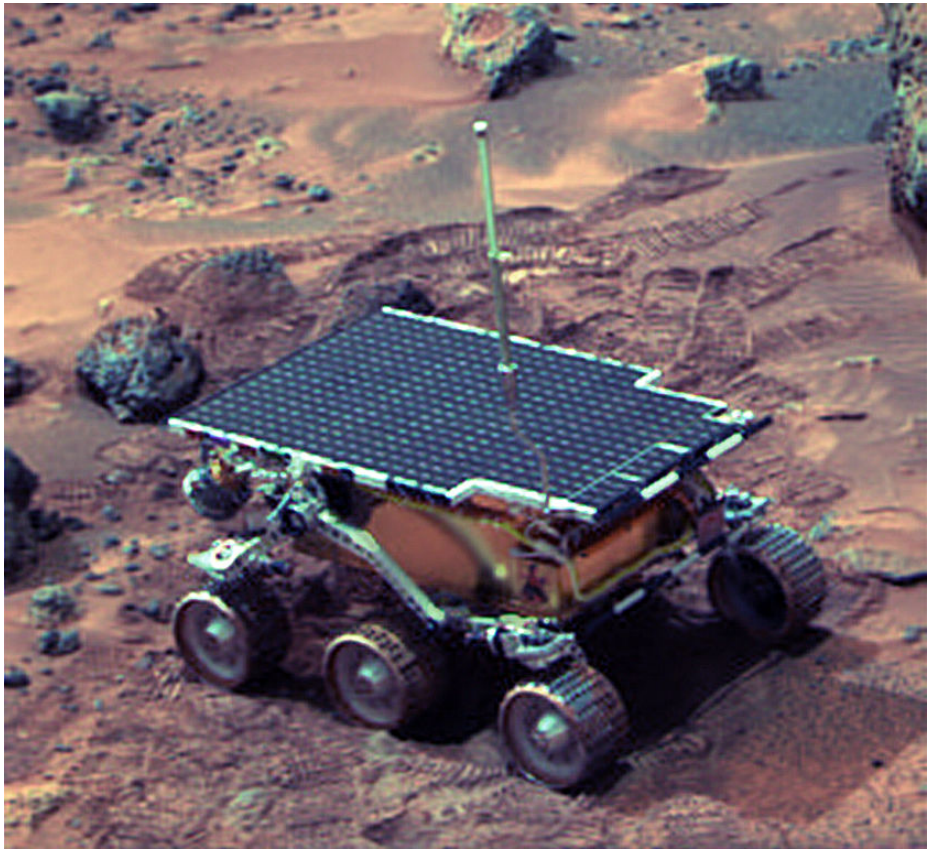
```
#define MAXCOLS   256
#define MAXROWS   256

/* Data type: Labyrinth - ASCII map, costs,
   directions */
typedef struct labyrinth
{
    char lab[MAXCOLS+2][MAXROWS];
    long costs[MAXCOLS][MAXROWS];
    long bestx[MAXCOLS][MAXROWS];
    long besty[MAXCOLS][MAXROWS];
    int maxrow;
    int startx;
    int starty;
    int treasurex;
    int treasurey;
}LabCell, *Lab_p;
```

# Feedback/Abschluss

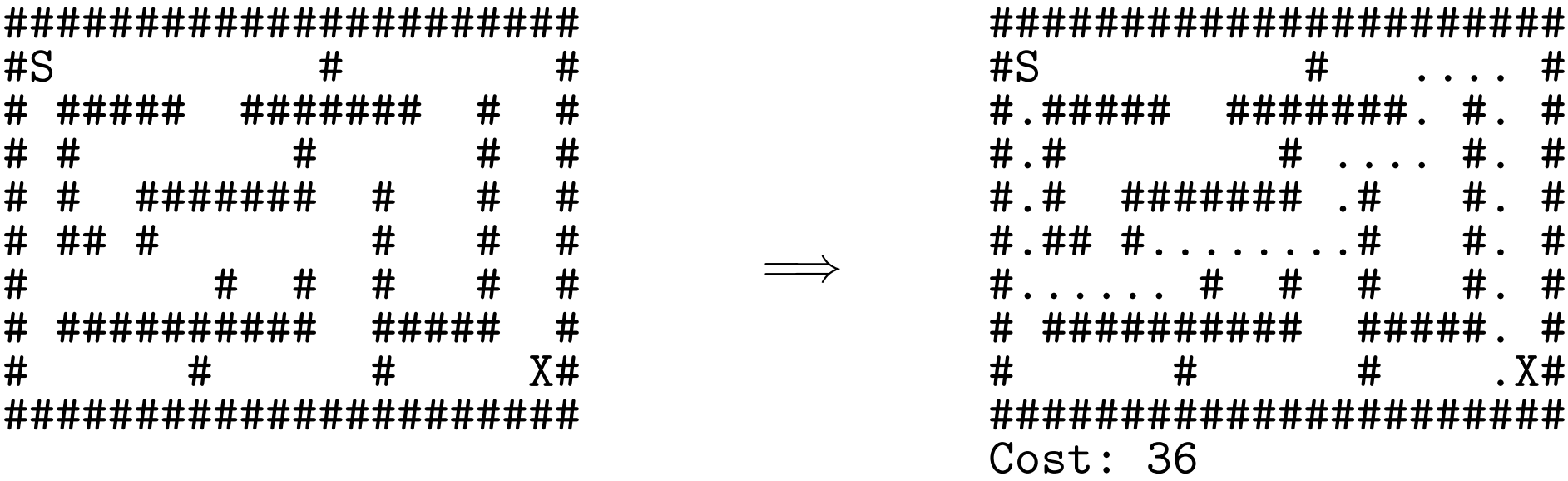
- Erfolg?
- Schwierigkeiten?
- Sonstiges?

# Ziele für Woche 6

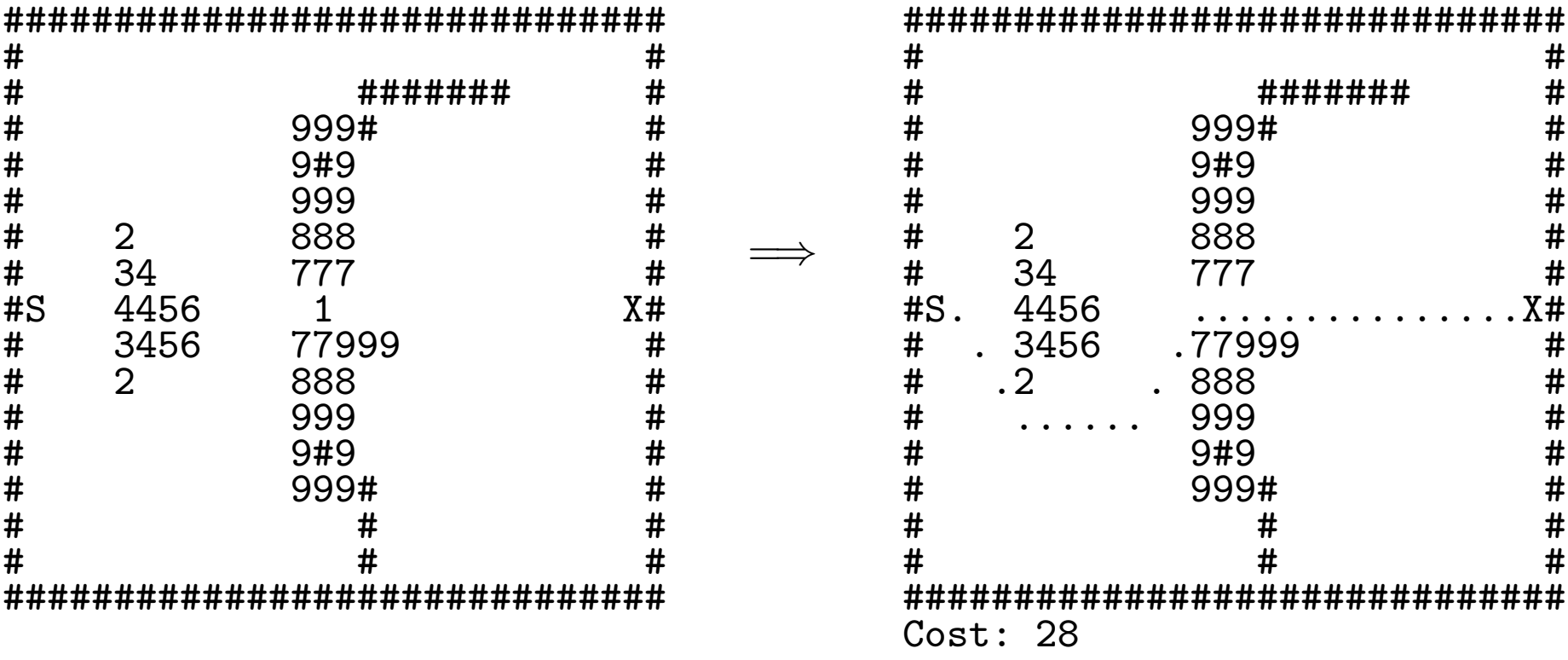
- Aufgabe: Navigation im Labyrinth
  - Optimale Wege
  - Diagonalen
  - Variable Kosten



# Basisnavigation



# Gelände


##################################
#                                #
#                  #######       #
#             999#               #
#             9#9                #
#             999                #
#    2        888                #
#    34       777                #
#S   4456      1                 X#
#    3456     77999              #
#    2        888                #
#             999                #
#             9#9                #
#             999#               #
#                #               #
#                #               #
##################################
##################################
#                                #
#                  #######       #
#             999#               #
#             9#9                #
#             999                #
#    2        888                #
#    34       777                #
#S.  4456     ..............X#
#  . 3456    .77999              #
#   .2      . 888                #
#    ......   999                #
#             9#9                #
#             999#               #
#                #               #
#                #               #
##################################
Cost: 28

# Lösungsansatz: Berechne *alle* kürzeste Wege

- Suche beginnt beim Startfeld
- Kostenarray speichert für jedes Feld:
  - War ich schon mal da?
  - Wenn ja, wie lang war der Weg bis dahin?
- Suchfunktion bekommt bisher aufgelaufene Weglänge mit
  - Beim Startfeld 0
  - Bei jedem neuen Feld die Kosten des Vorgängers + lokale Kosten (Normalfall: 1)
  - Wenn ein Feld betreten wird, Fallunterscheidung
    - Feld bekannt und gespeicherte Kosten günstiger oder gleich? Dann Ende!
    - Sonst: Speichere neue Kosten, und berechne rekursiv Kosten für alle Nachbarfelder
- Wenn die Funktion terminiert, haben wir zu jedem Feld die geringsten Kosten, es zu erreichen - auch zum Zielfeld!
  - Weg kann jetzt vom Zielfeld aus generiert werden (immer auf das Nachbarfeld mit den kleinsten Kosten)

# Aufgaben Woche 6

- Stellen Sie das Programm `labsolve` fertig:
  - Es liest ein ASCII-Labyrinth aus einer Datei ein
  - Es überprüft das Labyrinth
    - Genau ein Startpunkt `S`
    - Genau ein Ziel `X`
  - Es gibt das Labyrinth wieder aus
  - Es löst das Labyrinth
    - Es findet einen kürzesten Weg von `S` nach `X`
    - Es markiert diesen mit `'.'`
    - Es gibt das Labyrinth mit dem markierten Weg wieder aus
- Einige Beispiellabyrinthe finden Sie unter `http://wwwlehre.dhbw-stuttgart.de/~sschulz/algo2016.html`
- Bonusaufgabe: Erlauben Sie auch diagonale Bewegung
- Bonus-Bonus: Berücksichtigen Sie verschiedene Terrains (Ziffern von 1-9 in der Karte sind sie Kosten, auf das entsprechende Feld zu treten, andere Felder haben Kosten 1)

# Feedback/Abschluss

- Erfolg?
- Schwierigkeiten?
- Sonstiges?

# Ziele für Woche 7

- Sortier-Olympiade
  - Ein einfaches Verfahren
  - Ein Divide-and-Conquer-Verfahren
  - Vergleich der Performanz

# Timing unter UNIX (1)

```
#include <sys/time.h>

// Return the time in microseconds since the epoch.
long long GetUSecTime()
{
    struct timeval tv;

    gettimeofday(&tv, NULL);

    return (long long)tv.tv_sec*1000000ll+tv.tv_usec;
}
```

Windows: `ULONGLONG WINAPI GetTickCount64(void);`

# Timing unter UNIX (2)

```
long long time1, time2;
time1 = GetUSecTime();

// Do stuff that need timing here

time2 = GetUSecTime();
printf("Elapsed time: %lld usec\n", time2-time1);
```

Windows: Analog mit `GetTickCount64()`

# Aufgaben Woche 7

- Schreiben Sie ein Programm, dass folgende Aufgaben erfüllt:
  - Es ließt eine Liste von bis zu 100000 Ganzzahlen von einer Datei oder dem Terminal in ein Array ein
  - Es sortiert diese Liste mit Bubblesort, Selection Sort, oder Insert-Sort und bestimmt die Zeit, die der Sortiervorgang (ohne Eingabe und Ausgabe!) benötigt
  - Es gibt die sortiere Liste wieder aus
  - Es gibt die Zeit für das Sortieren aus
- Schreiben Sie ein analoges Programm mit Mergesort oder Quicksort
- Alternative: Schreiben Sie *ein* Programm, dass den Sortieralgorithmus per Kommandozeilenoption auswählt
- Bestimmen Sie jeweils die Laufzeit für Listen mit 10, 100, 1000, 10000 und 100000 Elementen. Was stellen Sie fest?

Testfiles:
`http://wwwlehre.dhbw-stuttgart.de/~sschulz/algo2016.html`

# Beispielausgabe

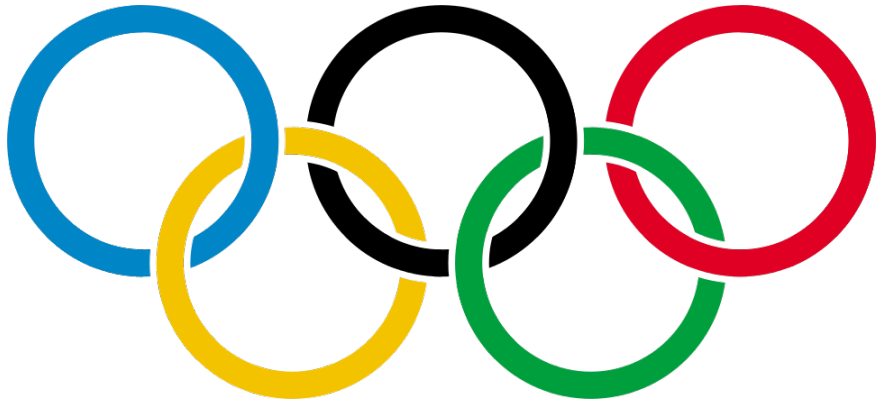
```
>  ./mysort listn_000010.txt
2262
21753
33749
41749
61914
72732
74884
80691
85378
95988
Sorting time: 2 usec
```

# Feedback/Abschluss

- Erfolg?
- Schwierigkeiten?
- Sonstiges?

# Ziele für Woche 8

- Sortier-Olympiade Teil 2
  - Heapsort
  - Optimierte Verfahren
  - Vergleich der Performanz



# Die Konkurrenz: `qsort()`

- `qsort()`: Teil der *C Standard Library*
  - Typischerweise optimiertes Quicksort
- Generisches Sortierverfahren
  - Sortiert Arrays beliebigen Typs
  - Die Ordnung wird vom Aufrufer bestimmt (per Funktionspointer)
- Vergleiche: `int cmpfun(const void* e1, const void* e2)`
  - Bekommt Pointer auf zwei Elemente des Arrays
  - Implementiert 3-Werte-Vergleich:
    - `*e1 < *e2`: Ergebnis ist kleiner als 0
    - `*e1 == *e2`: Ergebnis ist 0
    - `*e1 > *e2`: Ergebnis ist größer als 0
- Die Vergleichsfunktion muß wissen, welchen Typ die Argumente haben
  - Generische `void*` Pointer müssen gecastet werden
  - Heute etwas altmodisch, aber eines der Idiome, die C erfolgreich gemacht haben!

# qsort() Beispiel

```
#include <stdlib.h>
int cmpfun(const void* e1,
           const void* e2)
{
    const long *p1=e1, *p2=e2;
    if(*p1 > *p2)
    {   return 1;}
    if(*p1 < *p2)
    {   return -1;}
    return 0;
}
long numbers[MAXITEMS];
int main(int argc, char *argv[])
{
    long count = 0;
    ...
    qsort(numbers, count,
          sizeof(long), cmpfun);
     ...
}
```

- Argumente für `qsort()`
  - `numbers`: Das zu sortierende Array (in C also die Adresse, an der das erste zu sortierende Element steht)
  - `count`: Anzahl der Elemente
  - `sizeof(long)`: Größe eines einzelnen Elements
  - `cmpfun`: (Pointer auf) die Vergleichsfunktion

# qsort() Beispiel

```
#include <stdlib.h>
int cmpfun(const void* e1,
           const void* e2)
{
    const long *p1=e1, *p2=e2;
    if(*p1 > *p2)
    {   return 1;}
    if(*p1 < *p2)
    {   return -1;}
    return 0;
}
long numbers[MAXITEMS];
int main(int argc, char *argv[])
{
    long count = 0;
    ...
    qsort(numbers, count,
          sizeof(long), cmpfun);
     ...
}
```

- Argumente für `qsort()`
  - `numbers`: Das zu sortierende Array (in C also die Adresse, an der das erste zu sortierende Element steht)
  - `count`: Anzahl der Elemente
  - `sizeof(long)`: Größe eines einzelnen Elements
  - `cmpfun`: (Pointer auf) die Vergleichsfunktion

- Argumente für `qsort()`
  - Optimiert
  - Debugged

# Aufgaben Woche 8

- Erweitern Sie Ihr Programm von Woche 7:
  - Implementieren Sie Heapsort
  - Implementieren Sie ein optimiertes Sortierverfahren
    - Z.B. Bottom-Up Mergesort
    - Z.B. Quicksort mit gutem Pivot und Spezialbehandlung für kleine Listen

Herausforderung: Wie nahe kommen Sie an `qsort()`? Theoretisch sollten Sie es schlagen können!

Testfiles:
`http://wwwlehre.dhbw-stuttgart.de/~sschulz/algo2016.html`

# Feedback/Abschluss

- Erfolg?
- Schwierigkeiten?
- Sonstiges?

# Ziele für Woche 9

- Binäre Suchbäume
  - Aufbauen
  - Suchen
  - Ausgabe
  - (Löschen)

# Aufgaben Woche 9

- Implementieren Sie den Datentyp *Binärer Suchbaum* mit (mindestens) folgenden Operationen:
  - `insert(tree, key, value)`: Fügt das *key, value*-Paar in den Baum ein. Falls `key` schon existiert, wird der gespeicherte `value` mit dem neuen Wert überschrieben.
  - `find(tree, key)`: Sucht den Knoten mit dem entsprechenden Schlüssel und gibt ihn zurück. Falls kein Knoten existiert: `NULL`
  - `print_tree(tree)`: Gibt alle Key/Value-Paare geordnet (kleine Keys zuerst) aus
  - Bonus: `delete(tree, key`: Löscht Knoten mit Schlüssel `key` aus dem Baum
- Sie können folgende Annahmen machen:
  - Schlüssel sind Ganzzahlen (`int`)
  - Werte sind Strings (`char*`, maximal 32 Zeichen)
- Geschenkter Rahmen für Tests auf der Webseite!

# Ablaufbeispiel

Input:

```
I: 13,Dreizehn
I: 18,Achtzehn
I: 4,Vier
I: 6,Sechs
I: 12, Zwoelf
I: 14, Vierzehn
I: 2, Zwei
F: 7
=> Sollte fehlschlagen
F: 6
=> Sollte 6,Sechs ausgeben
F: 12
=> Sollte 12,Zwoelf ausgeben
P:
=> Sollte in etwa folgendes ausgeben:
(2, Zwei)
(4, Vier)
(6, Sechs)
(12, Zwoelf)
(13, Dreizehn)
(14, Vierzehn)
(18, Achtzehn)
D: 6
F: 6
=> Sollte jetzt fehlschlagen
I: 8,Hallo
P:
```

Output:

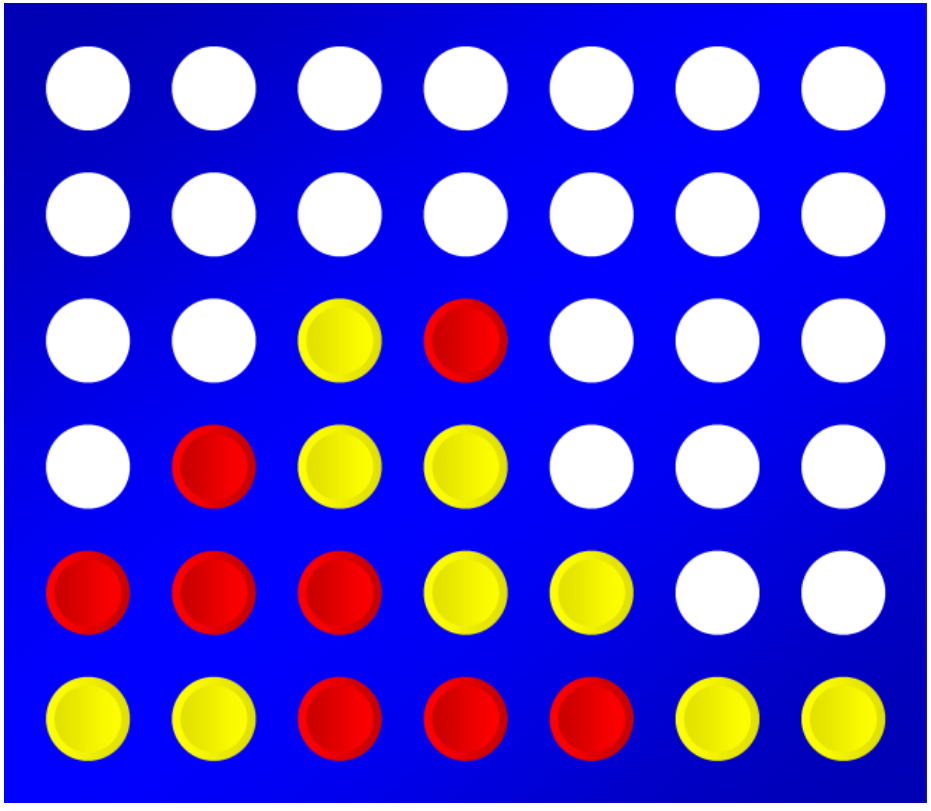
```
Inserting (13, Dreizehn) into the tree
Inserting (18, Achtzehn) into the tree
Inserting (4, Vier) into the tree
Inserting (6, Sechs) into the tree
Inserting (12,  Zwoelf) into the tree
Inserting (14,  Vierzehn) into the tree
Inserting (2,  Zwei) into the tree
Finding key 7 in the tree
Not found
Finding key 6 in the tree
Found (6, Sechs)
Finding key 12 in the tree
Found (12,  Zwoelf)
Current state:
(2,  Zwei)
(4, Vier)
(6, Sechs)
(12,  Zwoelf)
(13, Dreizehn)
(14,  Vierzehn)
(18, Achtzehn)
Removing record with key 6 from the tree
Finding key 6 in the tree
Not found
Inserting (8, Hallo) into the tree
 Current state:
(2,  Zwei)
(4, Vier)
(8, Hallo)
(12,  Zwoelf)
(13, Dreizehn)
(14,  Vierzehn)
```

# Feedback/Abschluss

- Erfolg?
- Schwierigkeiten?
- Sonstiges?

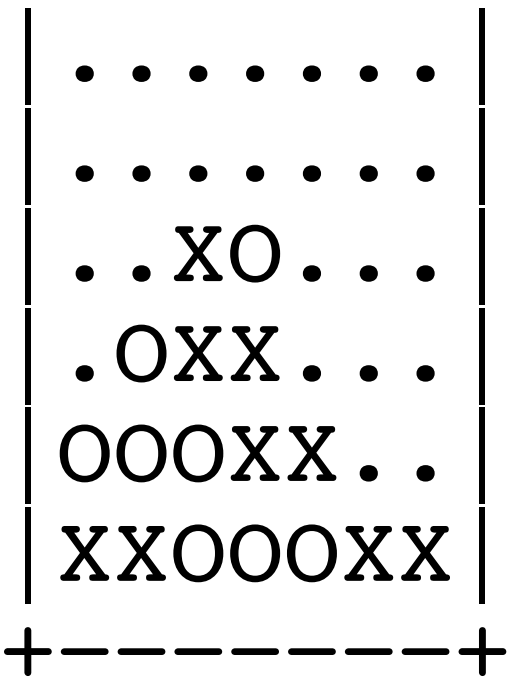
# Ziele für Woche 10&11

- Vier Gewinnt/Connect 4
  - Basisfunktionen
  - Anzeige
  - Einfacher Computerspieler
  - KI-Spieler



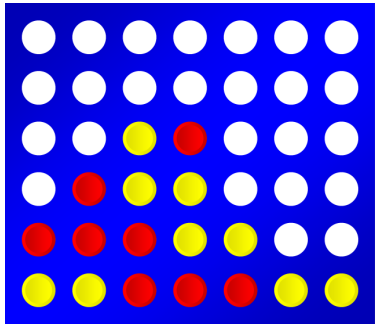
# Ziele für Woche 10&11

- Vier Gewinnt/Connect 4
  - Basisfunktionen
  - Anzeige
  - Einfacher Computerspieler
  - KI-Spieler



# Vier Gewinnt

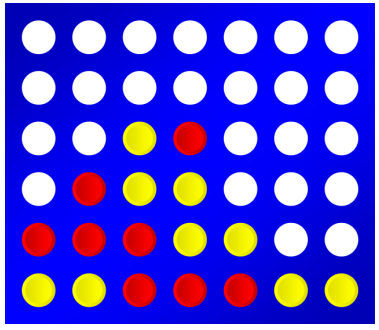
- Aufrecht stehendes Spielbrett, 7x6 Felder
- Zwei Spieler (Gelb/Rot bzw. `X`/`O`) ziehen abwechselnd
- Ziel: 4 eigene Steine in einer Reihe (horizontal/vertikal oder diagonal)
- Gespielte Steine fallen nach unten
- Unentschieden, wenn kein Zug mehr möglich ist



```
|.......|
|.......|
|..XO...|
|.OXX...|
|OOOXX..|
|XXOOOXX|
+-------+
```

# Vier Gewinnt

- Aufrecht stehendes Spielbrett, 7x6 Felder
- Zwei Spieler (Gelb/Rot bzw. `X`/`O`) ziehen abwechselnd
- Ziel: 4 eigene Steine in einer Reihe (horizontal/vertikal oder diagonal)
- Gespielte Steine fallen nach unten
- Unentschieden, wenn kein Zug mehr möglich ist



```
|.......|
|.......|
|..XO...|
|.OXX...|
|OOOXX..|
|XXOOOXX|
+-------+
```

Perfect Information Game - tyisches Beispiel für KI-Algorithmen (*Minimax*)

# Aufgabe Woche 10/11

1. Implementieren Sie das Spiel Vier Gewinnt
   - Sehen Sie zwei Spieler vor, von denen jeder Mensch oder Computer sein kann
   - Vorschlag: Stellen Sie das Board als 7x6 Array dar
2. Implementieren Sie einen einfachen Gegenspieler
   - Bewerten Sie ein Brett mit einer einfachen Heuristik
     - Spieler gewinnt: $+1$
     - Spieler verliert: -1
     - Ansonsten?
3. (Bonus) Implementieren Sie einen *guten* Gegenspieler
   - Informieren Sie sich über den Minimax-Algorithmus
   - Implementieren Sie Minimax
   - Doppelbonus: Implementieren Sie Alpha-Beta-Pruning

# Hilfreiche Hilfsfunktionen

- `BoardInit(Board_p board)`
- `BoardPrint(Board_p board)`
- `FindOpenRow(Board_p board, int column)`
  - Finde das unterste frei Feld in der angegeben Spalte
- `CheckVictory(Board_p board, char player)`
  - Teste, ob der Player mit Symbol `player` gewonnen hat
  - Ich habe dazu 4 Hilfsfunktionen gebraucht!
- `Move(Board_p board, int col, char player)`
  - Setze einen Stein für `player` in Spalte `col`
- `UnMove(Board_p board, int col`
  - Entferne den letzten Stein aus Spalte `col`
- `BoardBasicEval(Board_p board, char player, char opponent)`
  - Berechen Bewertung für den gegebene Spielstand aus der Sicht von `player`
- `FindMove(Board_p board, char player, char opponent, int level)`

# Feedback/Abschluss

- Erfolg?
- Schwierigkeiten?
- Sonstiges?