

Kapitel 11

Let's play

A strange game. The only winning move is not to play.

(WOPR im Film „Vargames“)

11.1 Einführung

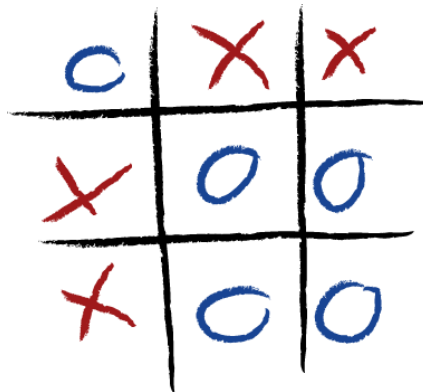
- Um die bisher gelernten Fähigkeiten im Zusammenhang anzuwenden, soll in diesem Kapitel ein größeres Projekt betrachtet werden
- Wir entwickeln ein **Spiel**

Achtung

In diesem Kapitel soll der Entwicklungsvorgang illustriert werden.
Deshalb werden auch vorläufige und falsche Lösungen präsentiert.

TicTacToe

- Spiel **Tic Tac Toe**
- **Regeln** (falls die jemand nicht kennt 😊):
 - Spielfeld ist ein Gitter mit 3×3 Feldern
 - Zwei Spieler setzen im Wechsel eigenes Symbol (meist Kreuz und Kreis) in ein Feld
 - Wer zuerst drei Symbole in einer Reihe oder Diagonalen hat, hat gewonnen
- Der Computer soll ein „Spieler“ sein



Grobstruktur

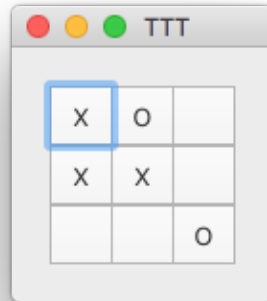
- TicTacToe ist ein rundenbasiertes Spiel
- Egal wer dran ist, es sollte immer der aktuelle Spielstand angezeigt werden
- Es wird bis zum Sieg oder Unentschieden gespielt
- Dies ergibt folgenden Grobalgorithmus:

Algorithm TTT-GENERAL

```
Input who will start → turn                                ▷ turn ∈ {player, computer}
repeat
  if turn = computer then
    Calculate move
    turn ← player
  else
    Input move
    turn ← computer
  end if
  Display move
until (somebody won) ∨ (draw)                                ▷ turn = player
```

Nutzerinterface

- Moderne Programme besitzen in der Regel eine graphische Nutzerschnittstelle (**GUI** - **graphical user interface**)



- Wir wollen aber ein Konsolenprogramm (Terminal) schreiben
- Wenn wir uns an den POSIX-Standard halten, läuft das Programm auf allen gängigen Betriebssystemen (sogar MS Windows 😊) ➔ **Portabilität**
- Auf dem Terminal könnte das Spiel etwa so aussehen:

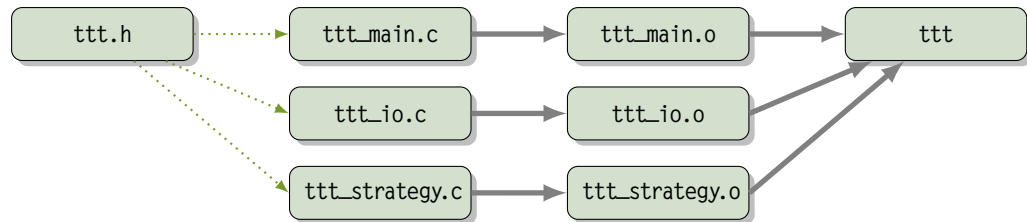
```
x|o|x
+-+
x|o|
+-+
o| |x
```

- Das ist nicht ganz so schön, aber erst einmal etwas einfacher und portabler
- Um eine spätere GUI nicht zu „verbauen“, sollte das Nutzerinterface so weit wie möglich von der anderen Programmlogik getrennt sein

Modularisierung

- Es liegt nahe, das Projekt in drei Module zu zerlegen:
 - „ttd_main.c“, das als Überbau fungiert und auch die main-Methode bereitstellt
 - „ttd_io.c“, das sich um die Ein- und Ausgabe kümmert
 - „ttd_strategie.c“, das für die Berechnung des Computerzuges verantwortlich ist

- Damit alle Module auf gemeinsame Typen und Interfaces zugreifen können, soll es eine gemeinsame Header-Datei „ttt.h“ geben



Exkurs: Make

- Wenn ein Projekt aus mehreren Modulen besteht, muss jedes einzeln zu einem Objektfile übersetzt (z.B. „gcc -c <file.c>“) und anschließend alle Objektdateien (mit den Bibliotheken) verlinkt werden
 - „Händische Ausführung“ ist umständlich
 - ➔ Batch-Job (Shellskript)
- Wird ein **einzelnes** Quellfile geändert, ist **nur** seine Übersetzung und das Linken zu wiederholen
 - ➔ Shellskript **ungeeignet**
- **Lösung: Buildsysteme**, die Abhängigkeiten berücksichtigen
- Betrachten das weitverbreitetste System: make
- make ist ein kleines Hilfsprogramm, das andere Programme **in Abhängigkeit von bestimmten Bedingungen** ausführt
- Dazu interpretiert make eine (einfache und namenlose) Programmiersprache, die in einem **Makefile** beschrieben wird
 - Die Make-Sprache folgt dem **Konditionalmodell** (siehe Kapitel 2)
 - Entsprechend besteht ein Makefile aus **Regeln**
 - Regeln beschreiben **Abhängigkeiten**
 - Außerdem gibt es **Variablen** und **Funktionen**

Variablen

<name>:= <wert> oder
 <name>= <wert>

- **Beispiel:**

```
CC:= cc
```

- Zugriff mit `$(CC)` oder `${CC}`
- Variante ohne Doppelpunkt („:=“) lässt Rekursion bei der Namensauflösung zu

Funktionen

```
$(<funktionsname> <arg 1>, <arg 2>, ...)
```

oder

```
${<funktionsname> <arg 1>, <arg 2>, ...}
```

• Beispiele:

```
SOURCEFILES= $(wildcard ttt_*.c)
OBJECTFILES= $(subst .c,.o,$(SOURCEFILES))
```

- Meist zur Textmanipulation
- Beispiele für Funktionen:
 - `$(subst <from>,<to>, <text>)`
Ersetzt in `<text>` alle Vorkommen von `<from>` durch `<to>`
 - `$(addprefix <prefix>,<list>)` / `$(addsuffix <suffix>,<list>)`
Fügt jedem Wort der Liste `<list>` den Prä- bzw. Suffix hinzu
 - `$(join <list1>,<list2>)`
Fügt die Listen wortweise zu einer gemeinsamen Liste zusammen
 - `$(foreach <var>,<list>,<text>)`
Kreiert für jeden Wert in `<list>` eine neue Instanz von `<text>`, in der jedes Vorkommen von `<var>` durch den Listenwert ersetzt wird
 - `$(shell <command>)`
Führt den Befehl `<command>` in einer Shell aus
- Mehr in der Dokumentation von „make“

Regeln

```
<zieldatei>: <quelldateien>↵
↳ <Befehle zur Generierung der Zieldatei>
```

• Beispiel:

```
euclid.o: euclid.c
$(CC) -c euclid.c -o euclid
```

- Regeln können auch für Klassen von Dateien geschrieben werden → „implizite Regeln“ oder „Musterregeln“
 - Prozentzeichen („%“) dient dabei als „Joker“
 - Spezielle (**automatische**) Variablen wie „\$*“ oder „\$@“ beziehen sich auf (Teile von) Zielen oder Vorbedingungen
- **Beispiel:**

```
%.o: %.c
$(CC) -c $< -o $@
```

- Einige automatische Variablen für Regeln:
 - \$@: Ziel einer Regel
 - \$<: Name der ersten Vorbedingung
 - \$?: Liste aller Vorbedingungen, die sich geändert haben
 - \$^: Liste aller Vorbedingungen
 - \${@D} bzw. \${<D}: Verzeichnis des Ziels bzw. der ersten Vorbedingung
 - {@F} bzw. {<F}: Dateiname (ohne Verzeichnis) des Ziels bzw. der ersten Vorbedingung
- Bei Aufruf von `make`

```
> make <ziel>
```

startet `make` alle Aktivitäten, die entsprechend der Datei „Makefile“ zur Aktualisierung von `<ziel>` notwendig sind

- Ob eine Datei aktuell ist, wird anhand des **Zeitstempels** der Datei entschieden
- Wenn `<ziel>` weggelassen wird, wird das Ziel der **ersten** Regel in „Makefile“ benutzt
- Regeln werden entsprechend ihrer Abhängigkeiten abgearbeitet

```

# Example Makefile
a: b
c: e f
  generates c from e and f
b: c d
  generates b from c and d

```

- In diesem Fall bei make a:
 - „d“, „e“ und „f“ müssen vorhanden sein
 - Zuerst wird „c“ aus „e“ und „f“ generiert/aktualisiert
 - Dann wird „b“ aus „c“ und „d“ generiert/aktualisiert
 - Da es für „a“ Ausführungsteil gibt, ist hier die Abarbeitung beendet

PAGE 3

DEPARTMENT	COURSE	DESCRIPTION	PREREQS
COMPUTER SCIENCE	CPSC 432	INTERMEDIATE COMPILER DESIGN, WITH A FOCUS ON DEPENDENCY RESOLUTION.	CPSC 432

Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/754/>

- Beispielsweise könnte die Datei „Makefile“ für unser TicTacToe-Projekt so aussehen:

```

PROG=ttt
CC=cc
CFLAGS= -std=c99 -pedantic -Wall -Wextra
LDFLAGS=

HEADER= $(wildcard ttt*.h)
SOURCEFILES= $(wildcard ttt_*.c)
OBJECTFILES= $(subst .c,.o,$(SOURCEFILES))

$(PROG): $(OBJECTFILES)
        $(CC) $(LDFLAGS) $(OBJECTFILES) -o $(PROG)

%.o: %.c
        $(CC) -c $(CFLAGS) $<

```

```

○
○
○ .PHONY: clean
○ clean:
○
○          rm -f $(PROG) *.o
○

```

11.2 Die Hauptschleife und Ein-/Ausgabe

Hauptschleife

- Betrachten Hauptschleife
- Bevor wir uns an den ersten C-Code machen, überlegen wir, ob noch mehr Eigenschaften gefragt sind
 - Erkennung, ob Gewinn oder Unentschieden \Rightarrow vermutlich eng mit Strategie verwandt, sollte dort bereit gestellt werden
 - Abbruch auch während des Spiels \Rightarrow muss in Hauptschleife ausgewertet werden
 - Mehrere Spiele in Serie \Rightarrow Hauptschleife
- Verfeinern Grobstruktur

Hauptschleife

Algorithm TTT-GENERAL (2. VERSION)

```

repeat
  Input who will start  $\rightarrow$  turn  $\triangleright$  turn  $\in$  {player, computer}
  repeat
    if turn = computer then
      Calculate move
      turn  $\leftarrow$  player
    else  $\triangleright$  turn = player
      Input move  $\triangleright$  Abort is special move
      turn  $\leftarrow$  computer
    end if
    Display move
  until (somebody won)  $\vee$  (draw)  $\vee$  (abort)
  Input if another game?
until not another game

```


Hauptmodul

- Die Hauptschleife in C könnte dann etwa so aussehen

```

8  do {      /* main loop */
9      int player = ttt_x_or_o();           // select symbol (x or o)
10     bool computer_turn= (player == 'o');
11     ttt_init_board(board);              // erase playground
12     int move, assessment;
13     do { /* main loop for a game instance */
14         ttt_update_display(board);        // display board
15         move = computer_turn?            // computer's move?
16             ttt_calculate_move(board, ttt_opponent(player)); // calculate
17             ttt_input_move(board);       // player's move
18         if (move != TTT_ABORT)           // continue?
19             board[move] = computer_turn? // computer's move?
20             ttt_opponent(player) : player; // apply move
21         computer_turn=!computer_turn;    // change the turn
22         assessment = ttt_won_or_draw(board,player); // game over?
23         if (assessment != TTT_UNDECIDED) {
24             ttt_update_display(board);    // display final board
25             ttt_output_result(assessment); // print result
26         }
27     } while ((move != TTT_ABORT) && (assessment == TTT_UNDECIDED));
28 } while(ttt_another_game() == true);

```

- Man beachte, dass wir Funktionen nutzen, die bisher unbekannt sind
- Diese sollen z.T. von anderen Modulen zur Verfügung gestellt werden

Best Practice

Namen, die global sichtbar sind (z.B. nicht-statische Funktionen) sollten einen Präfix bekommen, der das Projekt und möglicherweise auch das Modul beschreibt.

Auf diese Weise wird ein Namenskonflikt unwahrscheinlicher.

- In unserem Fall: „ttt_“ als Präfix

Datenstruktur

- Wie sollen die Daten in C dargestellt werden?
- Naheliegend für Spielsituation: multidimensionales Array (`int field[3][3];`)

- **Aber:**

- Stets zwei Indizes notwendig

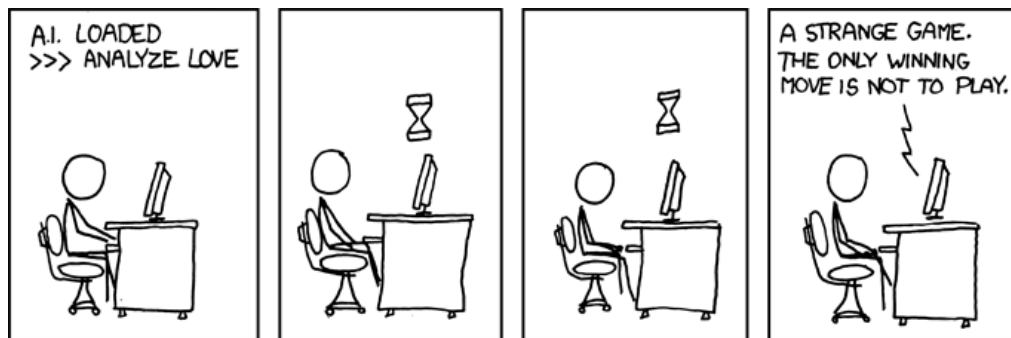
- ➔ Suche braucht dann stets zwei Schleifen

- **Außerdem:** Situation soll vermutlich an Funktion übergeben werden ➔ Array-Zerfall bei geschichtetem Array noch unübersichtlicher

- ➔ Nutzen „normales“ Array (int board[9], bzw. typedef int ttt_boear_t[NUMBER_OF_FIELDS];)

- Feldzuordnung:

0	1	2
3	4	5
6	7	8



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/601/>

Interface

- Deklarationen von Typen und Konstanten kommen in das gemeinsame Header-File

```
1 #include <stdbool.h>
2
3
4 /* general constants */
5 typedef enum {TTT_ABORT=-1, NUMBER_OF_FIELDS=9} ttt_constant_t;
6 /* game outcomes */
7 typedef enum {TTT_PLAYER_WINS, TTT_COMPUTER_WINS, TTT_DRAW,
8               TTT_UNDECIDED} ttt_result_t;
9 typedef int ttt_board_t[NUMBER_OF_FIELDS];
```

- ... genauso wie die Prototypen der (noch zu schreibenden) C-Funktion, die aus anderen Modulen aufgerufen werden

Hauptmodul

- Die beiden verbleibenden Funktionen im Hauptmodul sind relativ simpel:

```

33 void ttt_init_board(ttt_board_t f)
34 {
35     /* empties every field */
36     int i;
37     for(i=0; i<NUMBER_OF_FIELDS; i++)
38         f[i]=' ';
39 }
40
41 char ttt_opponent(char symbol)
42 {
43     switch(symbol)
44     {
45         case 'x': return 'o';
46         case 'o': return 'x';
47         default: return ' ';
48     }
49 }

```

Ein- und Ausgabe

- Es ist häufig so, dass das Nutzerinterface am aufwändigsten ist
- In diesem Fall nicht ⇒ Textinterface
- **Problem:** Trotzdem etwas Bequemlichkeit erzielen
- **Beispiel:** Bildschirm löschen
 - Rausscrollen ⇒ Fenstergröße muss bekannt sein
 - Unix-Befehl „clear“ ⇒ **sehr** langsam und nicht portabel
 - ANSI/VT100-Steuerzeichen ⇒ auch nicht 100% portabel
 - Beste Lösung: Nutzung von portablen Bibliotheken
 - * Hier z.B.: ncurses (für Windows auch: pdcurses)
 - Wir nutzen für TTT der Einfachheit halber aber die Lösung mit den Steuerzeichen 😊

```

1 /* Use VT100 ESC code to clean terminal */
2 static void ttt_clean_terminal(void)

```

```

3 {
4     printf( "%c[2J", 27 );
5 }

```

- Die Bibliotheksfunktion „getchar()“ liest einen Tastendruck ein und gibt den Integer(!)wert zurück
- **Problem:**
 - Das Terminal gibt eine Eingabe erst weiter, wenn <RETURN> gedrückt wurde
 - Das Zeichen „↵“ ist Teil des Eingabestroms
- **Lösung:**
 - Die Funktion wird zweimal aufgerufen und das zweite Ergebnis verworfen
- Beispiel:

```

74 bool ttt_another_game()
75 {
76     int input;
77     printf("Do you want to play another game [y/n] ->");
78     input=getchar(); getchar();
79     if ((input=='Y') || (input=='y'))
80         return true;
81     else
82         return false;
83 }

```

- Eine andere Alternative wäre die Änderung des Terminal-Modus (nicht portabel) oder wieder die Nutzung einer Bibliothek (z.B. wieder ncurses)
- Bei der Zugeingabe überprüfen wir gleich die Korrektheit der Eingabe:

```

44 int ttt_input_move(const ttt_board_t board)
45 {
46     int input;
47     printf("\nPlease enter the number of the field you want to occupy
48         (0 for abort)\n");
49     ttt_display_board(numberfield);
50     printf("Your move ->");
51     do{
52         do {
53             input=getchar(); getchar();
54             } while ((input<'0') || (input>'9'));
55         if (input=='0') return TTT_ABORT;
56     } while (1);
57 }

```

```

55     input=input-'1'; // character => integer index
56   } while (board[input]!=' ');
57   return input;
58 }

```

- Für die Ausgabe wird ein wenig „ASCII-Art“ gezeichnet:

```

void ttt_display_board(const ttt_board_t board)
{
  for(int i=0;i<3;i++){
    if(i) // not for first row
      printf("\n  -+-");
    printf("\n  ");
    for(int j=0;j<3;j++){
      if(j) printf("|"); // not for first column
      printf("%c",board[3*i+j]);
    }
  }
  printf("\n");
}

```

- Das sieht dann z.B. so aus:

```

o|x|x
+-+
x|x|o
+-+
o|x|

```

11.3 Strategie

- Zunächst entwerfen wir eine Grobstrategie
- Diese kann später verfeinert werden

Algorithm TTTCHOOSEMOVE (1. VERSION)

```

if Can I win? then
  Choose winning move
else
  if Can opponent win? then
    Block winning move
  else

```

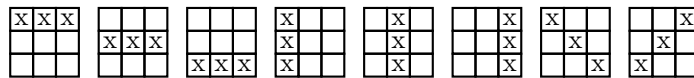
```

    if Can I win next time? then
        Prepare win
    else
        Whatever
    end if
end if
end if

```

Gewinnsituationen

- **Ziel:** Funktion, die Siegeszug bestimmt
- Es gibt nur acht verschiedene Möglichkeiten zu gewinnen:



- Eine solche Gewinnmöglichkeit soll „**Triple**“ genannt werden
- Ein Gewinn mit einem Triple ist möglich, wenn zwei der Triple-Felder mit dem eigenen Symbol besetzt sind und das dritte Feld frei ist
- Dies ergibt $8 \cdot 3 = 24$ Kombinationen für „Can I win?“

```

1  /* NOT in final version */
2  int winning_move(const ttt_board_t f, char s)
3  {
4      /* triplet (0,1,2) */
5      if((f[0]==s)&&(f[1]==s)&&(f[2]==' ')) return 2;
6      if((f[1]==s)&&(f[2]==s)&&(f[0]==' ')) return 0;
7      if((f[0]==s)&&(f[2]==s)&&(f[1]==' ')) return 1;
8
9      /* triplet (3,4,5) */
10     if((f[3]==s)&&(f[4]==s)&&(f[5]==' ')) return 5;
11     if((f[4]==s)&&(f[5]==s)&&(f[3]==' ')) return 3;
12     if((f[3]==s)&&(f[5]==s)&&(f[4]==' ')) return 4;
13     :
44     /* triplet (2,4,6) */
45     if((f[2]==s)&&(f[4]==s)&&(f[6]==' ')) return 6;
46     if((f[4]==s)&&(f[6]==s)&&(f[2]==' ')) return 2;
47     if((f[2]==s)&&(f[6]==s)&&(f[4]==' ')) return 4;

```

```

48 return -1;
49 }

```

- Eine solche Funktion (obgleich korrekt) ist unnötig groß und somit fehleranfällig
- **Idee:** Nutzung von Daten und Indirektionen

```

1 /* NOT in final version */
2 enum {NUMBER_TRIPLES=8, NONE=-1};
3 typedef int ttt_triple_t[3];
4 const ttt_triple_t triples[NUMBER_TRIPLES]={
5     {0,1,2},{3,4,5},{6,7,8},{0,3,6},{1,4,7},{2,5,8},{0,4,8},{2,4,6}
6 };
7
8 int winning_move(const ttt_board_t f,char s) {
9     for(int i=0; i<NUMBER_TRIPLES;++i)
10        for(int j=0; j<3; ++j) {
11            int idx2=(j+1)%3; /* 3 => 0 */
12            int idx3=(j+2)%3; /* 3 => 0; 4 => 1 */
13            if ((f[triples[i][j]]==' ') &&
14                (f[triples[i][idx2]]==s) && (f[triples[i][idx3]]==s))
15                return triples[i];
16        }
17     return NONE;
18 }

```

- Für den Schritt „Can opponent win?“ könnte die Funktion noch einmal genutzt werden:

```

1 /* NOT in final version */
2 int block_opponent_winning_move(const ttt_board_t f,char s)
3 {
4     return winning_move(f,ttt_opponent(s));
5 }

```

- Damit zeichnet sich eine Struktur ab: Es werden mehrere Funktionen benötigt, die die Stellung bewerten, ähnlich zu winning_move()
- **Idee:** Allgemeinere Bewertungsfunktion

Allgemeine Bewertung

- Was könnte von Interesse sein?

– Triple gewonnen \Rightarrow Feststellung des Spielendes:

X	.	.
.	X	.
.	.	X

– Triple kann zum Sieg führen \Rightarrow `winning_move()`

.	.	.
.	X	.
.	.	X

– Triple „bisher meins“ \Rightarrow könnte Gegner zwingen

X	.	.
.	.	.
.	.	.

– Triple leer

.	.	.
.	.	.
.	.	.

– Triple ist nutzlos

.	.	.
.	O	.
.	.	X

oder

O	.	.
.	X	.
.	.	X

oder

O	.	.
.	X	.
.	.	O

- Die ersten drei Bewertungen entsprechend für gegnerisches Symbol
- Wie können verschiedene Bewertungen nach einem gemeinsamen Schema durchgeführt werden?
- **Idee:** Nutzung von Primzahlen ($\neq 3$)
 - Eigenes Symbol zählt 2
 - Gegnersymbol zählt 5
 - Leeres Feld zählt 0
- Felder eines Triples werden addiert \Rightarrow
 - Eigener Sieg $\Rightarrow \Sigma = 6$
 - Sieg Gegner $\Rightarrow \Sigma = 15$
 - Eigener Sieg möglich $\Rightarrow \Sigma = 4$
 - Gegnerischer Sieg möglich $\Rightarrow \Sigma = 10$
 - Mein Triple $\Rightarrow \Sigma = 2$
 - Gegner-Triple $\Rightarrow \Sigma = 5$
 - Leeres Triple $\Rightarrow \Sigma = 0$
 - Nutzloses Triple $\Rightarrow \Sigma = 7 \vee \Sigma = 9 \vee \Sigma = 12$
- Bei einem solchen Ansatz muss jedes Triple nur einmal evaluiert werden
- **Nachteil:** Es muss anschließend **im** Triple nach dem leeren Feld gesucht werden \Rightarrow nehmen wir in Kauf, da nur einmal ausgeführt

```

1 enum {NUMBER_TRIPLES=8, I_KEY=2, OPP_KEY=5}; // more, if needed...
2

```



```

3 typedef int ttt_triple_t[3];
4 const ttt_triple_t triples[NUMBER_TRIPLES]={
5   {0,1,2},{3,4,5},{6,7,8},{0,3,6},{1,4,7},{2,5,8},{0,4,8},{2,4,6}
6 };
7
8 int ttt_evaluate(const ttt_board_t field, int tnr, int my_symbol) {
9   int val=0;
10  int opp_symbol=ttt_opponent(my_symbol);
11  for (int i=0; i<3; ++i) {
12    if (field[triples[tnr][i]]==my_symbol)
13      val+=I_KEY;
14    else if(field[triples[tnr][i]]==opp_symbol)
15      val+=OPP_KEY;
16  }
17  return val;
18 }

```

- Die Computerstrategie sähe unter Nutzung von ttt_evaluate() bis jetzt so aus:

```

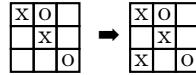
1 static int winning_move(const ttt_board_t board,
2   const ttt_tripleval_t eval, int who) {
3   for (int i=0; i<NUMBER_TRIPLES; i++)
4     if (eval[i]==who) // find winning triple
5       for(int j=0;j<3;j++) // find the empty place
6         if (board[triples[i][j]]==' ') return triples[i][j];
7   return NONE;
8 }
9
10 int ttt_calculate_move(const ttt_board_t board, char symbol) {
11   ttt_tripleval_t eval;
12   for (int i=0; i< NUMBER_TRIPLES; ++i)
13     eval[i]=ttt_evaluate(field,i,symbol);
14   int move=winning_move(board,eval,I_CANWIN);
15   if (move!=NONE) return move;
16   move=winning_move(board,eval,OPP_CANWIN);
17   if (move!=NONE) return move;
18   /* yet to implement */
19   return NONE;
20 }

```

Gewinn vorbereiten

- Betrachten wir noch einmal die Grobstrategie

- Wie kann ein Computer-Gewinn im **nächsten** Zug vorbereitet werden?
- **Idee:** Zwei gewinnbringende Triples vorbereiten ➔ **Gabelung**
- **Beispiel** (Computer hat x):



- In diesem Fall kann der o-Spieler nur **ein** Triple blockieren
- Das gemeinsame Feld soll **Achsfeld** genannt werden
- Um ein Achsfeld zu finden, müssen folgende Bedingungen erfüllt sein
 - Es muss zwei Triples geben, die bisher ausschließlich dem Computer gehören ➔ Bewertung: 2
 - Diese müssen ein **gemeinsames leeres** Element besitzen
- **Vorgehen:**
 - Wir betrachten alle acht Gewinntripel und untersuchen, ob es mit 2 bewertet wurde
 - Jedes Spielfeld, das in jeweils einem dieser Tripel **leer** vorkommt, erhält einen Punkt
 - Ein Spielfeld, das so mehr als einen Punkt erhalten hat, ist ein Achselement
- Im Array singles[NUMBER_OF_FIELDS] werden die „Punkte“ gezählt

```

1 int forking_move(const ttt_playground_t field, const ttt_triplev_t
  eval)
2 {
3   int i,j=0;
4   int singles[NUMBER_OF_FIELDS]={0,0,0,0,0,0,0,0,0,0};
5   for (i=0; i<NUMBER_TRIPLES; ++i)
6     {
7       if (eval[i]==I_KEY)
8         {
9           for(j=0;j<3;j++)
10            if (field[triples[i][j]]==' ')
11              ++singles[triples[i][j]];
12        }
13    }
14   for(i=0; i< NUMBER_OF_FIELDS; ++i)
15     if (singles[i]>=2) return i;
16   return NONE;

```

17 }

Offensive

- Angenommen, unsere Strategie hat bisher keinen Zug ergeben
- ~~Idee~~: Wie schon beim Sieges-Zug: Strategie spiegeln und verhindern
- **Problem**: Gegner könnte zwei Achselemente besitzen

Algorithm TTTCHOOSEMOVE (2ND VERSION)

```

if Can I win? then
  Choose winning move
else
  if Can opponent win? then
    Block winning move
  else
    if Can I fork? then
      Choose pivot element
    else
      if Can opponent fork? then
        Block opponent's pivot element
      else
        Whatever
      end if
    end if
  end if
end if

```

- **Alternative**: In die Offensive gehen
 - ➔ Zwingen, kommenden Sieg zu blockieren
- **Problem**: Gegner darf nicht in eine Gabel gezwungen werden
- **Beispiel** (aus Sicht von „x“):

–

O	X	
	O	
		X

– Es darf nicht auf Feld 8 gesetzt werden, weil dies zu gegnerischer Gabel führt

➔

O	X	
	O	
O	X	X

- Die Strategie sieht jetzt so aus:

Algorithm TTTCHOOSEMOVE (3RD VERSION)

```

if Can I win? then
    Choose winning move
else
    if Can opponent win? then
        Block winning move
    else
        if Can I fork? then
            Choose pivot element
        else
            if Can I force without opponent's fork? then
                Choose forcing move
            else
                Move to best available place
            end if
        end if
    end if
end if

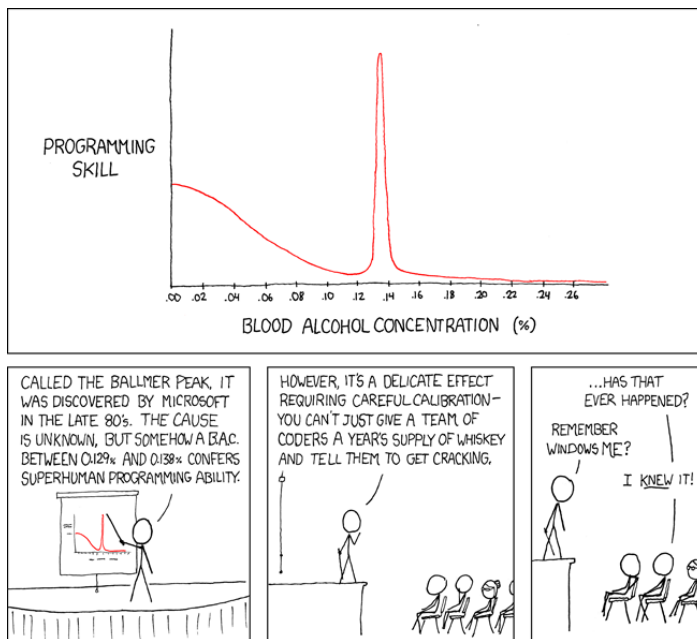
```

- Um den Offensiv-Zug zu finden, untersuchen wir zunächst, wohin wir **nicht** ziehen dürfen ⇒ Verbleibendes Feld ist Achselemente des Gegners
- Dann suchen wir Triples, in denen **genau ein** eigenes (und **kein** gegnerisches) Zeichen ist ⇒ Bewertung: 2
- Wenn dieses Triple **kein** gegnerisches Achselement enthält, wähle ein leeres Feld als Offensivzug
- Sonst wähle ein Achselement des Gegners

```

1 int forcing_move(const ttt_board_t board, const ttt_tripleva_t
   eval) {
2     int opp_singles[NUMBER_OF_FIELDS];
3     for (int i=0; i<NUMBER_TRIPLES; ++i) {
4         if (eval[i]==OPP_KEY) {
5             for(int j=0;j<3;j++)
6                 if (board[triples[i][j]]==' ')
7                     ++opp_singles[triples[i][j]];
8         }
9     }
10    for (int i=0; i<NUMBER_TRIPLES; ++i) {

```



Quelle: xkcd - A webcomic of romance, sarcasm, math, and language
<http://xkcd.com/323/>

```

11  if (eval[i]==I_KEY) {
12      if((opp_singles[triples[i][0]]<=1) &&
13          (opp_singles[triples[i][1]]<=1) &&
14          (opp_singles[triples[i][2]]<=1))
15          for(j=0;j<3;j++)
16              if (board[triples[i][j]]==' ')
17                  return triples[i][j];
18          for(j=0;j<3;j++)
19              if(opp_singles[triples[i][j]]>1)
20                  return triples[i][j];
21      }
22  }
23  return NONE;
24  }

```

Was bleibt

- Wenn alles andere fehlschlägt, wählen wir irgendein Feld
- Unterschiedliche Wertigkeiten: Mitte ➔ Ecke ➔ Rand

```

1 int best_remaining_move(const ttt_board_t board)
2 {
3     const int best[]={4,0,2,6,8,1,3,5,7};
4
5     for(int i=0; i<NUMBER_OF_FIELDS; ++i)
6         if(board[best[i]]==' ') return best[i];
7     return NONE; /* Should never happen */
8 }

```

Gesamtstrategie

- Mit Hilfe der beschriebenen Funktionen der Zug relativ berechnet werden

```

120 int ttt_calculate_move(const ttt_board_t field, char symbol)
121 {
122     ttt_tripleval_t eval;
123     for (int i=0; i< NUMBER_TRIPLES; ++i)
124         eval[i]=evaluate(field,i,symbol);
125     int move=winning_move(field,eval,I_CANWIN);
126     if (move!=NONE) return move;
127     move=winning_move(field,eval,OPP_CANWIN);
128     if (move!=NONE) return move;
129     move=forking_move(field,eval);
130     if (move!=NONE) return move;
131     move=forcing_move(field,eval);
132     if (move!=NONE) return move;
133     return best_remaining_move(field);
134 }

```

- Nach „außen“ muss nur ttt_calculate_move (und die noch nicht betrachtete Funktion ttt_won_or_draw) sichtbar sein
- Deshalb deklarieren wir alle „internen“ Funktionen als static

Spielende

- Die Entscheidung über das Spielende funktioniert ähnlich zur Zugentscheidung

```

136 ttt_result_t ttt_won_or_draw(const ttt_board_t board, char symbol)
137 {
138     bool undecided=false;

```

```

139 for(int i=0; i<NUMBER_TRIPLES; i++) {
140     int state=evaluate(board,i,'x');
141     switch (state) {
142     case I_WIN:
143         if (symbol=='x') return TTT_PLAYER_WINS;
144         else return TTT_COMPUTER_WINS;
145     case OPP_WIN:
146         if (symbol=='o') return TTT_PLAYER_WINS;
147         else return TTT_COMPUTER_WINS;
148     case DRAW_I: /* trough */
149     case DRAW_OPP: /* trough */
150         break;
151     case DRAW: /* trough */
152     default:
153         undecided=true;
154     }
155 }
156 if (undecided) return TTT_UNDECIDED;
157 else return TTT_DRAW;
158 }

```

Fertig!

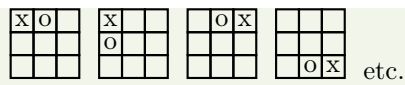
- Damit ist das Tic-Tac-Toe-Programm im Wesentlichen fertiggestellt
- Im Anhang C.2 des Skript ist der Code des kompletten Programms dokumentiert

Alternativen

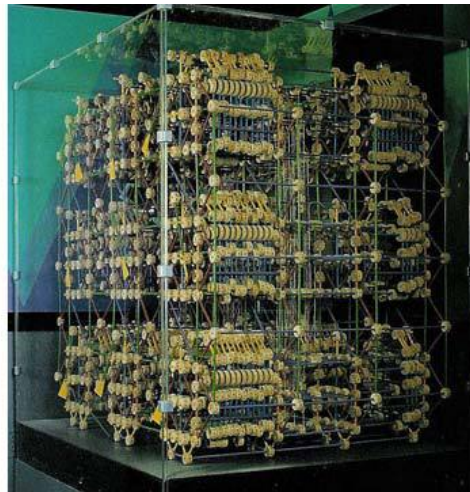
- Die hier dargestellte Strategie stellt **eine** Möglichkeit dar
- **Alternativen:**
 - Da es „nur“ 255168 verschiedene Spielverläufe gibt, könnte man alle vorher berechnen und jeweils einen Zug aussuchen, der zu einem günstigen Ausgang führt
 - Durch Transformation (Spiegelung und/oder Drehung) können die Endstellungen auf 138 verschiedene Möglichkeiten reduziert werden, so dass man den Zug so wählt, dass er zu einer günstigen Endstellung passt

Beispiel

Folgende Situationen sind äquivalent:



TicTacToe-Computer



- Der TicTacToe-Computer am M.I.T., ausschließlich aus Tinkertoy™-Bausteinen gebaut
- Er arbeitet mit der zweiten Alternativstrategie

Aufgaben

Aufgabe 11.1

Erweitern Sie das TicTacToe-Programm um folgende Elemente:

- Ändern Sie die Ausgabe so, dass das Spielfeld beispielsweise so wie auf der Abbildung aussieht!
- Alternativ: Schreiben Sie eine GUI!
- Ermöglichen Sie, den Spielstand in eine Datei zu speichern und später wieder rückzulesen!
- Gestalten Sie den Spielverlauf durch Zufallselemente (z.B. mit der Funktion `random()` aus der Standardbibliothek) variabler, ohne der Strategie zu schaden!







+-----+-----+-----+					
	000		X X		
	0 0		X X		
	0 0		X		3
	0 0		X X		
	000		X X		
+-----+-----+-----+					
	X X		X X		000
	X X		X X		0 0
	X		X		0 0
	X X		X X		0 0
	X X		X X		000
+-----+-----+-----+					
	000				X X
	0 0				X X
	0 0		7		X
	0 0				X X
	000				X X
+-----+-----+-----+					

Aufgabe 11.2

Machen Sie sich mit der „Texas Hold'em“-Variante von Poker vertraut (http://de.wikipedia.org/wiki/Texas_Hold'em).

- a) Entwickeln Sie einen Algorithmus, der vor/nach dem **Flop**, nach dem **Turn** und nach dem **River** aus den offenen und Ihren Handkarten berechnet, welche Gewinnkombinationen **sicher** und welche **möglich** sind!

Beispiel:

Hand:   Tisch:     

„Ein Paar“ ist sicher, „zwei Paare“, „Drillinge“, „Full House“ oder „Poker“ ist möglich, alles andere nicht.

- b) Erweitern Sie Ihren Algorithmus/Ihr Programm so, dass die **Wahrscheinlichkeiten** für eine Gewinnkombination ausgegeben wird!

