Programmieren in C++

SS 2018

Vorlesung 4, Dienstag 15. Mai 2018

(Felder, Strings, Zeiger, Debugger, nochmal make)

Prof. Dr. Hannah Bast
Professur für Algorithmen und Datenstrukturen
Institut für Informatik
Universität Freiburg

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem Ü3 — Hüpfball mit Schwerkraft
- Inhalt
  - Felder und Zeiger — Operatoren [] und *
  - Debugging — gdb
  - Hinweise zum Ü4 — Maus, Zellen, zwei Felder
  - Noch mehr zu make — ganz generisch, .PRECIOUS
  - **Übungsblatt 4:** "Game of Life" mit Interaktion via Maus

Erfahrungen mit dem Ü3 1/3

Aber sehr wenig Code, siehe Musterlösung

- Zusammenfassung / Auszüge
  - Mehr Denkarbeit als beim Ü1 und Ü2 notwendig
  - "Graphische Ausgabe ist cool", "befriedigend anzuschauen"
  - Einige Verständnisprobleme, wie Position, Geschwindigkeit und Beschleunigung zusammenhängen (sollen)

    Auf dem Ü3 stand es schon recht detailliert und auch im Forum wurde noch einmal viel Hilfestellung gegeben
  - Ein Video vom dem gewünschten Verhalten hätte geholfen

    Haben wir uns überlegt, aber uns dann dagegen entschieden, um Ihnen nicht die Freude am Endergebnis zu rauben
  - "Ich hatte das Gefühl, als ob ich LOREM IPSUM lesen würde."

Erfahrungen mit dem Ü3 2/3

UNI FREIBURG

- Wahrnehmung gleichzeitiger Ereignisse
  - "Der Weihnachtsmann hat Vorfahrt und ich lasse ihn vorbei"
  - "Ich würde mich wundern, was dieser übergewichtige Mann bei dem heißen Wetter im Kostüm macht"
  - "Natürlich den Osterhasen - im Straßenverkehr konzentriere ich mich schließlich auf mein Handy"
  - "Nehme prinzipiell keine Anrufe von mir unbekannten Wesen an"
  - "Ich sehe weder Weihnachtsmann noch höre ich den Anruf vom Osterhasen, weil sich all meine Neuronen in eckige, springende Bälle auf Terminals verwandelt haben"
  - "Ist das ein Test, wer die Blätter zu Ende liest?"

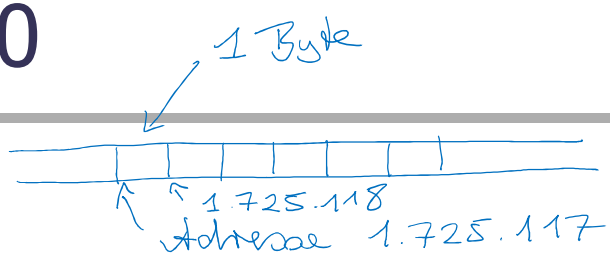
Erfahrungen mit dem Ü3 3/3

UNI FREIBURG

- Psychologische Refraktärzeit
  - Wir können zwei gleichzeitige Eindrücke X und Y nicht bewusst gleichzeitig wahrnehmen, sondern nur hintereinander

    Selbst von verschiedenen Sinnesorganen
  - Ein Eindruck, sagen wir Y, wird dann im Unterbewusstsein zwischengespeichert und muss sozusagen warten
  - Wenn man nach der zeitlichen Abfolge gefragt wird, wird man sagen, dass Y kurz nach X passiert ist
  - Die Informationsverarbeitung unseres Bewusstseins ist also inhärent sequentiell

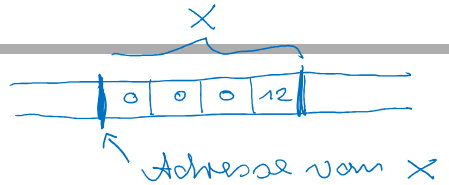
Felder und Zeiger 1/10



- Hauptspeicher
  - Der **Hauptspeicher** von einem Rechner ist (konzeptuell) eine Menge von Speicherzellen
  - Jede Speicherzelle fasst **1 Byte = 8 Bits**

    Also eine Zahl zwischen 0 und 255 (einschließlich)
  - Die Speicherzellen sind fortlaufend nummeriert
  - Die Nummer einer Speicherzelle nennen wir ihre Adresse

Felder und Zeiger 2/10



- Variablen
  - **Variablen** sind Namen für ein Stück Speicher, z.B.

    ```
    int x = 12;  // One int (typically 4 bytes).
    ```

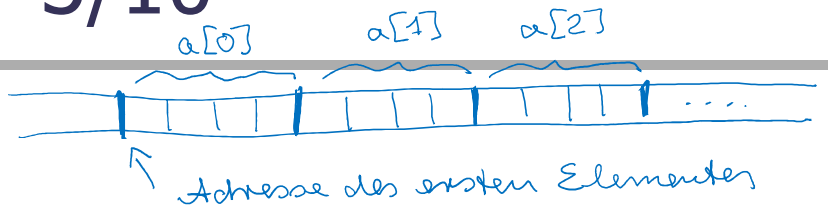
  - Je nach Typ umfasst die Variable eine oder mehrere Bytes ... diese Anzahl bekommt man mit sizeof:

    ```
    printf("%zu\n", sizeof(x));  // Use %zu for type size_t.
    ```

  - Der Variablenname steht für den **Wert** dieser Speicherzellen, interpretiert gemäß Typ
  - Die **Adresse** im Speicher bekommt man mit dem & Operator, mehr dazu in einer späteren Vorlesung

    ```
    printf("%p\n", &x);  // Use %p to print an address.
    ```

Felder und Zeiger 3/10



- Felder
  - **Felder** sind Folgen von Variablen vom selben Typ, auf die man alle mit demselben Namen und einem sogenannten Index zugreifen kann
  - Zugriff auf ein Element des Feldes mit dem [] Operator, wobei das erste Element Index 0 hat, das zweite 1, usw.

```
int a[10];                    // Array of 10 integers.
printf("%zu\n", sizeof(a));   // Prints 4 * 10 = 40.
printf("%d\n", a[2]);         // Prints third element.
```

  - Die Elemente stehen **hintereinander** im Speicher
  - Der Feldname steht für die **Adresse** (nicht den Wert) des ersten Elementes, mehr dazu auf Folie 12

Felder und Zeiger 4/10

UNI FREIBURG

- Initialisierung eines Feldes
  - Wie bei einfachen Variablen, kann man den Feldelementen schon bei der Deklaration Werte zuweisen

```
int a[3] = {1, 2, 3};

int a[3] = {1, 2, 3, 4};  // Too many values -> compiler error.

int a[3] = {1, 2};        // Missing values initialized to zero!

int a[3] = {0};           // Initializes all elements to zero.
```

  - **Achtung:** ohne Initialisierung ist der Inhalt der betreffenden Speicherzellen laut C/C++ Standard beliebig
  - Manche Systeme initialisieren Felder trotzdem mit Nullen, man sollte sich aber nicht darauf verlassen

Felder und Zeiger 5/10

UNI FREIBURG

- Zwei-dimensionale Felder
  - Ein zwei-dimensionales Feld deklariert man z.B. so

    ```
    int b[4][2];  // Space for 4 x 2 ints.
    ```

  - Initialisierung geht dann so

    ```
    int b[4][2] = { {0, 1}, {10, 11}, {20, 21}, {30, 31} };
    ```

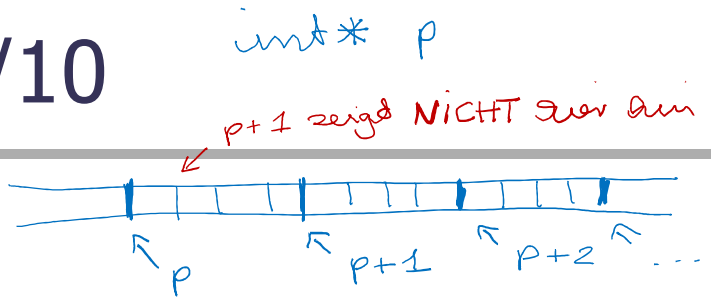
  - Unser Compiler akzeptiert auch das hier:

    ```
    int b[4][2] = { 0, 1, 10, 11, 20, 21, 30, 31 };
    ```

    Das ist aber kein guter Stil !

    Bei manchen Compilern kommt dann eine Warnung

Felder und Zeiger 6/10



- Zeiger
  - **Zeiger** sind Variablen, deren Wert eine **Adresse** ist
  - Bei der Deklaration gibt man an, wie der Inhalt des Speichers an dieser Adresse zu interpretieren ist

    ```
    int* p;  // Pointer to an integer (8 bytes on 64-Bit CPU).
    ```

  - Zugriff auf diesen Wert mit dem * **vor** der Variablen

    ```
    printf("%d\n", *p);  // Print the (int) value pointed to.
    ```

  - Man kann eine Zahl zu einem Zeiger dazu addieren **...** die wird dann automatisch mit der Größe des Typs multipliziert

    ```
    printf("%d\n", *(p + 3));  // Int at p + 3 * sizeof(int).

    p = p + 3;  // Increase the address by 3 * sizeof(int).
    ```

Felder und Zeiger 7/10

UNI FREIBURG

- Zeiger vs. ein-dimensionale Felder in C / C++
  - Den Namen eines ein-dimensionalen Feldes kann man benutzen wie einen Zeiger auf das erste Element des Feldes

```
int a[5] = {3, 9, 2, 14, 5};
int* p = a;
printf("%d\n", *p);  // Will print 3.
```

  - Der Zugriff über [...] macht **exakt** dasselbe wie die auf der vorherigen Folie beschriebene Zeigerarithmetik:

```
int a[5] = {3, 9, 2, 14, 5};
int* p = a;
printf("%d\n", p[3]);      // Prints the fourth element (14).
printf("%d\n", *(p +3)); // Does exactly the same.
```

Felder und Zeiger 8/10

UNI FREIBURG

- Zeiger vs. zwei-dimensionale Felder in C / C++
  - Den Namen eines zwei-dimensionales Feldes kann man benutzen wie einen Zeiger auf ein ein-dimensionales Feld

```
int b[4][2] = { {0, 1}, {10, 11}, {20, 21}, {30, 31} };
int (*q)[2] = b;                // Pointer to array of size 2.
printf("%d\n", sizeof(q));      // Prints 8 (size of an address).
printf("%d\n", b[3][1]);        // Prints 31.
printf("%d\n", q[3][1]);        // Exactly the same.
```

  - Man könnte auch einfach die Adresse des ersten Elementes nehmen, verliert dann aber die 2D-Arithmetik:

```
int* p = &b[0][0];              // Same address as b and q.
printf("%d\n", p[3][1]);        // Does not compile.
printf("%d\n", p[7]);           // This does, and prints 31.
```

Felder und Zeiger 9/10

UNI FREIBURG

- Zeichenketten / Strings
  - Eine **Zeichenkette** ist auch nur ein Feld bzw. Zeiger ... und zwar von Elementen vom Typ **char** = 1 Zeichen

    ```
    char a[4] = {'D', 'o', 'o', 'f'};

    char* p = a + 3;  // Points to the cell containing the 'f'.
    ```

  - Kann man auch einfacher so initialisieren

    ```
    const char* s = "Doof";  // s points to the byte with the 'D'.
    ```

    Ohne das const gibt es eine Compiler-Warnung, siehe VL6

  - Strings in C/C++ sind **null-terminated**, d.h. bei "Doof" wird Platz für **fünf** Zeichen gemacht, und am Ende steht **\x00**

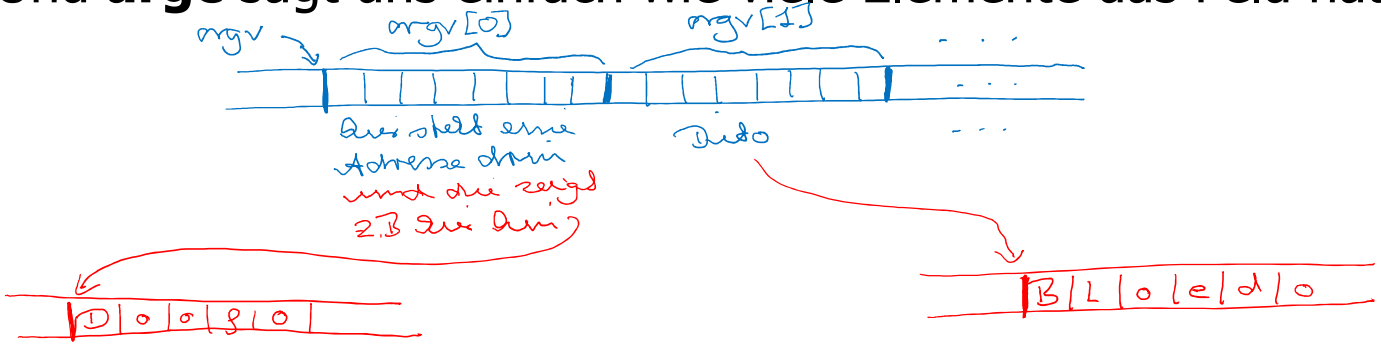
    Damit man weiß, wo die Zeichenkette aufhört

Felder und Zeiger 10/10

- Felder von Zeichenketten
  - Das erklärt auch den Typ von **argv** in der main Funktion

    int main(int argc, **char** argv**)
  - Und zwar ist **char**** ein Zeiger auf Werte vom Typ **char***
  - Mit anderen Worten: **argv** ist ein Feld von Zeigern, die auf Zeichenketten (die Felder von Zeichen sind) zeigen

    Und **argc** sagt uns einfach wie viele Elemente das Feld hat



Debugging 1/6

UNI FREIBURG

- Fehler im Programm kommen vor
  - Und jetzt, wo Sie Felder und Zeiger kennen, werden Sie gemeine **seg**mentation **faults** produzieren
  - Das passiert bei versuchtem Zugriff auf Speicher, der Ihrem Programm gar nicht gehört, zum Beispiel

```
int* p = NULL;    // Pointer to address 0.
*p = 42;          // Will produce a segmentation fault.
```

  - Schwer zu debuggen, es kommt dann einfach etwas wie:

```
Segmentation fault (core dumped)
```

    Ohne Hinweis auf die Fehlerstelle im Code (gemein)
  - Manche Fehler sind zudem nicht deterministisch, weil sie von nicht-initialisiertem Speicherinhalt abhängen

Debugging 2/6

UNI FREIBURG

- Methode 1: printf
  - printf statements einbauen
    - an Stellen wo der Fehler vermutlich auftritt
    - von Variablen wo man denkt, dass etwas falsch läuft
  - **Vorteil:** geht ohne zusätzliches Hintergrundwissen
  - **Nachteil 1:** man muss jedes mal neu kompilieren, das kann bei größeren Programmen lange dauern
  - **Nachteil 2:** kann dauern, bis man sich so an die Stelle herangetastet hat, wo der Fehler auftritt

Debugging 3/6

UNI FREIBURG

- Methode 2: gdb
  - Mit dem **gdb**, das ist der **GNU debugger**
  - Der kann so Sachen wie
    - Anweisung für Anweisung durch das Programm gehen
    - Sogenannte breakpoints im Programm setzen und zum nächsten breakpoint springen
    - Werte von allen möglichen Variablen ausgeben
  - Das wollen wir jetzt mal anhand eines Beispiels machen
  - **Vorteil:** viel interaktiver als mit printf statements
  - **Nachteil:** ein paar gdb Kommandos merken

Debugging 4/6

UNI FREIBURG

- Ein paar grundlegende gdb Kommandos
  - **Wichtig:** Programm kompilieren mit der –g Option!
  - gdb aufrufen, z.B. gdb ./ArraysAndPointersMain
  - Programm starten mit run <command line arguments>
  - stack trace (nach seg fault) mit backtrace oder bt
  - breakpoint setzen, z.B. break Number.cpp:47
  - breakpoints löschen mit delete oder d
  - Weiterlaufen lassen mit continue oder c
  - Wert einer Variablen ausgeben, z.B. print x oder p i

Debugging 5/6

UNI FREIBURG

- Weitere gdb Kommandos
  - Nächste Zeile im Code ausführen step bzw. next

    step folgt Funktionsaufrufen, next führt sie ganz aus
  - Aktuelle Funktion bis zum Ende ausführen finish
  - Aus dem gdb heraus make ausführen make
  - Kommandoübersicht / Hilfe help oder help all
  - gdb verlassen mit quit oder q
  - Wie in der bash command history mit Pfeil hoch / runter

    Es geht auch Strg+L zum Löschen des Bildschirmes

Debugging 6/6

UNI FREIBURG

- Methode 3: valgrind
  - Mit Zeigern kann es schnell passieren, dass man über ein Feld hinaus liest / schreibt … oder sonst wie unerlaubt auf Speicher zugreift
  - Solche Fehler findet man gut mit **valgrind**

    Das wäre für heute zu viel auf einmal und machen wir erst in Vorlesung 6 (dynamische Speicherverwaltung)

Noch mehr zu Make 1/3

UNI FREIBURG

- Generisches Makefile
  - Mit den Funktionen aus der letzten Vorlesung kann man das Makefile vollständig generisch/automatisch machen
  - Das heißt, die diversen Header, .o Dateien und ausführbaren Dateien werden automatisch ermittelt
  - Für die zukünftigen Übungsblätter kann man dieses Makefile dann einfach vom letzten Blatt kopieren, ohne Anpassung

    Ausnahme: wenn eine Bibliothek (nicht mehr) gebraucht wird
  - Das ist Aufgabe 5 vom Ü4

Noch mehr zu Make 2/3

UNI FREIBURG

- Das .PRECIOUS target
  - Je nach Konfiguration von make, kann es sein, dass am Ende von make compile die **.o** Dateien gelöscht werden
  - Grund: make unterscheidet zwischen "Endprodukten" und "Zwischenprodukten", zum Beispiel:
    - Bei make BallMain ist BallMain das Endprodukt
    - Die ganzen **.o** Dateien sind Zwischenprodukte ... weil man sie zum Ausführen der BallMain nicht braucht
  - Je nach System löscht make manche Zwischenprodukte
  - **Lösung:** folgende Zeile am Anfang vom Makefile

```
.PRECIOUS: %.o
```

Noch mehr zu Make 3/3

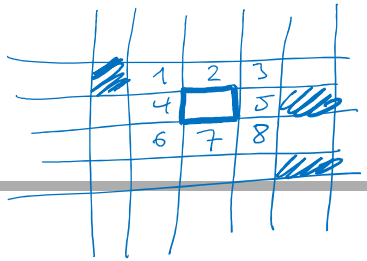
UNI FREIBURG

- Das .PHONY target
  - Wir haben ja schon seit der ersten Vorlesung die vier "phony" targets: compile, test, checkstyle, clean
  - Damit werden ja nicht die entsprechenden Dateien gebaut, sondern wir benutzen die als "shortcut"
  - Problem: falls es eine dieser Dateien tatsächlich gäbe, z.B. compile, würde make compile einfach nichts machen
  - **Lösung:** folgende Zeile am Anfang vom Makefile

    .PHONY: compile test checkstyle clean

    Damit wird das entsprechende make immer ausgeführt, unabhängig davon, ob es die Dateien gibt oder nicht

Hinweise zum Ü4 1/6



UNI FREIBURG

- Game of Life https://en.wikipedia.org/wiki/Conways_Game_of_Life
  - Ein sehr einfacher zellulärer Automat
  - In jedem Schritt ist jede Zelle **tot** oder **lebendig**
  - Jede Zelle hat genau acht Nachbarn
  - Eine tote Zelle mit genau drei lebendigen Nachbarn ist im nächsten Schritt lebendig
  - Eine lebendige Zelle mit weniger als zwei oder mehr als drei lebendigen Nachbarn ist im nächsten Schritt tot
  - Für das Ü4 sollen zu Beginn alle Zellen tot sein und man soll das Spiel jederzeit anhalten und den Zustand einer Zelle durch Mausklick invertieren können

  Das implementieren wir jetzt mal zusammen

Hinweise zum Ü4 2/6

UNI FREIBURG

- Ncurses und die Maus, Initialisierung
  - Um mit ncurses Mausklicks verarbeiten zu können, brauchen wir bei der Initialisierung eine Zeile mehr

```
#include <ncurses.h>

initscr();                  // Initialization.
cbreak();                   // Don't wait for RETURN.
noecho();                   // Don't echo key presses.
curs_set(false);            // Don't show the cursor.
nodelay(stdscr, true);      // Don't wait until key pressed.
keypad(stdscr, true);       // for KEY_LEFT, KEY_UP, etc.

mousemask(ALL_MOUSE_EVENTS, NULL);
```

Hinweise zum Ü4 3/6

UNI FREIBURG

- Ncurses und die Maus, Klick abfragen
  - Nach der Initialisierung auf der vorherigen Folie kriegt man die Position des Mausklicks mit folgendem Code:

```
// Check if mouse clicked and get click coordinates.
MEVENT event;
int key = getch():
if (getmouse(&event) == OK) {
   if (event.bstate & BUTTON1_CLICKED) {
      int x = event.x;  // x-coordinate of click (col index)
      int y = event.y;  // y-coordinate of click (row index)
   }
 }
```

Hinweise zum Ü4 4/6

UNI FREIBURG

- Speichern der Zustände der Zellen
  - Für das Ü4 sollen Sie den aktuellen Zustand der Zellen in einem **zwei-dimensionalen** Feld speichern
  - Da wir zur Kompilierzeit nicht wissen, wie groß der Bildschirm ist, definieren wir eine genügend große Konstante

    ```
    const int MAX = 200;
    bool cells[MAX][MAX];
    ```

  - Ein Feldelement soll **true** sein wenn die entsprechende Zelle lebendig ist, und **false** wenn sie tot ist
  - **Optional:** zentrieren Sie das Feld, so dass die Zelle cells[MAX/2][MAX/2] in der Bildschirmmitte liegt

    Zustandsfeld geht dann rundherum über Bildschirm hinaus

Hinweise zum Ü4 5/6

UNI FREIBURG

- Update des Zustandsfeldes
  - Um die neuen Zustände der Zellen berechnen zu können, brauchen Sie **zwei** Felder (mit den gleichen Dimensionen):

    Ein Feld für den bisherigen Zustand

    Ein Feld für den neuen Zustand
  - Nach dem Update-Schritt könnten sie dann das aktuelle Feld mit den neuen Zuständen überschreiben

    Das ist aber ineffizient (es wird mehr kopiert als nötig)
  - Besser: Sie merken sich, welches der beiden Felder das aktuelle ist (abwechselnd das eine und das andere)

    Das geht ganz prima mit einem Zeiger, überlegen Sie wie!

Hinweise zum Ü4 6/6

UNI FREIBURG

- Mehrere Zeichen pro Zelle
  - Ein Zeichen pro Zelle sieht etwas "dünn" aus und auf den meisten Terminals alles andere als quadratisch
  - Lösung: einfach **sx mal sy** Zeichen pro Zelle malen, im einfachsten Fall einfach zwei Zeichen nebeneinander

    Das ist freiwillig, aber auch nicht besonders schwierig und sieht definitiv **viel** besser aus

    Wir hatten das ja für das Ü3 für den Ball auch schon gemacht, nur dass wir da keine Interaktion (mit der Maus hatten)

Literatur / Links

UNI FREIBURG

- Felder / Arrays
  - http://www.cplusplus.com/doc/tutorial/arrays
- Zeiger / Pointers
  - http://www.cplusplus.com/doc/tutorial/pointers
- Zeichenketten / Strings
  - http://www.cplusplus.com/doc/tutorial/ntcs
- Debugger / gdb
  - http://sourceware.org/gdb/current/onlinedocs/gdb