Programmieren in C++

SS 2022

Vorlesung 4, Dienstag 17. Juni 2022

(Felder, Strings, Zeiger, Debugger, Game of Life)

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 — Die Horrorschlange
  - Fragen auf dem Forum — Erinnerung an die Regeln
- Inhalt
  - Felder und Zeiger — Operatoren [ ] und *
  - Debugging — gdb
  - Hinweise zum Ü4 — Maus, Zellen, zwei Felder
  - Noch etwas zu make — .PRECIOUS und –O3
  - **Ü4:** "Game of Life" mit Interaktion via Maus

Erfahrungen mit dem Ü3 1/3

UNI FREIBURG

- Feedback zur Aufgabe
  - Die meisten fanden die Aufgabe an sich sehr schön
  - Für ca. 1/3 kein Problem, für ca. 1/3 viel Arbeit aber machbar, ca. 1/3 mehr Probleme; sehr viele haben **zu spät** angefangen

    Werden wir bei den nächsten Übungsblättern berücksichtigen!

    Im Vergleich zu den Vorjahren diesmal mehr Probleme. Wie gesagt: ohne Grundkenntnisse im Programmieren ist diese Lehrveranstaltung viel mehr Arbeit als 6 ECTS

    Vermutung: Ist vielleicht irgendetwas in der "Einführung in die Programmierung" im WS 21/22 anders gelaufen?
  - Das Testen war aufwendig und dazu wurde nichts vorgemacht

    Stimmt + sorry, das war auch geplant, aber dafür fehlte in der V3 leider am Ende die Zeit

Erfahrungen mit dem Ü3 2/3

UNI FREIBURG

- Zitate aus den Rückmeldungen

  "Mir hat das Blatt viel Spaß gemacht. Die Vorlesung hat gut darauf vorbereitet und es war eine wirklich schöne Aufgabe"

  "Aufgaben im Grunde gut machbar und ohne Leichtsinnsfehler vermutlich auch deutlich schneller machbar gewesen"

  "Hat trotz großem Aufwand sehr viel Spaß gemacht, da richtiges Erfolgserlebnis, als Schlange auf dem Bildschirm aufgetaucht ist"

  "Das Blatt war deutlich komplizierter als in den vergangenen zwei Wochen, dafür war die Aufgabe aber auch interessanter"

  "Etwas mehr zum Testen in der Vorlesung wäre schön"

  "Es gibt noch sehr viele Bugs und es läuft nicht alles wie es soll. Frustrierendes Ergebnis nach so viel Arbeit ☹"

  "Zeitaufwand 20 Stunden. Alles Scheiße. Draußen Sonnenschein"

Erfahrungen mit dem Ü3 3/3

UNI FREIBURG

- Für die, die Schwierigkeiten haben
  - Versuchen Sie größere Programme **inkrementell** zu schreiben

    Ein paar Zeilen Funktionalität hinzufügen und es dann wieder zum Laufen bekommen, nicht alles auf einmal und dann Chaos
  - Wir versuchen, Tempo und Schwierigkeit anzupassen so gut es geht, können es aber bei einer Veranstaltung mit über 200 aktiv Teilnehmenden nicht allen recht machen
  - Der Schwierigkeitsgrad wird nicht weiter so stark steigen, wie gefühlt für manche vom Ü2 zum Ü3

    Zum Beispiel sollte das Ü4 konzeptionell nicht schwieriger sein als das Ü3 und auch eher weniger statt mehr Arbeit
  - Frust ist verständlich, aber versuchen Sie bitte sachlich und konstruktiv zu bleiben mit Ihrer Kritik

Fragen auf dem Forum

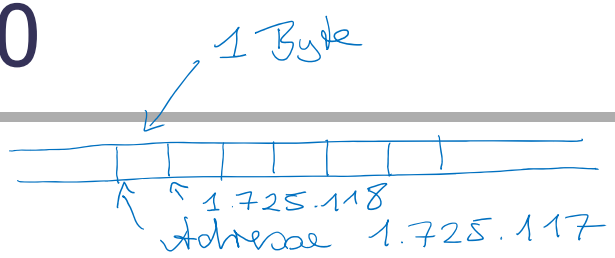
UNI FREIBURG

- Bitte die Anleitung auf dem Wiki beachten
  - Es wird sehr viel auf dem Forum gefragt, was super ist
  - Viele Fragen kommen **allerdings doppelt und dreifach**
  - Geben Sie vor einer Frage bitte kurz ein paar Stichworte in die Forumssuche ein, um zu schauen, ob die Frage schon einmal gestellt (und beantwortet) wurde

    https://daphne.informatik.uni-freiburg.de/forum/search.php
  - Es werden auch relativ viele Fragen gestellt, die genau so schon in der Vorlesung behandelt wurde

    Das deutet daraufhin, dass die Vorlesung bzw. die Materialien dazu nicht ganz so aufmerksam studiert wurden, wie man das vielleicht hätte tun sollen

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).
    ```

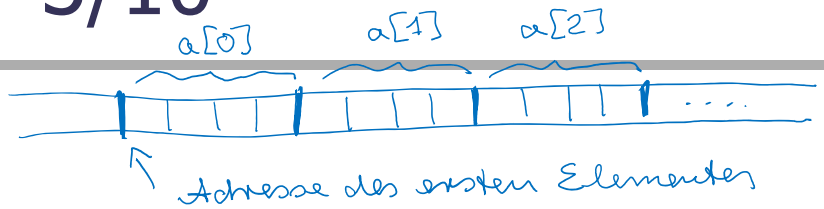
  - Je nach Typ umfasst die Variable eine oder mehrere Bytes ... diese Anzahl bekommt man mit sizeof:

    ```
    printf("%zu", 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 zu & in einer späteren Vorlesung)

    ```
    printf("%p", &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", sizeof(a));   // Prints 4 * 10 = 40.
printf("%d", 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

- Zweidimensionale Felder
  - Ein zweidimensionales Feld deklariert man z.B. so

```
int b[5][2];  // Space for 5 x 2 ints.
```

  - Initialisierung geht dann so

```
int b[5][2] = { {0, 1}, {2, 3}, {4, 5}, {6, 7}, {8, 9} };
```

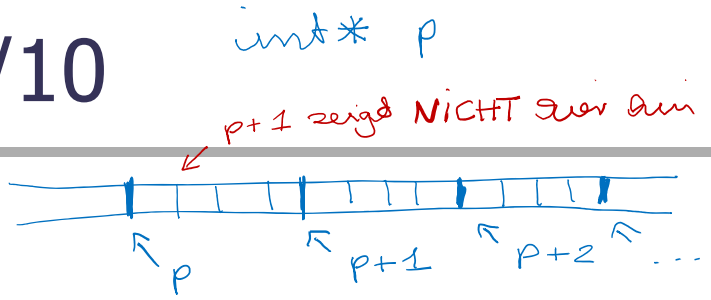
  - Unser Compiler akzeptiert auch das hier (aber kein guter Stil)

```
int b[5][2] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

  - Zweidimensionale Felder sind in C/C++ aber **nicht äquivalent** zu eindimensionalen Feldern

```
printf("%d", b[3][1]);     // Print elem at position 3 * 2 + 1 = 7.
printf("%d", b[7]);        // Does not compile.
```

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", *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", *(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

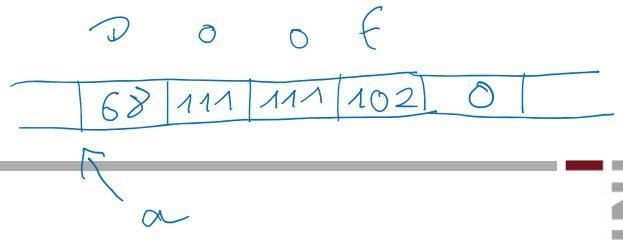
- Zeiger vs. Felder in C / C++
  - Den Namen eines eindimensionalen Feldes kann man oft benutzen wie einen Zeiger auf das erste Element des Feldes

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

Felder und Zeiger 8/10



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

    ```
    char a[5] = { 'D', 'o', 'o', 'f', 0 };

    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 V6

  - Strings in C/C++ sind **null-terminated**, d.h. bei "Doof" wird Platz für **fünf** Zeichen gemacht, und am Ende steht 0

    Damit man weiß, wo die Zeichenkette aufhört

Felder und Zeiger 9/10 $(2 \cdot (5-7)18+5)^2$

UNI FREIBURG

- Zeiger auf Zeiger
  - Zeiger können auf alles zeigen, auch andere Zeiger
  - Ein typisches Beispiel sind Zeiger auf Zeichenketten (und Zeichenketten sind ja vom Typ Zeiger auf char)

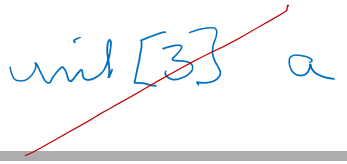
```
char* s[2] = { "bla", "blubb" };   // Each string is a char*.
printf("%s", s[1]);                // Prints "blubb".

char** p = s;                      // Array of X is like pointer to X.
printf("%s", s[1]);                // Also prints "blubb".
```

  - char** benutzt man, um Argumente von der Kommandozeile an die "main" Funktion zu übergeben (argc sagt wie viele)

```
int main(int argc, char** argv) { ... }
```

    Mehr dazu in einer späteren Vorlesung (V6 oder V7)

Felder und Zeiger 10/10



UNI FREIBURG

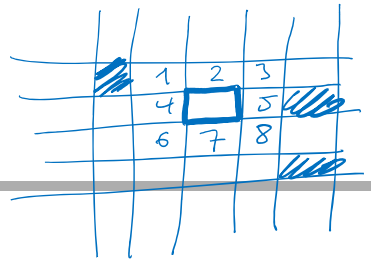
- Zeiger auf Zeiger oder auf Felder
  - Bei Zeiger auf Feld ist die Syntax etwas verwirrend, weil man den * vor die Variable schreibt und die [ ] danach

```
int a[3] = { 1, 2, 3 };      // An array of three ints.
int (*p)[3];                 // Pointer to an array of three ints.
p = &a;                      // Let p point to a.

int* q[3];                   // An array of three pointers to int.
```

  - Für zweidimensionale Felder analog (braucht man fürs Ü4)

```
bool grid[5][2] = { … };  // A 2D array of 5 times 2 bools.
bool (*p)[5][2];          // A pointer to such an array.
p = &grid;                // Let p point to the grid array.

(*p)[3][1];               // Access an element of the
                          // 2D array pointed to by p.
```

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

  Wir fangen jetzt mal an, das gemeinsam zu implementieren

Hinweise zum Ü4 2/6

UNI FREIBURG

- Ncurses und die Maus, Initialisierung
  - Um mit ncurses Mausklicks verarbeiten zu können, brauchen wir bei der Initialisierung zwei Zeilen 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);
mouseinterval(0);           // For faster response time.
```

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_PRESSED) {
      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 = 1000;
bool cells[MAX][MAX];
```

  - Ein Feldelement soll **true** sein wenn die entsprechende Zelle lebendig ist, und **false** wenn sie tot ist
  - Optional: das Feld kann größer sein, als der Teil, den wir auf dem Bildschirm anzeigen, das ist für das Spiel sogar gar nicht schlecht

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

    Für den Vorlesungscode malen wir einfach zwei (invertierte) Leerzeichen nebeneinander, das sieht auf den meisten Terminals schon ziemlich quadratisch aus

    Man muss dann die Bildschirmkoordinaten in die "logischen" Koordinaten der Zellen umrechnen und umgekehrt

```
logicalRow = screenRow;
logicalCol = screenCol / 2;

mvprintw(logicalRow, logicalCol * 2, "  ");
```

Debugging 1/6

UNI FREIBURG

- Fehler im Programm kommen vor
  - Und jetzt, wo Sie Felder und Zeiger kennen, werden Sie öfter auf diese Fehlermeldung stoßen

    Segmentation fault (core dumped)
  - 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, weil es überhaupt keinen Hinweis gibt auf die Stelle im Code, an der das aufgetreten ist

    Das ausführbare Programm ist ja Maschinencode
  - 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 **d**elete (es reicht auch d)
  - Weiterlaufen lassen mit **c**ontinue (es reicht auch c)
  - Wert einer Variablen ausgeben, z.B. **p**rint x

Debugging 5/6

UNI FREIBURG

- Weitere gdb Kommandos
  - Nächste Zeile im Code ausführen **s**tep bzw. **n**ext

    step folgt Funktionsaufrufen, next führt sie ganz aus
  - Aktuelle Funktion bis zum Ende ausführen **fin**ish
  - Code an der aktuellen Stelle zeigen **l**ist
  - Aus dem gdb heraus make ausführen make
  - Kommandoübersicht / Hilfe **h**elp oder **h**elp all
  - gdb verlassen mit **q**uit
  - 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 etwas zu make und g++ 1/2

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 SnakeMain ist SnakeMain das **Endprodukt**
    - Die ganzen **.o** Dateien sind **Zwischenprodukte** ... weil man sie zum Ausführen der SnakeMain nicht braucht
  - Je nach System löscht make manche Zwischenprodukte
  - Die folgende Zeile am Anfang vom Makefile verhindert das

    .PRECIOUS: %.o

Noch etwas zu make und g++ 2/2

UNI FREIBURG

- Kompilieren mit –Wall –Wextra –O3
  - Der g++ Compiler ist stark konfigurierbar, siehe man g++
  - Bis auf Weiteres empfehlen wir folgende Optionen:

    –Wall gibt mehr Warnungen aus als normal (allerdings bei weitem nicht alle, trotz des Namens der Option)

    –Wextra gibt noch mehr Warnungen aus

    –O3 erzeugt schnelleren Code, dazu in einer späteren Vorlesung noch mehr

    Achtung: beim Debuggen kein –O3 verwenden, da sonst die 1-1 Verbindung zwischen C++ Code und Maschinencode verloren geht

Literatur / Links

- 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