Programmieren in C++

SS 2012

Vorlesung 4, Dienstag 22. Mai 2012

(Felder, Strings, Zeiger, const, Debugger)

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

UNI FREIBURG

Blick über die Vorlesung heute

- Organisatorisches
  - Erfahrungen mit dem 3. Übungsblatt
  - Hinweis zu Fragen im Forum
- Themen heute
  - Felder, Strings, Zeiger ... und dass das alles dasselbe ist
  - Die Operatoren [] und *
  - const bei Deklarationen
  - Debugging mit gdb
  - Übungsaufgabe: vom Ball zum Wurm

Erfahrungen mit dem Ü3 (Animation)

- Zusammenfassung / Auszüge Stand 22.5 13:31
  - Hat den meisten Spaß gemacht
  - Escape-Codes sind cool
  - Freiwilliger Aufgabenteil war gut ... Erweiterungen: größerer Ball, variable Geschwindigkeit, Abbruch durch Tastendruck, ...
  - Vorlesung: zuviel Allgemeines / Fragen am Anfang, dann am Ende zu wenig Zeit für anderes
  - X für row und Y für column ist verwirrend, besser andersrum?
  - Verschwindender Cursor war nervig
  - Makefile löscht .o Dateien automatisch ... doof bzw. warum?
  - Generisches Makefile ist unübersichtlich / schwer zu verstehen

Erfahrungen mit dem Ü3 (Animation)

UNI FREIBURG

- Fortsetzung
  - "5 Minuten Pause sind super"
  - "5 Minuten Pause sind doof"
  - "Scheinbar bringen die Unit Tests wirklich was …"
  - Glücklich über die Vorlesungsaufzeichnung
  - "Vielleicht kann die Professorin ein bisschen langsamer auf der Tastatur spielen"
  - An den #includes, sieht man doch schon welche .cpp Datei welche .h Dateien benötigen … warum muss man das im Makefile dann nochmal extra sagen?
  - Warum werden bei mir mit ll keine .swp Dateien angezeigt?

Makefile .PRECIOUS

- Manchen ist aufgefallen
  - … dass nach make compile die .o Dateien von selber wieder gelöscht werden, Grund dafür:
  - make unterscheidet zwischen "Endprodukten" und "Zwischenprodukten"
  - Zum Beispiel bei make AnimationMain
    - ist AnimationMain das Endprodukt
    - die ganzen .o Dateien sind Zwischenprodukte … weil man sie zum Ausführen der AnimationMain nicht braucht
  - Je nach System löscht make manche Zwischenprodukte
  - Lösung: folgende zusätzliche Zeile am Anfang vom Makefile

    .PRECIOUS: %.o

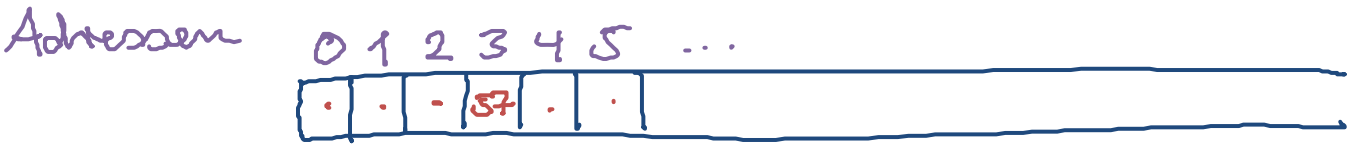
Fragen im Forum

UNI FREIBURG

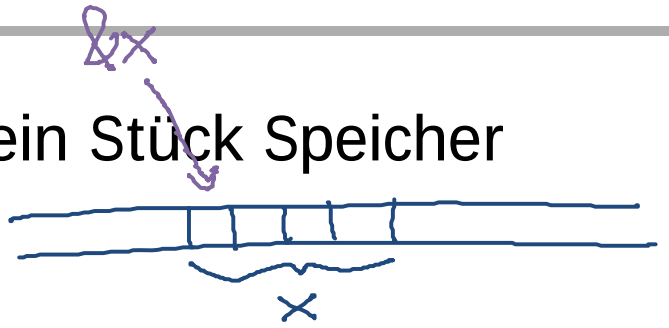
- Wenn irgendwas bei Ihnen nicht funktioniert
  - … gerne fragen, aber dabei **bitte** folgendes beachten:
  - Vollständige Angabe der Fehlermeldung
    - Ausnahme: seitenlange Fehlermeldungen z.B. vom Linker, dann reicht ein Auszug
  - Oft bezieht sich die Fehlermeldung auf eine Zeile im Code (und die Zeile steht explizit in dabei)
    - … dann bitte diese Codezeile auch mit angeben
    - … und evtl. anderen für den Fehler relevanten Code
    - Bei mehr als einer Zeile bitte Angabe wo die Zeile von der Fehlermeldung ist

Speicher

- Der Hauptspeicher von einem Rechner
  - Ist (konzeptuell) einfach eine Menge von Speicherzellen
  - Jede Speicherzelle fasst 1 Byte = 8 Bits
    - also eine Zahl zwischen 0 und 255 (einschließlich)
  - Die Speicherzellen sind fortlaufend durchnumeriert
  - Die Nummer einer Speicherzelle nennen wir ihre Adresse



Variablen



- Variablen sind Namen für ein Stück Speicher
  - Zum Beispiel

    ```
    int x = 12;
    ```

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

    ```
    printf("%zu\n", sizeof(x));
    ```

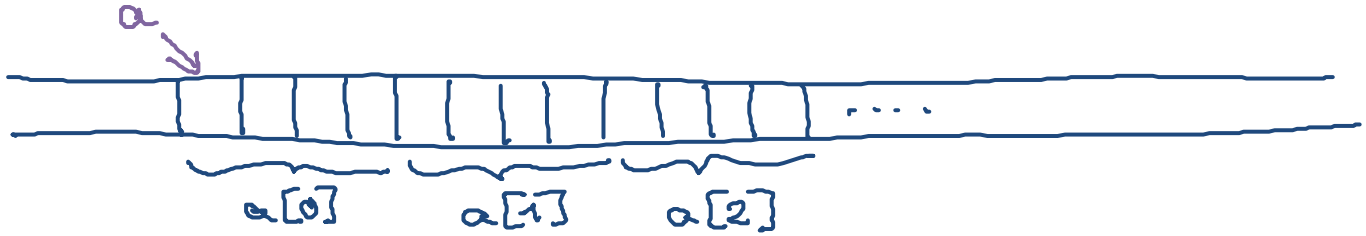
  - Der Variablenname steht für den **Wert** diesen Speicherzellen, interpretiert gemäß Typ
    - die **Adresse** bekommt man mit &x
    - zum & Operator mehr in einer späteren Vorlesung

Felder, engl. Arrays 1/2

- 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. Beware: not initialized.
printf("%d\n", a[2]);  // Print the *third* element.
```

  - Die Elemente stehen **hintereinander** im Speicher
  - Der Variablenname steht für die **Adresse** (nicht den Wert) des ersten Elementes

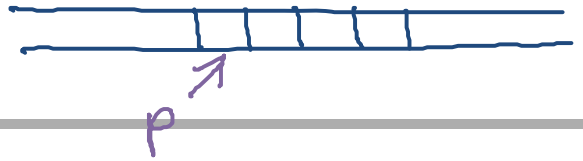


Felder, engl. Arrays 2/2

- Initialisierung
  - 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};  // No compiler error for too few values.
```

Zeiger 1/2



- 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 (usually 4 or 8 bytes).
    ```

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

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

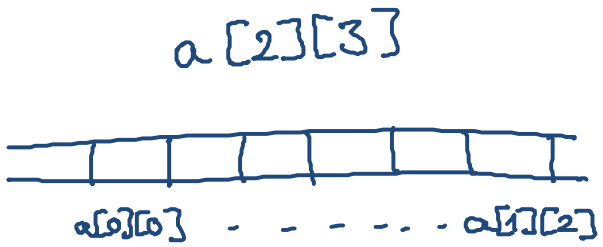
  - Man kann aber auch den [] Operator benutzen

    ```
    printf("%d\n", p[0]);  // p[0] and *p are synonymous.
    printf("%d\n", p[2]);  // Works, but probably undesired.
    ```

Zeiger 2/2

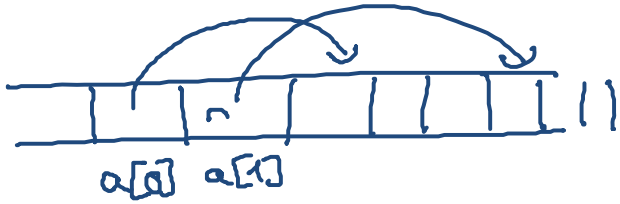
- Zeiger und Felder sind **exakt** dasselbe in C / C++
  - Insbesondere könnte man schreiben

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



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

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



Const 1/2

- Const steht vor Deklarationen einer Variable

```
const int Pi = 3;
```

  - Das const bedeutet, dass man den Wert dieser Variablen nicht mehr verändern darf
  - Wann immer Variablen nur zum Lesen gedacht sind, sollte man const davor schreiben, als Schutz
  - Ein const **vor** einem Zeiger bedeutet, dass man den Speicher, auf den der Zeiger zeigt, **nicht** verändern darf, aber schon wohin der Zeiger zeigt

```
const int* p = &x;  // Adress of variable x.
*p = 1;  // Will produce a compiler error.
p = &y;  // This is fine though.
```

Const 2/2

- Das Umgekehrte gibt es auch
  - Ein const **nach** einem Zeiger bedeutet, dass man den Speicher, auf den der Zeiger zeigt, verändern darf, aber nicht wohin er zeigt

```
int* const p = &x;  // Adress of variable x.

*p = 1;  // This is fine now.

p = &y;  // This will give a compiler error now.
```

  - Aber das braucht man äußerst selten und ist hier nur der Vollständigkeit halber erwähnt

Zeichenkette, engl. Strings

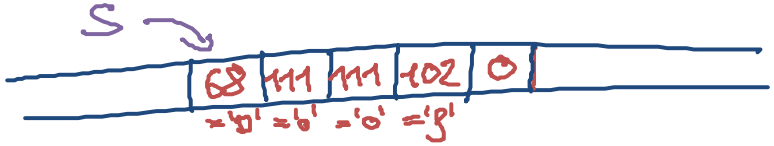
- Eine Zeichenkette ist auch nur ein Feld bzw. Zeiger
  - … und zwar von Elementen vom Typ char = 1 Zeichen

    ```
    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'.
    ```

    Achtung: ohne das const gibt es eine Compiler-Warnung!
  - Strings in C / C++ sind null-terminated, d.h. bei "Doof" wird Platz für **fünf** Zeichen gemacht, und am Ende steht eine 0
    - damit man weiß, wo die Zeichenkette aufhört



argv

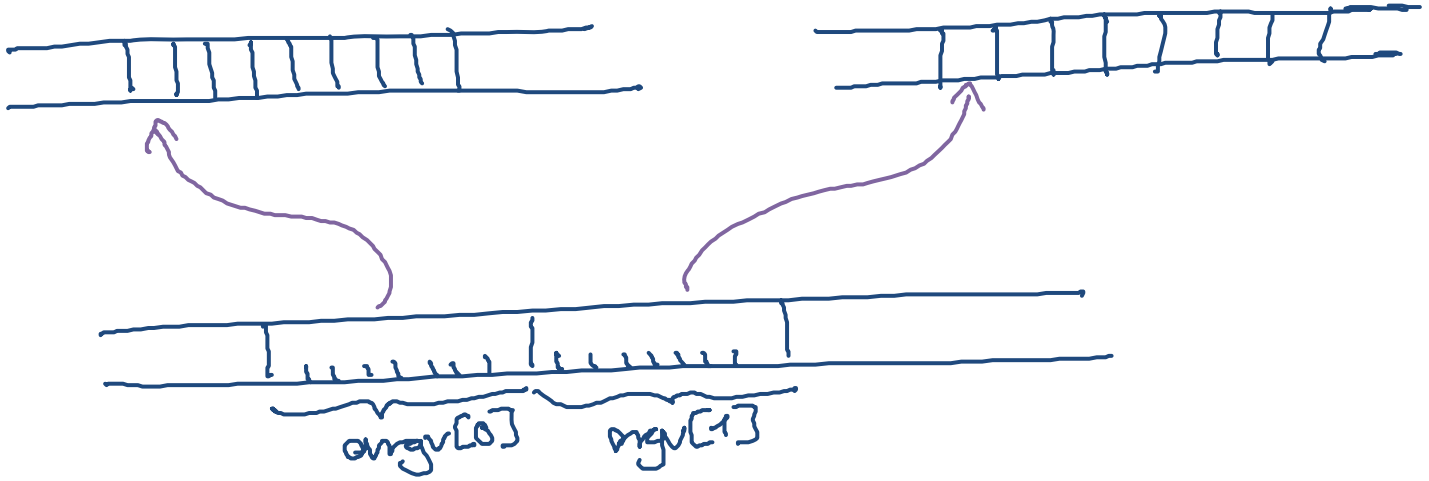
- Jetzt können wir auch das argv von main verstehen

  int main(int argc, **char** argv**)

  – char** ist ein Zeiger auf Werte vom Typ char*

  – Bzw. ein Feld von Elementen vom Typ char*, d.h. jedes Element zeigt auf eine Zeichenkette

  – Und argc sagt einfach wieviele Elemente es gibt



Debugging 1/4

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

```
int* p = NULL;  // Address 0 is written as NULL.
*p = 42;  // Will produce a segmentation fault.
```

  - Schwer zu debuggen, weil man nicht weiß, wo im Programm der Fehler aufgetreten ist, man bekommt einfach nur die Fehlermeldung segmentation fault
  - Manche Fehler sind zudem "nicht-deterministisch", d.h. mal treten sie auf und mal nicht

Debugging 2/4

- Methode 1 ("printf")
  - printf statements einbauen
    - an Stellen wo der Fehler vermutlich auftritt
    - von Variablen wo man denkt, dass etwas falsch läuft
  - Haupt-Vorteil
    - Geht ohne zusätzliches Hintergrundwissen
  - Haupt-Nachteil
    - Man muss jedesmal neu kompilieren
    - Das kann bei größeren Programmen lange dauern
    - Außerdem kann es dauern, bis man sich so an die Stelle herangetastet hat, wo der Fehler auftritt

Debugging 3/4

- 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
  - Haupt-Vorteil:
    - Viel interaktiver als mit printf statements
  - Haupt-Nachteil:
    - Man muss sich ein paar gdb Kommandos merken

Debugging 4/4

- Ein paar grundlegende GDB Kommandos
  - **Wichtig:** Programm kompilieren mit der –g Option!
  - gdb aufrufen, z.B. gdb ./ListProcessingMain
  - 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
  - Nächste Programmzeile ausführen step oder s
  - Wert einer Variablen ausgeben, z.B. print x oder p i
  - Kommandoübersicht / Hilfe mit help oder help all
  - gdb verlassen mit quit oder q
  - Wie in der bash command history mit Pfeil hoch / runter

assert

- Gerade wenn man Felder / Zeiger benutzt
  - … gibt es viele Gelegenheiten etwas falsch zu machen
  - Diese Fehler können sich auf die merkwürdigsten Arten manifestieren und sind auch mit gdb schwer zu finden
  - Deshalb kritische Bedingungen mit assert überprüfen

```
const int MAX_WORM_LENGTH = 100;
int posX[MAX_WORM_LENGTH];
int posY[MAX_WORM_LENGTH];
const char* worm = argv[1];  // Get worm string from cmd line.
int n = strlen(worm);  // Length of the worm string.
assert (n < MAX_WORM_LENGTH);
for (int i = 0; i < n; i++) {
  posX[i] = maxX / 2;
  posY[i] = (maxY – n) / 2 + i;
}
```

Übungsblatt — Hilfestellung 1/2

- Rotation eines Richtungsvektors
  - Wie rotiert man dirX, dirY um 45° nach links oder rechts?
  - Man könnte für alle **acht** Fälle explizit die neue Richtung zuweisen, in einem riesigen if ... else if ... else if ...
  - Aber das geht auch viel eleganter … mit etwas Mathematik

Rotationsmatrix $\begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}$

für $\alpha = 45°$ $\begin{pmatrix} \frac{1}{2}\sqrt{2} & -\frac{1}{2}\sqrt{2} \\ \frac{1}{2}\sqrt{2} & \frac{1}{2}\sqrt{2} \end{pmatrix} = \frac{1}{2}\sqrt{2}\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}$

$\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} dirX \\ dirY \end{pmatrix} = \begin{pmatrix} dirX - dirY \\ dirX + dirY \end{pmatrix}$ → auf Länge 1 normalisieren

entsprechend für $\alpha = -45°$

Übungsblatt — Hilfestellung 2/2

- Zufallszahlen
  - Dafür gibt es mehrere Funktionen

    random(), rand(), drand48(), ...
  - Siehe die entsprechenden man pages, z.B. man 3 rand
  - Wenn man ein Stück Code mit einer Wahrscheinlichkeit p ausführen möchte, ist drand48() am nützlichsten
  - Das liefert nämlich eine Zufallszahl zwischen 0 und 1

```
#include <stdlib.h>
...
double p = drand48();  // Random number in [0,1).
if (p < 0.1) {
  ...  // Code will be executed with probability 10%.
}
```

Literatur / Links

- Felder / Arrays
  - http://www.cplusplus.com/doc/tutorial/arrays
- Zeichenketten / Strings
  - http://www.cplusplus.com/doc/tutorial/ntcs
- Zeiger / Pointers
  - http://www.cplusplus.com/doc/tutorial/pointers
- Const
  - http://www.cplusplus.com/doc/tutorial/constants
- GNU debugger (gdb)
  - http://sourceware.org/gdb/current/onlinedocs/gdb