Programmieren in C++

SS 2018

Vorlesung 7, Dienstag 12. Juni 2018

(Eingabe / Ausgabe, Optionen, ASSERT_DEATH)

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

UNI FREIBURG

Blick über die Vorlesung heute

UNI FREIBURG

- Organisatorisches
  - Erfahrungen mit dem Ü6 — String Klasse
- Inhalt
  - Ein- und Ausgabe — fopen, fgets, stdin, stdout
  - Kommandozeilenoptionen — getopt_long
  - Testen von IO und Optionen — ASSERT_DEATH

Ü7: mit Bitcoin reich werden (zumindest in der Konsole)

Weil's beim Ü6 so schön war, sind die Tests noch mal vorgegeben

Erfahrungen mit dem Ü6 1/4

UNI FREIBURG

- Zusammenfassung / Auszüge
  - Interessante Aufgabe, gut machbar, die letzte Aufgabe (split) war anspruchsvoller als die anderen
  - Das Test-driven development fanden die meisten gut
  - Sehr gute Aufgabe, um die bisherigen Konzepte zu üben
  - "Entschuldigung für die 72 valgrind Fehler"
  - Bei einigen akkumulieren sich die Verständnisprobleme, die Programme werden ja zunehmend komplexer
  - Wunsch nach genaueren Erklärungen in V6 zum Ü6
  - Auf dem Ü5 steht "designed", es sollte "designt" heißen außerdem: geklikt, sampeln, geemailt, …

Erfahrungen mit dem Ü6 2/4

UNI FREIBURG

- Verantwortung ohne freien Willen, **Ihre** Kommentare
  - Es ist irrelevant, ob man für seine Taten verantwortlich ist oder nicht, man muss mit den Konsequenzen leben
  - Da es Gesetze und Strafen gibt, sind die Konsequenzen unserer unfreiwilligen Taten genauso vorherbestimmt
  - Man mag sich die Welt als deterministisch denken, dann sind eben die präventiv wirkenden Strafandrohungen und Moralisierungen auch vorprogrammiert, na und?
  - Studie der Uniklinik Freiburg von 2016, die die Ergebnisse von Benjamin Libet (das Unterbewusstsein entscheidet, das Bewusstsein zieht nach) zumindest relativieren
  - Eigentlich sollte ich jetzt determiniert schlafen gehen

Erfahrungen mit dem Ü6 3/4

UNI FREIBURG

- Freier Wille
  - Selbstversuch: machen Sie mal eine Zeit lang "was Sie wollen" und beobachten Sie dabei Ihren Körper, Ihre Gefühle und Ihre Gedanken

    Achtung: Sie könnten dabei erleuchtet werden 
  - Beobachtung: nehmen wir an, alles ist determiniert und wir entwickeln uns auch "nur" nach den Regeln der Physik

    Dann kann man sich auf jeden Fall mal entspannen, weil man kann es ja eh nicht ändern 

    Die meisten Menschen nehmen das Leben und Ihre Rolle darin ja tendenziell zu ernst bzw. wichtig

Erfahrungen mit dem Ü6 4/4

UNI FREIBURG

- Determinismus ≠ Vorhersagbarkeit
  - Bei sehr einfachen Systemen schon, aber bei komplexeren Systemen ist eine Simulation oft die einzige Möglichkeit herauszufinden, wie sich das System entwickeln wird

    Es gibt z.B. im Game of Life i.A. keine bessere Möglichkeit, den Zustand einer bestimmten Zelle nach X Iterationen vorherzusagen, als das Spiel bis dahin laufen zu lassen
  - Tatsächlich ist eine Theorie für den Sinn von Bewusstsein:

    Eine Simulation eines Teils der Welt (im eigenen Kopf) um möglichst gute Prognosen machen zu können

    Lebewesen, die das können, sind evolutionär eindeutig im Vorteil

Ein- und Ausgabe 1/7

UNI FREIBURG

- Beispielprogramm in C

```
#include <stdio.h>

FILE* file = fopen("doof", "r");    // Open "doof" for reading.
const int max = 1000;               // Max. length of a line.
char line[max + 1];                 // +1 for trailing null.
fgets(line, max, file);             // Read until next newline.
if (feof(file)) { ... }             // End of file reached
fclose(file);                       // Close the file.
```

  - **Achtung:** man muss beim Lesen einer Zeile eine Obergrenze für die Anzahl der Zeichen angeben, damit nicht in Speicher geschrieben wird, der eigentlich eine andere Funktion hat

    Ein "buffer overflow" war die Hauptursache hinter vielen Sicherheitslücken in Software; auch heute noch relevant

Ein- und Ausgabe 2/7

UNI FREIBURG

- Beispielprogramm in C++

```
#include <iostream>

std::ifstream file("doof");          // Open file for reading.
std::string line;                    // Max. length of a line.
std::fgets(file, line);              // Read until next newline.
if (file.eof()) { … }                // End of file reached
file.close();                        // Close the file.
```

  - Sehr ähnlich zu dem C-Code, außer dass man keine Obergrenze für die Länge einer Zeile braucht
  - Benötigt aber Templates und die STL (die C++ Standard Bibliothek), die kommen erst in V8 und V9 dran

  Von daher in der Vorlesung heute erstmal noch C-Style

Ein- und Ausgabe 3/7

UNI FREIBURG

- Die wichtigsten C-Befehle im Überblick
  - fopen öffnet eine Datei, liefert FILE* zurück
  - fclose schließt die Datei wieder
  - feof sagt, ob wir schon am Ende der Datei sind
  - fread liest eine gegeben Anzahl Bytes aus einer Datei
  - fwrite schreibt eine gegebene Anzahl Bytes in eine Datei
  - fprintf schreibt formatiert in eine Datei, analog zu printf
  - fgets liest die nächste Zeile aus einer Datei
  - Details dazu wie üblich mit man, z.B. man 3 fopen

Ein- und Ausgabe 4/7

UNI FREIBURG

- Ein paar Besonderheiten
  - Wenn fopen fehlschlägt, wird NULL zurückgegeben

    ```
    FILE* file = fopen(...);
    if (file == NULL) { perror("..."); exit(1); }
    ```

    fgets o.ä. mit file == NULL testen gibt einen seg fault
  - Das Ende der Datei wird behandelt wie ein eigenes Zeichen (EOF = end of file)
  - Wenn man das letzte "richtige" Zeichen aus einer Datei gelesen hat, ist also man noch **nicht** am Ende der Datei
  - Sondern erst nach dem nächsten Lesezugriff

Ein- und Ausgabe 5/7

UNI FREIBURG

- Benutzereingabe / Bildschirmausgabe
  - Das sind in der Unix/Linux–Welt auch "Dateien" !
  - Die "Datei" für Benutzereingabe heißt **standard input**

    ```
    fgets(line, max, stdin);  // Read line from user input.
    ```
  - Die "Datei" für Bildschirmausgabe heißt **standard output**

    ```
    fprintf(stdout, "Doof\n");  // Write to the console.
    ```
  - Außerdem gibt es noch die Fehlerausgabe **standard error**

    ```
    fprintf(stderr, "Falsch\n");  // Write to standard error
    ```

    Geht per default auf den Bildschirm, umleiten in der bash geht zum Beispiel mit ./InputOutputMain 2> error.log

Ein- und Ausgabe 6/7

UNI FREIBURG

- Testen einer Methode mit Eingabedatei
  - **Variante 1:** als Teil vom Test eine Testdatei erzeugen

    Dann am Ende vom Test wieder löschen (mit unlink)
  - **Variante 2:** Testdatei von Hand schreiben

    Integraler Bestandteil vom Test = muss mit ins SVN

  - **Achtung:** in jedem Fall soll die Testdatei klein sein

    Zur Erinnerung: Unit Tests sollen grundsätzlich nur auf kleinen Beispielen laufen und nur wenig Zeit benötigen

Ein- und Ausgabe 7/7

UNI FREIBURG

- Namenskonventionen C vs. C++
  - In C++ verwenden wir **CamelCase**

    ClassName, methodName, variableName, …

    Für Konstanten möchte cpplint.py: **k**MaxLineLength
  - In C verwenden wir **lower_case**

    c_string, max_length, …

    Zum Beispiel bei den ganzen FILE* Funktionen

    In der C++ Standard-Bibliothek (STL) wird allerdings auch lower_case verwendet … siehe Vorlesung 8

Parsen von Optionen 1/8

UNI FREIBURG

- Beispiel mit "langen" Optionennamen
  - Typischer Aufruf von der Kommandozeile

    ./InputOutputMain --head=3 --numbers example.csv
  - Zur Erinnerung: Argumente der main Funktion

    int main(int argc, char** argv);
  - Die Werte von argv sehen dann so aus:

    argc == 4

    argv[0] : ./InputOutputMain
    argv[1] : --head=3
    argv[2]: --numbers
    argv[3] : example.csv

Parsen von Optionen 2/8

UNI FREIBURG

- Beispiel mit "kurzen" Optionennamen
  - Äquivalenter Aufruf zu dem von der Folie vorher:

    ./InputOutputMain -h 3 -n example.csv

  - Argumente der main Funktion

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

  - Die Werte von argv sehen jetzt so aus: argc == 5

    argv[0] : ./InputOutputMain
    argv[1] : -h
    argv[2] : 3
    argv[3]: -n
    argv[4] : example.csv

Parsen von Optionen 3/8

UNI FREIBURG

- Verarbeitung mit getopt, Teil 1/3
  - Zuerst definiert man eine Struktur, die sagt, welche Optionen es gibt und wie sie heißen (lang und kurz)

```
#include <getopt.h>

struct option options[] = {
   { „head", 1, NULL, ‚h' },        // Option with 1 arg.
   { "numbers", 0, NULL, 'n' },     // Option with 0 args.
   { NULL, 0, NULL, 0 }             // End of array.
};
```

  - An der jeweils dritten Stelle kann man anstatt von NULL auch einen Zeiger auf eine Variable (Typ int*) übergeben

    Siehe "man 3 getopt" für die Semantik davon

Parsen von Optionen 4/8

UNI FREIBURG

- Verarbeitung mit getopt, Teil 2/3
  - Jetzt kann man die Optionen verarbeiten:

```
optind = 1;                          // Start with argv[1].
while (true) {
   // s and n = short names, the : means with argument.
   char c = getopt_long(argc, argv, "s:n", options, NULL);
   if (c == -1) break;               // No more options.
   switch (c) {                      // c is the short name.
   case 's': shift = atoi(optarg);   // Argument in optarg.
             break;
   case 'n': numbers = true;         // Option without arg.
  }
}
```

Parsen von Optionen 5/8

UNI FREIBURG

- Verarbeitung mit getopt, Teil 3/3
  - Jetzt noch die Nicht-Options Argumente

    ```
    if (optind + 1 != argc) { printUsageAndExit(); }
    fileName = argv[optind];
    ```

  - Achtung: optind zeigt nach der vorherigen Schleife auf das nächste Argument, das keine Option mehr ist
  - Das funktioniert, weil die Schleife auf der Folie vorher die Optionen und Ihre Argumente "nach vorne tauscht"

    Vorher: ./InputOutputMain –h 3 example.csv –n
    und optind == 1, so dass argv[optind] == "-h" ist

    Nachher: ./InputOutputMain –h 3 –n example.csv
    und optind == 4, so dass argv[optind] == "example.csv"

Parsen von Optionen 6/8

UNI FREIBURG

- Zwei wichtige globale Variablen aus getopt.h
  - **optind** ist der Index von dem Argument, das getopt_long als nächstes bearbeitet

    optind ist zwar mit 1 initialisiert, aber Achtung:

    beim Testen hat man die getopt_long Schleife evtl. mehrmals hintereinander, deswegen vorher immer optind = 1 setzen
  - **optarg** ist das Argument der zuletzt verarbeiteten Option, sofern sie ein Argument hatte (sonst NULL)

    optarg ist immer vom Typ char*, braucht man einen anderen Typ, muss man selber im Code konvertieren

Parsen von Optionen 7/8

UNI FREIBURG

- Testen ob ein Programm "abbricht" wenn es soll
  - Zum Beispiel, wenn es nicht mit den richtigen Optionen aufgerufen wurden

    ```
    InputOutput io;
    ASSERT_DEATH(io.parseArgs(0, NULL), "Usage: .*");
    ```

    Argument 1: Aufruf, der zum Programmabbruch führt

    Argument 2: regulärer Ausdruck für Fehlermeldung, die das Programm dann (korrekterweise) produzieren soll
  - Achtung: Funktioniert nur für Ausgabe, die nach **stderr** geschrieben werden, etwa so:

    ```
    fprintf(stderr, "Usage: ./InputOutputMain ...\n");
    ```

Parsen von Optionen 8/8

UNI FREIBURG

- Noch eine Besonderheit von ASSERT_DEATH
  - Interne Realisierung ist recht kompliziert, es wird ein sogenannter **fork** verwendet

    Der Prozess wird in zwei Prozesse aufgespalten
  - Das gibt potenziell Probleme, wenn das Programm **threads** verwendet, also Teile hat, die unabhängig voneinander nebeneinander her laufen
  - Deshalb vor der Verwendung von ASSERT_DEATH folgende Zuweisung:

    ::testing::FLAGS_gtest_death_test_style = "threadsafe";

    Sonst kommt bei Ausführung der Tests eine Warnung

Hinweise zum Testen 1/3

UNI FREIBURG

- Const cast
  - Das Parsen der Kommandozeilenparameter sollte man grundsätzlich auch testen (für das Ü7: Tests vorgegeben)
  - Für den Test Case muss man **argv** explizit setzen:

    ```
    char* argv[2] = { "arg1", "arg2" };
    ```

  - Das meckert aber der Compiler, weil die Elemente von argv vom Typ **char*** sind, aber "..." vom Typ **const char***
  - In solchen Fällen muss man dem Compiler explizit mitteilen, dass er const char* in char* umwandeln soll

    ```
    char* argv[2] = { const_cast<char*>("arg1"),
                      const_cast<char*>("args") };
    ```

    Andersrum ginge auch const_cast<const char*>

Hinweise zum Testen 2/3

UNI FREIBURG

- Vergleich von zwei Strings im Unit Test
  - Umständliche Variante:

```
ASSERT_EQ('d', result._contents[0]);
ASSERT_EQ('o', result._contents[1]);
ASSERT_EQ('o', result._contents[2]);
ASSERT_EQ('f', result._contents[3]);
ASSERT_EQ(0, result._contents[4]);
```

  - Dasselbe in einer Zeile:

```
ASSERT_STREQ("doof", result._contents);
```

  - Das geht aber (wie praktisch alle C/C++ Funktionen, die auf Zeichenketten operieren) nur mit 0-Terminierung

Hinweise zum Testen 3/3

UNI FREIBURG

- Füttern einer Datei als Standardeingabe
  - Um eine Datei als Standardeingabe für ein Programm zu nutzen, kann man Folgendes schreiben

    ./InputOutputMain < doof.txt …
  - Alternativ geht auch:

    cat doof.txt | ./InputOutputMain …
  - Um die Standardausgabe in eine Datei umzulenken:

    ./InputOutputMain > bloed.txt …
  - Um sowohl stdout und stderr umzulenken (bash):

    ./InputOutputMain **2&>1** > bloed.txt …

Literatur / Links

UNI FREIBURG

- C-style input / output
  - man 3 fopen
  - Dito für fgets, fprintf, feof, fread, fwrite, fclose, …
- Parsen von Optionen
  - man 3 getopt
- ASSERT_DEATH
  - https://github.com/google/googletest

    googletest → docs → advanced.md → "Death Tests"
- const_cast
  - http://www.cplusplus.com/doc/tutorial/typecasting