Programmieren in C++

SS 2011

Vorlesung 7, Mittwoch 22. Juni 2011

(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 6. Übungsblatt
- Themen heute
  - Ein- und Ausgabe
  - Kommandozeilenoptionen
  - ASSERT_DEATH
  - Übungsblatt: ein "grep"-ähnliches Programm
    - Gegeben eine Datei und ein "Muster", finde alle Zeilen die zu dem Muster passen
  - Ich mache anhand einer einfacheren Aufgabe (naive Verschlüsselung) alles vor, was Sie dafür brauchen

Erfahrungen mit dem Ü6 (string Klasse)

UNI FREIBURG

- Zusammenfassung / Auszüge Stand 22.6 15:50
  - Hat vielen Spaß gemacht
  - String Klasse schreiben ist eine gute Idee
  - String Klasse schreiben wieso, gibt doch schon eine
  - Einfacher und weniger Arbeit als die vorherigen Üs
  - Vorgabe / Spezifizierung in String.h war gut und hilfreich
  - ASSERT_EQ(NULL, ...) geht nicht, ASSERT_EQ(0, ...) geht
  - Lieber abstrakte(re) Probleme, design patterns, etc.
  - Ü5 (o-o machen) war Arbeitsbeschaffungsmaßnahme
  - Klarer sagen, was ist C und was ist C++? Mehr C++?
  - Wie bricht man eine void Funktion ab?

Wie bricht man eine void Funktion ab

UNI FREIBURG

- Ganz einfach mit return ohne Argument

```
void printSetOfIntegers(int* elements, int n)
{
  if (n == 0)
  {
    printf("[empty set]\n");
    return;
  }
  ...
}
```

Warum lernen wir nicht mehr C++

UNI FREIBURG

- Antwort 1:
  - Die Niveau-Unterschiede im Auditorium sind sehr groß
- Antwort 2:
  - Sie lernen hier in erster Linie gutes (objekt-orientiertes) Programmieren für die Praxis, und in zweiter Linie C++
- Antwort 3:
  - Das A und O beim guten Programmieren sind die Basics: guter Stil, gute Struktur + Namen, Dokumentation, Tests, Grundkonstrukte wie const und & und *, usw.
  - Die fortgeschrittenen Sachen machen vielleicht 20% von dem aus, was in der Praxis wichtig ist, also werden wir uns in der Vorlesung auch nur 20% damit beschäftigen

Ein- und Ausgabe 1/4

UNI FREIBURG

- Wir machen das erstmal C-style
  - Das heißt mit FILE*, fopen, fclose, fgets, fprintf, usw.
  - In C++ gibt es dafür sogenannte streams und die Operatoren << und >>
  - Die basieren aber auf templates, und die kommen erst nächste Woche dran (aus gutem Grund)
  - Aus dem selben Grund benutzen wir auch noch printf (und heute fprintf) anstatt cout
  - Außerdem sind FILE*, fopen, etc. maschinennäher (gut für's Verständnis) und effizienter
    - insbesondere basieren die C++ streams auf FILE*

Ein- und Ausgabe 2/4

UNI FREIBURG

- Unterschied Datei / Bildschirm?
  - Das ist in der Unix/Linux–Welt im Prinzip dasselbe
    - d.h. auch die Benutzer Ein- und ausgabe sind Dateien
    - auf der Kernel-Ebene heißen die alle Dateideskriptoren
      - das sind einfach ganze Zahlen, die als ID dienen
  - Die Benutzereingabe heißt standard input oder abgekürzt stdin, die Benutzerausgabe heißt standard output oder abgekürzt stdout
  - Für die Ausgabe von Fehlern gibt es noch standard error oder abgekürzt stderr; die geht standardmäßig auf den Bildschirm so wie stdout, kann aber umgeleitet werden

Ein- und Ausgabe 3/4

UNI FREIBURG

- Die wichtigsten 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 gebenene Anzahl Bytes in eine Datei
  - fprintf schreibt formatiert in eine Datei, analog zu printf
  - fgets liest die nächste Zeile aus einer Datei
  - Code dazu schreiben wir gleich zusammen
  - Ansonsten siehe die Linux **man pages**
    - man fopen, man fprintf, etc.

Ein- und Ausgabe 4/4

UNI FREIBURG

- Ein paar Besonderheiten
  - Wenn das Öffnen mit fopen fehlschlägt, wird NULL zurückgegeben
  - Unbedingt testen, sonst gibt es einen segmentation fault beim nächsten fgets, fwrite, etc.
  - Das Ende der Datei ist wie ein eigenes Zeichen
    - EOF = end of file
    - Insbesondere heißt das, dass wenn man das letzte richtige Zeichen bzw. die letzte richtige Zeile gelesen hat, gibt feof noch **nicht** true zurück, sondern erst nach dem nächsten Lesezugriff

Parsen von Optionen 1/3

UNI FREIBURG

- Oft will man in der Kommandozeile Optionen übergeben, zum Beispiel

  ./EncryptorMain --shift=3 --boldface-output MyFile.txt

  – Optionen sind auch erstmal ganz normale Argumente

  ```
  argv[0] :  ./EncryptorMain
  argv[1] :  --shift=3
  argv[2] :  --boldface-output
  argv[3] :  MyFile.txt
  ```

  – Da man das sehr oft hat, gibt es für das Parsing eine C-Bibliothek getopt, es reicht dafür am Anfang der Datei

  ```
  #include <getopt.h>
  ```

  – Siehe Codebeispiel gleich und man 3 getopt

    - Die Linux-Sachen sind **Section 3** von den man pages!

Parsen von Optionen 2/3

UNI FREIBURG

- Kurze und lange Optionen
  - Ursprünglich waren Optionen einzelne Buchstaben und man schrieb nur ein Minus davor, und bei Optionen mit Argument kein Gleichheitszeichen

    ./EncryptorMain -s 3 -b MyFile.txt
  - Lange Optionen sind besser lesbar, aber dauern auch länger zu tippen
  - Deswegen ist es üblich, beides zu unterstützen bzw. zumindestens für die häufig benutzten Optionen einen Abkürzungsbuchstaben zu haben
  - Das geht mit getopt_long, siehe Codebeispiel gleich
  - Mit getopt_long werden die Argumente von links nach rechts, eins nach dem anderen abgearbeitet

Parsen von Optionen 3/3

UNI FREIBURG

- Globale Variablen aus getopt.h
  - optind ist der Index von dem Argument, das getopt_long als nächstes bearbeit
    - optind ist mit dem Wert 1 initialisiert, aber Achtung, beim Testen hat man die getopt_long Schleife vielleicht 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 vom Typ char*
  - Achtung: getopt_long ordnet die Argumente in argv um, so dass am Ende die Argumente, die keine Optionen waren, am Ende stehen
    - und optind ist dann der Index von dem ersten davon

switch

UNI FREIBURG

- Einfach eine Abkürzung für if ... else if ... else if ...

```
void printMonth(int i)
{
  switch (i)
  {
    case 1: printf("Jan"); break;
    case 2: printf("Feb"); break;
    ...
    default: printf("Invalid month index: %d\n", i);
  }
}
```

  - **Achtung:** das break nicht vergessen, sonst werden die folgenden case Anweisung auch alle abgeprüft, das ist ineffizient und möglicherweise auch falsch

assert 1/3

UNI FREIBURG

- Einfacher Schutz vor Fehlern
  - Wenn ein Befehl unter bestimmten Bedingungen zu einem (unsanften) Programmabsturz führen kann, ist es ratsam diese Bedingung vorher abzuprüfen

```
#include <assert.h>
...
// Read the given file and so something with it.
void processFile(const char* fileName)
{
  assert(fileName != NULL);  // Should point to something.
  FILE* file = fopen(fileName, "r");
  ...
}
```

assert 2/3

UNI FREIBURG

- Was passiert bei assert(...) ?
  - Die Bedingung in Klammern wird ausgewertet
  - Falls true geht es weiter im Programm
  - Falls false bricht das Programm an der Stelle mit einer aussagekräftigen Fehlermeldung ab

    ...Main.cpp:12 ... Assertion "fileName != NULL" failed
- Im Prinzip wie ASSERT_EQ etc. bei gtest
  - Nur dass es hier nicht um Testfälle geht
  - Sondern um das Abprüfen von Vorbedingungen zur Laufzeit, und um die Fehlersuche zu erleichtern, falls die Vorbedingungen unerwarteterweise einmal nicht erfüllt ist

assert 3/3

UNI FREIBURG

- Man verwendet assert an Stellen
  - ... wo das Nicht-Gelten bestimmter Bedingungen zu ansonsten schwer zu debuggenden Folgefehlern führen können
  - ... die Gefahr besteht, dass Sie nicht gelten; Gegenbeispiel

```
for (int i = 0; i < n; i++)
{
  assert(i < n);  // Not necessary, because it's obvious.
  A[i] = B[i];
}
```

  - ... die Bedingungen eigentlich immer gelten sollten; Gegenbeispiel

```
FILE* file = fopen(fileName, "r");
assert(file != NULL);  // Better do proper error handling instead.
```

ASSERT_DEATH 1/2

UNI FREIBURG

- Manchmal will man testen ob ein Programm "abstürzt" wenn es soll
  - Z.B. mit exit(1) oder ein assert(...) schlägt fehl
  - Das macht man mit ASSERT_DEATH

    ASSERT_DEATH(encryptor.parseOptions(0, 0), "Usage:.*");
  - Das erste Argument ist der Befehl der zu einem exit (mit Wert ≠ 0) oder einem assert failure führen soll
  - Das zweite Argument ist ein regulärer Ausdruck der zu der Fehlermeldung passen muss die dann kommt
  - Wichtig: Diese Fehlermeldung muss nach stderr geschrieben werden, zum Beispiel so

    fprintf(stderr, "ERROR: digit expected at position %d\n", i);

ASSERT_DEATH 1/2

UNI FREIBURG

- Besonderheiten
  - Die interne Realisierung von ASSERT_DEATH ist recht kompliziert, es wird ein sogenannter fork verwendet
    - Prozess wird in zwei Prozesse aufgespalten
  - Das gibt potenziell Probleme, wenn das Programm threads verwendet, also Teile hat, die unabhängig voneinander nebeneinander her laufen
  - Unsere Programme benutzen keine threads, und das müssen wir gtest mitteilen, sonst gibt es eine Warnung

    ::testing::FLAGS_gtest_death_test_style = "threadsafe";

    (das ist die Mitteilung, nicht die Warnung)

Type casting, hier: const_cast

UNI FREIBURG

- Implizite Typkonvertierung
  - Manche Typenkonvertierungen finden automatisch statt

```
bool flag = true;
int x = flag;  // Will get the value 1 (true = 1, false = 0).
```

- Explizite Typkonvertierung
  - Manchmal muss man einen Typ in einen anderen, eigentlich imkompatiblen, konvertieren

```
const char* message = "Hi";
char** argv = new char*[2];
argv[1] = message;  // This will not compile.
argv[1] = const_cast<char*>(message);  // This will.
```

  - es gibt auch noch static_cast, dynamic_cast und reinterpret_cast aber die brauchen wir später erst

Literatur / Links

UNI FREIBURG

- fopen, fclose, feof, fgets, fprintf, fread, frwrite, ...
  - http://linuxmanpages.com/
  - Oder einfach man 3 fopen etc. in ein Terminal eingeben
  - Codebeispiele siehe die Encryptor.cpp aus der Vorlesung
- Parsen von Optionen
  - http://linuxmanpages.com/man3/getopt.3.php
  - Oder man 3 getopt
- ASSERT_DEATH
  - http://code.google.com/p/googletest/wiki/GoogleTestAdvancedGuide#Death_Tests
- const_cast
  - http://www.cplusplus.com/doc/tutorial/typecasting