Proseminar „C-Programmierung“
Uni HH - SS 2011

Schriftliche Ausarbeitung

Error-handling in C

Lucas Georg

29.09.2011

Inhaltsverzeichnis

1. Einführung

Diese Ausarbeitung befasst sich im Rahmen des Proseminars „C-Programmierung - Grundlagen und Konzepte” mit der Fehlerbehandlung (engl. „error-handling”) in C. Dabei wird zuerst auf die Bibliothek „errno” eingegangen. Anschließend werden die Themen „Exceptions” und „GError” behandelt.

1.1 Motivation

In jedem Programm können Fehler auftreten, die zur Terminierung des Programms führen können, wenn sie nicht behandelt werden. Wichtig ist dabei die Unterscheidung von Programmier- und Umgebungsfehlern. Programmierfehler sollten mit den angesprochenen Verfahren nicht behandelt werden, sondern bereits bei der Programmierung behoben werden, sobald sie erkannt werden. Umgebungsfehler sind Fehler, die zur Laufzeit auftreten können, auf die der Programmierer aber während der Entwicklung keinen Einfluss hat. Dazu zählen zum Beispiel ungültige Benutzereingaben oder fehlende Dateizugriffsrechte. Diese Fehler sollten zur Laufzeit behandelt werden, ohne dass das Programm terminieren muss.

Bei der Fehlerbehandlung werden die Informationen üblicherweise an andere Programmebenen weitergeleitet, um den Fehler an einer Stelle zu behandeln, an der absehbar ist, welche Auswirkungen der Fehler auf das Programm hat. Dies ist selten die Stelle, an der der Fehler aufgetreten ist. Daher sollte es bei einer guten Fehlerbehandlung möglich sein, die Fehlerinformationen weiterzuleiten.

Wie der Begriff „Exceptions” schon andeutet, sollen hauptsächlich Ausnahmefälle behandelt werden, die verhältnismäßig selten auftreten, aber eventuell große Auswirkungen auf den Programmfluss haben können. Wegen ihrer Seltenheit, ist es sinnvoll, Programmcode zur Fehlerbehandlung von der restlichen Programmlogik syntaktisch zu trennen, um den Regelfall nicht zu beeinträchtigen.

2. Errno.h

Die Benutzung der Bibliothek „errno.h” ist eine weit verbreitete Methode zur Fehlerbehandlung. Ein entscheidender Vorteil dabei ist, dass viele Funktionen aus anderen Standardbibliotheken „errno.h” benutzen, um über eventuell aufgetretene Fehler zu informieren.

2.1 Eigenschaften

In der Bibliothek „errno.h” wird die globale Variable „errno” deklariert. Sie hat den Typ int und ist in einigen Implementationen threadlokal statt global, um bei Multithreading-Unterstützung die Fehleranfälligkeit zu reduzieren. Je nach aufgetretenem Fehler wird dieser Variable ein Wert zugewiesen, der an anderer Stelle überprüft werden kann, um dann die entsprechende Fehlerbehandlung durchzuführen.

In errno.h werden außerdem Makros definiert, um die Arbeit mit der Bibliothek zu erleichtern. Im C99-Standard sind nur 3 solcher Makros vorgegeben:

Makro	Name	Bedeutung
EDOM	"Domain Error"	Bei mathematischen Funktionen wurde ein ungültiger Wert übergeben.
ERANGE	"Range Error"	Die Zahl ist zu groß bzw. zu klein für den geforderten Wertetyp.
EILSEQ	"Illegal Sequence"	Eine ungültige Zeichenkette wurde übergeben.

Die meisten Implementationen definieren allerdings noch viele weitere Makros. Welchen Wert diese Makros haben ist nicht vorgegeben. Daher sollte immer darauf geachtet werden, nur mit den Makros und nicht mit konkreten Werten zu arbeiten, um die Kompatibilität mit anderen Implementationen möglichst wenig zu beeinträchtigen. Die einzige Ausnahme ist der Wert 0, dieser steht für einen fehlerfreien Programmablauf.

Die Benutzung von „errno” ist zwar äußerst simpel, allerdings leider auch sehr fehleranfällig. Da es nur eine Variable „errno” gibt, ist es nicht möglich festzustellen, wo ein Fehler aufgetreten ist. Man erfährt lediglich, um welche Art von Fehler es sich handelt. Dies kann leicht dazu führen, dass Fehler an einer anderen Stelle vermutet werden, als sie tatsächlich aufgetreten sind und somit nicht die richtige Fehlerbehandlung durchgeführt wird.

2.2 Beispiel

```
1. int main(void)
2. {
3.      float ergebnis = 0;
4.      float x = 0;
5.      errno = 0;
6.
7.      printf("Bitte eine Zahl eingeben: ");
8.      scanf("%d", &x);
9.      ergebnis = sqrt(x);
10.
11.     if(errno == 0)
12.     {
13.             printf("Die Wurzel von %f ist %f", x, ergebnis);
14.     }
15.     else
16.     {
17.             printf("Ungültige Eingabe");
18.     }
19.     errno = 0;
20.
21.     return 0;
22. }
```

An diesem Beispiel werden einige Gefahren beim Umgang mit „errno” deutlich. „Errno” wird in Zeile 5 mit 0 belegt, es wird aber nicht überprüft, welchen Wert errno zuvor hat, somit bleibt ein eventuell zuvor aufgetretener Fehler unbehandelt. In Zeile 11 wird geprüft, ob errno gleich 0 ist. Falls nicht, wird von einer ungültigen Eingabe ausgegangen. Es ist allerdings auch möglich, dass ein Fehler in printf oder scanf aufgetreten ist, daher müsste gegebenenfalls jeder mögliche Fehler überprüft werden, der im Zusammenhang mit printf, scanf und sqrt auftreten kann, was zu einer sehr unübersichtlichen Fehlerbehandlung führen würde. Alternativ könnte nach jedem Methodenaufruf überprüft werden, ob ein Fehler aufgetreten ist. Ein weiteres Problem ist, dass die Methode trotzdem weiterläuft und auch einen Rückgabewert hat. Das hat den Nachteil, dass auch bei ungültigen Eingaben ein Rückgabewert „erfunden” werden muss. Außerdem könnte mit diesem falschen Wert in der aufrufenden Methode irrtümlich weitergearbeitet werden.

3. Exceptions

Exceptions sind in vielen Programmiersprachen eine gute Methode zur Fehlerbehandlung. Leider sind sie nicht Teil des C-Standards, daher existieren sehr viele unterschiedliche Lösungen, um in C mit Exceptions zu arbeiten. Am Beispiel von Java werde ich die Funktionsweise von Exceptions erläutern, um sie dann zu einer Implementation in C zu vergleichen. Dabei muss beachtet werden, dass Java objektorientiert ist und sich somit deutlich von C unterscheidet.

3.1 Exceptions in Java

Exceptions sind in Java eine Klasse, die Informationen über einen Fehler enthalten kann. Tritt ein Fehler auf, wird ein Exception-Objekt an die aufrufende Methode weitergereicht, damit diese den Fehler behandeln kann. Man sagt auch, die Exception wird „geworfen". Methoden, die eine Exception werfen können, müssen dies in der Signatur mit dem Schlüsselwort „throws" kennzeichnen. Wird eine solche Methode aufgerufen, muss sich die aufrufende Methode um die Fehlerbehandlung kümmern. Dies geschieht entweder, indem die Methode in einem „Try"-Block ausgeführt wird und die Fehlerbehandlung im anschließenden „Catch"-Block durchgeführt wird, oder indem die aufrufende Methode selbst das „throws" Schlüsselwort in der Signatur hat und somit die Fehlerbehandlung wiederum weiterreicht.

3.1.1 Beispiel in Java

```
double kehrwert(double x) throws Div0Exception {...}
```

...

```
1. try
2. {
3.      x = kehrwert(15);
4. }
5. catch(Div0Exception e)
6. {
7.      System.out.println("Fehler: Division durch 0");
8. }
```

Eine Methode kann beliebig viele verschiedene Exceptions werfen, allerdings nur eine pro Methodenaufruf. Wird mit dem Schlüsselwort „throw" eine Exception geworfen, wird die aktuelle Methode verlassen und liefert keinen Rückgabewert. Auch der „try"-Block wird verlassen und es wird zum „catch"-Block übergegangen ohne den Rest des „try"-Blockes auszuführen.
Die Fehlerbehandlung kann die verschiedenen Exceptions natürlich unterschiedlich behandeln, indem mehrere „catch"-Blöcke nach einem „try"-Block stehen. Durch die geschickte Benutzung von Vererbung kann eine Exception-Hierarchie so gebaut werden, dass nicht jeder Exception-Typ einen eigenen „catch"-Block benötigt. Gibt es zum Beispiel

einen „catch”-Block für den Typ „Exception”, können dort alle aufgetretenen Exceptions behandelt werden. Abschließend kann ein „finally”-Block folgen, dieser wird in jedem Fall ausgeführt, unabhängig davon, ob eine Exception geworfen wurde oder nicht.

3.2 Exceptions in C

Wie bereits erwähnt, gibt es das Konzept der Exceptions in C nicht. Allerdings möchten einige C-Entwickler nicht auf die Vorteile von Exceptions verzichten. Daher gibt es viele Versuche, das Konzept auch in C zu implementieren. Es gibt aber leider oft Einschränkungen im Vergleich zu anderen Sprachen, wie zum Beispiel Java. Dies werde ich an einer Implementation von Exceptions verdeutlichen. Zwar haben andere Implementationen andere Einschränkungen, aber es zeigt, welche Probleme dabei in C auftreten können. Die von mir behandelte Implementation ist unter http://www.nicemice.net/cexcept/ zu finden.

3.2.1 Beispiel in C

```
1. define_exception_type(int);
2. struct exception_context the_exception_context[1];
3.
4. int main(void)
5. {
6.      int e;
7.
8.      Try
9.      {
10.             kehrwert(2);
11.             kehrwert(0);
12.     }
13.     Catch(e)
14.     {
15.             printf("Fehler Nr.: %d\n", e);
16.     }
17.}
```

In Zeile 1 wird der Exception-Typ festgelegt, dies muss geschehen werden, bevor Exceptions benutzt werden. Dadurch ist es aber nicht mehr möglich, in einem „Try”-Block verschiedene Exception-Typen zu benutzen. Es wird außerdem die Variable

„the_exception_context” benötigt, damit diese Implementation funktioniert, allerdings wird man nur sehr selten selbst darauf zugreifen, hier werden Kontext-Informationen gespeichert, welche die Sprunganweisungen benötigen, die diese Lösung benutzt. Die Exception-Variable (hier „e”) muss auch zuvor deklariert werden, hier wird der Wert der Exception im Fehlerfall gespeichert. Auf keinen Fall sollte dieser Variable direkt ein Wert zugewiesen werden.

Sollen verschiedene Exceptions auch unterschiedlich behandelt werden, führt dies meist zu vielen „if”-Blöcken im „Catch”-Block. Falls, unabhängig vom aufgetretenen Fehler, immer die gleiche Fehlerbehandlung durchgeführt werden soll, kann statt „Catch(e)” auch „Catch_anonymous” verwendet werden, es wird dann keine Exception-Variable benötigt.

Mit dieser Implementation von Exceptions hat man auch in C die Möglichkeit, auf das komfortable Konstrukt der Exceptions zuzugreifen. Allerdings gibt es in C keine Klassen, die es in Java einfacher machen, mit Exceptions zu arbeiten. Damit diese Lösung funktioniert, müssen auch einige Details beachtet werden, die teils nicht einmal eine offensichtliche Relevanz haben, wie zum Beispiel die Variable „the_exception_context”. Ein großer Vorteil von Exceptions ist die Übersichtlichkeit, da die Teile des Quellcodes, die für Fehlerbehandlung zuständig sind, optisch leicht vom Rest zu unterscheiden sind. Durch die zusätzlich benötigten Anweisungen und Variablen bei dieser Lösung ist dieser Vorteil allerdings eingeschränkt.

4. GError

Eine weitere Methode zur Fehlerbehandlung ist die Benutzung von GError aus der GLib. GError ist eine Struktur, die Informationen über einen aufgetretenen Fehler enthält. Der Zeiger auf ein Exemplar von GError wird beim Methodenaufruf übergeben, welches anschließend überprüft werden muss, um festzustellen, ob ein Fehler aufgetreten ist. Üblicherweise ist der Zeiger auf dieses Exemplar NULL, wenn kein Fehler aufgetreten ist. Die Struktur GError ist folgendermaßen definiert:

```
struct GError {
        GQuark      domain;
        gint        code;
        gchar       *message;
};
```

(Quelle: http://developer.gnome.org/glib/unstable/glib-Error-Reporting.html#GError)

Das Thema GQuark soll hier nicht vertieft werden. In diesem Zusammenhang ist nur wichtig, dass mit einem GQuark Strings gespeichert werden können. Die Typen gint beziehungsweise gchar sind die bekannten Typen int und char.

In dem Feld „domain” wird der Bereich des Fehlers gespeichert, wie zum Beispiel „Fehler beim Dateizugriff”. Das Feld „code” speichert eine Zahl, die für einen spezifischen Fehler steht, zum Beispiel „Datei nicht gefunden”. Üblicherweise werden für diese Fehler Makros definiert. In der GLib sind die Makros für diese beiden Beispiele G_FILE_ERROR und G_FILE_ERROR_NOENT. In dem Feld „message” wird eine, für den Menschen lesbare, Beschreibung des Fehlers gespeichert.

4.1 Wichtige Funktionen

In der GLib gibt es einige Funktionen, die die Arbeit mit GError erleichtern.

```
GError* g_error_new(GQuark domain, gint code, const gchar *format, ...)
```

Erstellt ein neues GError-Objekt und gibt es zurück. Die Parameter entsprechen den Komponenten der GError-Struktur. Die optionalen Parameter sind für die Formatierung der Nachricht.

```
void g_set_error(GError **err, GQuark domain, gint code, const gchar *format, ...)
```

Ähnlich wie g_error_new, hat aber keinen Rückgabewert, sondern speichert in „err”, allerdings muss „*err” NULL sein. Ist „err” NULL geschieht nichts.

```
void g_error_free(GError *error)
```

Gibt „error” wieder frei. Sollte aufgerufen werden, nachdem der Fehler behandelt wurde.

```
void g_propagate_error(GError **dest, GError *src)
```

Kopiert „src” nach „*dest”. Dafür muss „*dest” NULL sein, aber „dest” darf nicht NULL sein, sonst wird nur „src” freigegeben. Diese Methode wird benutzt, wenn die aktuelle Methode den Fehler aus einer aufgerufenen an die aufrufende Methode weiterleitet.

Eine Liste von allen Methoden, die mit GError arbeiten, ist in der offiziellen Dokumentation der GLib zu finden (siehe Quelle [3]).

4.2 Beispiel

```
1. #include <glib.h>
2.
3. int main(void)
4. {
5.      GError *error = NULL;
6.      double x = kehrwert(3, &error);
7.
8.      if(error != NULL)
9.      {
10.             printf("%s", error->message);
11.             g_error_free(error);
12.     }
13.     else
14.     {
15.             printf("Der Kehrwert von 3 ist %f", x);
16.     }
17.
18.     return 0;
19. }
```

Es ist wichtig, dass das in Zeile 6 übergebene „error” NULL ist, da einige Methoden der GLib sonst eventuell unerwartete Ergebnisse liefern. In „kehrwert” ist darauf zu achten, dass statt einem GError, auch NULL übergeben werden kann. Dies könnte sonst zu Fehlern führen, wenn versucht wird, dem GError Parameter direkt etwas zuzuweisen. In diesem Fall sollte die Funktion den Fehler, wenn möglich, durch einen vorher definierten Rückgabewert melden. In Zeile 11 wird „g_error_free” aufgerufen, um sicherzustellen, dass der selbe Fehler nicht ein zweites Mal behandelt wird, falls „error” wiederverwendet werden soll.

```
1. int foo(int x, GError **err)
2. {
3.      int ergebnis = 0;
4.      GError *sub = NULL;
5.
6.      ergebnis = bar(x, &sub);
7.
8.      g_propagate_error(err, sub);
9.
10.     return ergebnis;
11. }
```

Durch die Funktionsweise von „g_propagate_error” ist eine manuelle Überprüfung von „err” und „sub” nicht nötig. Tritt ein Fehler in „foo” auf, wird „sub” nicht kopiert und ein eventueller Fehler in „bar” bleibt unbehandelt. Im Idealfall sollte dies aber nicht auftreten, da „foo” nach aufgetretenem Fehler keine weiteren Berechnungen durchführen sollte und direkt die Fehlerbehandlung an die aufrufende Funktion weiterleiten sollte.

GError bietet eine gute Möglichkeit zur Fehlerbehandlung. Sie bietet fast alle Vorzüge von Exceptions, allerdings gibt es leider keine syntaktische Trennung von der Programmlogik, sodass das Programm ohne sinnvolle Namenskonventionen und einen festgelegten Programmierstil leicht unübersichtlich werden kann.

5. Fazit

Die drei vorgestellten Verfahren zur Fehlerbehandlung in C haben Stärken und Schwächen. Die Bibliothek „errno” ist zwar leicht zu benutzen und recht übersichtlich, die größte Schwäche ist aber, dass von jedem Punkt im Programm auf die Variable „errno” zugegriffen werden kann. Dadurch ist nicht nachvollziehbar, ob der Fehler wirklich an der Stelle aufgetreten ist, wo er vermutet wird. Außerdem können Fehler von einer anderen Stelle im Programm „überschrieben” werden und bleiben dadurch unbehandelt.

Exceptions sind ein sehr komfortables Werkzeug zur Fehlerbehandlung. Leider bietet C diese Möglichkeit nicht, daher versuchen viele Entwickler, diese Funktion selbstständig „nachzurüsten”. Die „perfekte” Implementation von Exceptions in C wurde allerdings noch nicht gefunden, daher gibt es Einschränkungen, welche die Vorteile von Exceptions schmälern. Es werden Funktionen und Variablen benötigt, die syntaktisch nicht von der Programmlogik getrennt sind, wodurch die Übersichtlichkeit leidet. Es kann immer nur ein Typ zur Zeit als Exception-Variable benutzt werden, was es kompliziert macht, wenn die Methoden in einem Programm unterschiedliche Exception-Typen verwenden.

GError ist eine weitere Methode zur Fehlerbehandlung in C. Sie bietet zwar eine gute Möglichkeit, Fehlerinformationen an geeignete Stellen zur Behandlung weiterzuleiten, allerdings gibt es hier keine Trennung von der Programmlogik, sodass das Programm unübersichtlich wird und im schlimmsten Fall Verwechslungen zwischen Logik und Fehlerbehandlung auftreten können.

6. Quellen

[1] http://openbook.galileocomputing.de/c_von_a_bis_z/030_c_anhang_b_004.htm
(13.6.2011)

[2] http://de.wikipedia.org/wiki/Ausnahmebehandlung
(13.6.2011)

[3] http://developer.gnome.org/glib/unstable/glib-Error-Reporting.html
(14.6.2011)

[4] http://pubs.opengroup.org/onlinepubs/009695399/functions/errno.html
(11.6.2011)

[5] http://www.nicemice.net/cexcept/
(17.6.2011)