

2 GLib

2.1 Willkommen in der G-Welt

Der Buchstabe G symbolisiert in den Namen von GTK+, GLib, GObject, GNOME und zahlreicher anderer Pakete eine ganz eigene Welt der Softwareentwicklung.

Am Anfang einer Einführung in diese Welt muss eine Einführung in ihr Fundament, die Bibliothek GLib (*libglib-2.0*), stehen. Sie stellt die grundlegenden Datenstrukturen und Hilfsfunktionen bereit. In diesem Kapitel lernen Sie ihre Architektur und Programmierung kennen. Das Objektsystem GObject wird in Kapitel 3 gesondert besprochen.

libglib-2.0

Wenn Sie mit GTK+ und GNOME entwickeln, kommen Sie um die GLib nicht herum und sollten sie daher auch kennen und verstehen. Die Abstraktionen und Hilfen, die Ihnen GLib an die Hand gibt¹, sind jedoch auch für fast jedes andere Programmierunterfangen nützlich und vereinfachen die Portierbarkeit. Um sie zu verwenden, binden Sie den Header *glib.h* ein.

glib.h

2.1.1 Namenskonventionen

Alle Funktions- und Datentypnamen in GLib wie auch im größten Teil der restlichen G-Welt halten sich an gewisse Regeln:

Funktionsnamen sind durchgängig kleingeschrieben, und die Namens-
teile sind durch Unterstriche getrennt: `g_timer_new()`, `g_list_`
`append()`.

Typnamen komplexer Datentypen enthalten keine Unterstriche, und
jeder Namensteil beginnt mit einem Großbuchstaben: `GTimer`, `GList`.

Funktionen und Typnamen einer Bibliothek beginnen mit einem ein-
heitlichen **Präfix**; bei GLib ist dies `g_` beziehungsweise `G`.

¹Zitat eines Entwicklers: »GLib is STL for C.«

Auf einem bestimmten Datentyp arbeitende Funktionen haben als Präfix den in Funktionsnamenschreibweise – also klein und mit Unterstrichen – dargestellten Namen dieses Typs: Zu `GTimer` gehören die Funktionen auf `g_timer_`, zu `GList` die auf `g_list_`.

2.2 Einfache Typen

Die erste Umstellung, die sich beim Betreten der G-Welt für Sie ergibt, sollte der Umstieg auf die von der GLib bereitgestellten Datentypen sein (Tabelle 2-1). Dies stellt sicher, dass Typen auf allen Plattformen richtig definiert und genutzt werden, auch wenn es darum geht, Dinge wie eine vorzeichenlose Ganzzahlvariable von genau 16 Bit Breite zu deklarieren.

Die Typen `gpointer` und `gconstpointer` werden Ihnen beim Umgang mit GLib-Datenstrukturen häufig begegnen, da diese generell nur *Blindzeiger* auf Datenblöcke verwalten, um deren Typisierung Sie sich selber kümmern müssen. Auch bei der Übergabe von Daten an Callback-Funktionen, Vergleichsfunktionen beim Sortieren oder Iteratoren werden solche Blindzeiger verwandt.

Für den Typen `gboolean` definiert die GLib die beiden Konstanten `TRUE` und `FALSE`.

2.3 Routinen

Des Weiteren liefert die GLib eine Menge hilfreicher Routinen mit, die den alltäglichen Umgang mit C und dem darunter liegenden System etwas einfacher machen.

2.3.1 Speicherverwaltung

Indem Sie die Speicherverwaltung in Ihren Programmen mit den Speicherverwaltungsfunktionen der GLib (Tabelle 2-2) betreiben, können Sie sich eine Menge Kopfschmerzen ersparen.

`g_malloc(0)()`
`g_realloc()`
`g_free()`

An die Stelle von `malloc()`, `realloc()` und `free()` treten `g_malloc()`, `g_realloc()` und `g_free()`, die auf gleiche Weise funktionieren. Um in neu bereitgestelltem Speicher gleich alle Bytes auf Null zu setzen, verwenden Sie `g_malloc0()`, das im Gegensatz zum ähnlichen `calloc()` dieselbe Aufrufsyntax wie `malloc()` verwendet.

Der Vorteil dieser Funktionen gegenüber den Standardfunktionen ist die eingebaute Fehlerbehandlung; falls ein Problem bei einem Vorgang auftritt, wird mit `g_error()` (siehe Abschnitt 2.3.7) eine Fehlermel-

GLib-Datentyp	Entsprechender C-Datentyp	Wertebereich
gchar	char	
guchar	unsigned char	
gint	int	G_MININT – G_MAXINT
guint	unsigned int	0 – G_MAXUINT
gshort	short	G_MINSHORT – G_MAXSHORT
gushort	unsigned short	0 – G_MAXUSHORT
glong	long	G_MINLONG – G_MAXLONG
gulong	unsigned long	0 – G_MAXULONG
gfloat	float	-G_MAXFLOAT – G_MAXFLOAT; kleinster darstellbarer Wert >0 = G_MINFLOAT
gdouble	double	-G_MAXDOUBLE – G_MAXDOUBLE; kleinster darstellbarer Wert >0 = G_MINDOUBLE
gint8	int mit genau 8 Bit Breite	G_MININT8 – G_MAXINT8
guint8	unsigned int mit genau 8 Bit Breite	0 – G_MAXUINT8
gint16	int mit genau 16 Bit Breite	G_MININT16 – G_MAXINT16
guint16	unsigned int mit genau 16 Bit Breite	0 – G_MAXUINT16
gint32	int mit genau 32 Bit Breite	G_MININT32 – G_MAXINT32
guint32	unsigned int mit genau 32 Bit Breite	0 – G_MAXUINT32
gint64	int mit genau 64 Bit Breite	G_MININT64 – G_MAXINT64
guint64	unsigned int mit genau 64 Bit Breite	0 – G_MAXUINT64
gsize	Typ von sizeof()-Resultaten	0 – G_MAXSIZE
gssize	wie gsize, aber mit Vorzeichen	G_MINSSIZE – G_MAXSSIZE
goffset	Typ für Datei-Offsets	G_MINOFFSET – G_MAXOFFSET
gpointer	void *, Zeiger auf unbekanntem Typ	
gconstpointer	const void *, Zeiger auf unbekanntem konstantem Typ	
gboolean	Wahrheitswert	TRUE oder FALSE

Tabelle 2-1
Einfache Datentypen
der GLib

dung abgesetzt, was normalerweise zu einem Programmabbruch mit Coredump führt.

Hinweis

Wann immer es möglich ist, sollten Sie die Funktionen auf `g_slice_` in der ersten Tabellenspalte verwenden. Diese benehmen sich genauso wie herkömmliche Speicherverwaltungsfunktionen, arbeiten aber mit einem extrem effizienten Allokationsmechanismus (*slice allocator*), der auf das häufige Anlegen und Freigeben gleich großer, kleiner Blöcke ausgelegt ist. Die Verwendung ist möglich ab einer Blockgröße von $2 * \text{sizeof}(\text{void}^*)$. Einzige Einschränkung ist, dass die damit angelegten Blöcke nicht mehr in ihrer Größe geändert werden können und beim Freigeben noch einmal die Blockgröße angegeben werden muss.

Slice-Funktion	Konventionelle Funktion	Entsprechende C-Funktion
<code>gpointer g_slice_alloc(gsize n_bytes)</code>	<code>gpointer g_malloc(gsize n_bytes)</code>	<code>void *malloc(size_t size)</code> mit Fehlerbehandlung
<code>gpointer g_slice_alloc0(gsize n_bytes)</code>	<code>gpointer g_malloc0(gsize n_bytes)</code>	wie <code>malloc()</code> , initialisiert Speicher wie <code>calloc()</code>
<code>-/-</code>	<code>gpointer g_try_malloc(gsize n_bytes)</code>	wie <code>malloc()</code> ohne Fehlerbehandlung
<code>-/-</code>	<code>gpointer g_realloc(gpointer mem, gsize n_bytes)</code>	wie <code>void *realloc(void *ptr, size_t size)</code> mit Fehlerbehandlung
<code>-/-</code>	<code>gpointer g_try_realloc(gpointer mem, gsize n_bytes)</code>	wie <code>realloc()</code> ohne Fehlerbehandlung
<code>g_slice_free(gsize n_bytes, gpointer mem)</code>	<code>void g_free(gpointer mem)</code>	<code>void free(void *ptr)</code>

Tabelle 2-2

Speicherverwaltung
mit GLib

Üblicherweise wird die Speicherblockgröße beim Aufruf von Funktionen wie `malloc()` oder `g_malloc()` nicht numerisch angegeben, sondern als ganzzahliges Vielfaches einer Strukturgröße mit dem `sizeof()`-Operator ermittelt; der zurückgegebene Zeiger wird meistens gleich in einen passenden Zeigertypen zurechtgecastet. In der GLib gibt es daher Makros, um dies abzukürzen:

```

typedef struct _footyp footyp;
footyp* dreimalfoo;

/* Platz für drei footyp-Strukturen allozieren -- lang */
dreimalfoo = (footyp*)g_malloc(sizeof(footyp) * 3);

/* dasselbe in kurz */
dreimalfoo = g_new(footyp, 3);

/* Nullinitialisieren geht auch */
dreimalfoo = g_new0(footyp, 3);

/* Block erweitern -- Platz für eine Struktur mehr */
dreimalfoo = (footyp*)g_realloc(dreimalfoo, sizeof(footyp)*4);

/* und wieder dasselbe in kurz */
dreimalfoo = g_renew(footyp, dreimalfoo, 4);

```

Die Funktionsweise dürfte aus dem Listing recht klar hervorgehen. `g_new()` ist hier das Pendant zu `g_malloc()`, `g_new0()` entspricht `g_malloc0()`, und `g_renew()` ist eine Kurzschreibweise für `g_realloc()`. Auch hier existieren Slice-Pendants, `g_slice_new()` und `g_slice_new0()`. Ein zusätzliches Makro, `g_slice_free()`, funktioniert wie `g_slice_free1()`, erwartet aber als erstes Argument einen Typen und keine Blockgröße.

Anders als bei `sizeof()` muss bei diesen Funktionen zur Größenbestimmung des Speicherblocks immer ein Typ angegeben werden. Der Ausdruck `b = g_new(a, 1)` kompiliert nicht, wenn `a` eine Variable ist.

Das von den Standardfunktionen `malloc()` und `realloc()` her bekannte Verhalten, bei einem Fehlschlag ohne weitere Fehlermeldung `NULL` zurückzugeben, bieten die ansonsten äquivalenten Funktionen `g_try_malloc()`, `g_try_realloc()`, `g_try_new()`, `g_try_new0()` und `g_try_renew()`.

Speicher, den ein Programm zu seinem Funktionieren nicht unbedingt benötigt, können Sie theoretisch also mit den `*_try_*`-Funktionen verwalten und Speicherfehler selber abfangen. Dies ist normalerweise nur in Situationen sinnvoll, wenn extreme Mengen Speicher angefragt werden und es, sofern dieser nicht verfügbar ist, möglich ist, die Anwendung ohne diesen weiterlaufen zu lassen. Ein Beispiel wäre das Laden sehr großer Grafiken – schlägt dies fehl, kann man die Grafik durch einen Platzhalter ersetzen und trotzdem weitermachen.

Listing 2.1

Abkürzungen für gebräuchliche Speicherwaltungsakte

```

g_(slice_)new(0)()
g_renew()
g_slice_free()

```

Warnung

```
g_try_*()
```

Hinweis

2.3.2 Quarks

Wenn die Notwendigkeit besteht, etwas in einem Programm zu kennzeichnen (beispielsweise eine Datenstruktur), gibt es dafür prinzipiell zwei Möglichkeiten: eine numerische Kennung oder eine Zeichenkette als Bezeichner.

Die numerische Kennung hat das Problem, nicht besonders verständlich zu sein. Dafür sind Zahlen sehr schnell miteinander zu vergleichen. Falls Ihnen im Voraus bekannt ist, wie viele Kennungen Sie brauchen, können Sie einen Aufzählungstypen definieren und somit Symbole für die benötigten Kennziffern festschreiben. Nur lässt sich so eine Aufzählung zur Laufzeit nicht erweitern. Eine Zeichenkette hingegen ist ein leicht verständlicher Bezeichner, doch dauert das Vergleichen zweier Zeichenketten um Größenordnungen länger als das zweier Zahlen.

Die GLib bietet etwas an, was die Vorteile von numerischen Kennungen mit denen von Zeichenketten verbindet: die sogenannten Quarks. Der hierfür definierte Datentyp `GQuark` ist nichts anderes als eine Ganzzahl. Die für `GQuarks` definierten Funktionen ordnen jeder Zeichenkette eine positive Zahl zu, und zwar gleichen Zeichenketten stets die gleiche Zahl: Zu jedem `GQuark` gehört eine Zeichenkette; definierte `GQuarks` sind immer größer 0.

Das `GQuark` zu einer Zeichenkette wird zurückgegeben durch die Funktion `g_quark_from_string()`, die eine Zeichenkette als einziges Argument entgegennimmt. Dasselbe macht `g_quark_from_static_string()`, allerdings ohne eine Kopie der übergebenen Zeichenkette anzulegen.

Warnung

Dies spart Speicher, erfordert aber, dass die Zeichenkette während der gesamten Laufzeit existiert, also eine Konstante übergeben wurde.

Falls Sie das `GQuark` zu einer Zeichenkette ermitteln beziehungsweise prüfen wollen, ob es zu einer Zeichenkette überhaupt ein `GQuark` gibt, kommt `g_quark_from_string()` natürlich nicht in Frage, da dieses im Falle der Nichtexistenz einfach ein neues Quark anlegt. Darum die Funktion `g_quark_try_string()`; sie funktioniert wie `g_quark_from_string()`, legt allerdings, wenn es zu der übergebenen Zeichenkette noch kein Quark gibt, keines an, sondern gibt 0 zurück.

Im Gegenzug liefert `g_quark_to_string()` mit einem `GQuark` als einzigem Argument die zugehörige Zeichenkette zurück (die Zeichenkette selbst, keine Kopie!).

Listing 2.2

`GQuark`

```
GQuark quark = 0;
quark = g_quark_from_string("Topfen");
if (!g_quark_try_string("Hüttenkäse"))
```

```
g_print("Zu »Hüttenkäse« existiert noch kein Quark\n");
g_print("quark entspricht der Zeichenkette »%s«\n",
      g_quark_to_string(quark));
```

GQuarks sind Zeichenketten zugeordnet, aber Zahlen und daher effizient zu vergleichen. Beachten Sie jedoch, dass GQuarks keine Sortierschlüssel darstellen: Sie verschlüsseln keine Information darüber, wo ihre zugeordneten Zeichenketten in der Sortierordnung stehen.

Hinweis

2.3.3 C-Zeichenketten

Zu den Hilfsfunktionen im Angebot der GLib gehört auch eine Reihe von Werkzeugen zur Bearbeitung von C-Zeichenketten (nicht zu verwechseln mit den Funktionen zum GLib-eigenen Zeichenkettentyp GString, der in Abschnitt 2.4.1 besprochen wird). Diese Funktionen stellen sich in eine Reihe mit den C-Standardfunktionen zur Zeichenkettenbearbeitung wie `sprintf()`, `strdup()` oder `strstr()`.

Zunächst seien jene Funktionen genannt, die eine neu angelegte Zeichenkette zurückgeben, die Sie später selber freigeben müssen:

- `gchar* g_strdup(const gchar *zkette)`** Legt eine neue Kopie von *zkette* an und gibt sie zurück *g_strdup()*
- `gchar* g_strdup(zkette, gsize laenge)`** Dito, kopiert aber nur die ersten *laenge* Zeichen von *zkette*. Die zurückgegebene Zeichenkette ist immer *laenge*+1 Zeichen lang und mit einem NUL-Zeichen abgeschlossen; ist *zkette* kürzer als *laenge* Zeichen, wird der Rückgabewert dazu gegebenenfalls mit NULs aufgefüllt.
- `gchar* g_strnfill(laenge, gchar zeichen)`** Legt eine neue Zeichenkette der Länge *laenge* an und füllt sie mit dem Zeichen *zeichen* aus *g_strnfill()*
- `gchar* g_strdup_printf(const gchar *format, ...)`** Akzeptiert ein Format und eine Werteliste wie `printf()`, legt eine neue Zeichenkette mit passender Länge an und schreibt die Ausgabe hinein. Dies ist die sicherere und komfortablere Alternative zu `sprintf()`, und diese Funktion wird Ihnen in diesem Buch noch häufiger begegnen. *g_strdup_*
(v)printf()
- `gchar* g_strdup_vprintf(format, va_list liste)`** Dito, nimmt aber eine *va_list* entgegen, zum Beispiel zur Verwendung in selbstgemachten Funktionen mit variabler Parameterzahl
- `gchar* g_strescape(zkette, const gchar *ausnahmen)`** Diese Funktion gibt eine neu angelegte Kopie von *zkette* zurück, in der `'\b'`, `'\f'`, `'\n'`, `'\r'`, `'\t'`, `'\'` und `'\"` sowie alle ASCII-Steuerzeichen und Nicht-ASCII-Zeichen durch die entsprechende C-Darstellung (Backslash-Sequenzen) umschrieben sind. Falls *ausnahmen* irgend- *g_strescape()*

welche Zeichen enthält, werden diese vom Umschreiben ausgenommen.

- g_strcompress()* **gchar* g_strcompress(*zkette*)** Die Umkehrfunktion: Gibt eine neu angelegte Kopie von *zkette* zurück, in der C-Backslash-Sequenzen in Sonderzeichen umgewandelt sind
- g_strconcat()* **gchar* g_strconcat(*zkette*, ...)** Verkettet eine mit NULL abgeschlossene Liste von beliebig vielen C-Zeichenketten miteinander zu einer neu angelegten Zeichenkette und gibt diese zurück
- g_strjoin()* **gchar* g_strjoin(const gchar **trenner*, *zkette*, ...)** Dito, nur fügt diese Funktion die als erstes Argument übergebene Zeichenkette als Feldtrenner zwischen den Teilzeichenketten ein. Der Effekt von *g_strjoin*("", *arg1*, *arg2*, *arg3*, NULL) ist derselbe wie der von *g_strconcat*(*arg1*, *arg2*, *arg3*, NULL).

Im Gegensatz dazu arbeiten folgende Funktionen auf bereits existierenden Zeichenketten. Wie manche C-Standardfunktionen geben einige von ihnen die Zeichenkette, auf der sie gearbeitet haben (keine Kopie!), zusätzlich noch zurück.

- g_stpcpy()* **gchar* g_stpcpy(gchar **ziel*, const char **quelle*)** Kopiert die Zeichenkette *quelle* samt dem abschließenden NUL-Zeichen in den Puffer *ziel* und gibt einen Zeiger auf die Stelle zurück, an der dieses NUL-Zeichen im Puffer steht. Diese Funktion kann nützlich sein, um Zeichenketten zu verketteten.
- g_strstr_len()* **gchar* g_strstr_len(const gchar **heuhausen*, *laenge*, const gchar **nadel*)** Durchsucht die ersten *laenge* Zeichen der Zeichenkette *heuhausen* nach der Zeichenkette *nadel* und gibt einen Zeiger auf den Beginn deren ersten Auftretens zurück. Wurde *nadel* nicht gefunden, ist das Ergebnis NULL.
- g_strrstr(_len)()* **gchar* g_strrstr(*heuhausen*, *nadel*)** Durchsucht *heuhausen* nach der Zeichenkette *nadel* und gibt einen Zeiger auf deren *letztes* Auftreten darin zurück. Auch hier wird NULL zurückgegeben, falls kein Treffer vorkam.
- gchar* g_strrstr_len(*heuhausen*, *laenge*, *nadel*)** Dito, durchsucht aber nur die ersten *laenge* Zeichen von *heuhausen*
- g_(v)snprintf()* **gint g_snprintf(*zkette*, gulong *n*, *format*, ...)** Wie *sprintf()* schreibt diese Funktion eine durch *format* formatierte, sich daran anschließende Werteliste in den Puffer *zkette*. Im Gegensatz zu *sprintf()* schreibt sie jedoch maximal *n* Zeichen (das abschließende NUL eingeschlossen) hinein.

Der Rückgabewert von `g_snprintf()` ist standardkonform, also die Anzahl Zeichen, die die Funktion ausgabe, wäre der Puffer groß genug. Dies entspricht nicht dem Verhalten traditioneller `sprintf()`-Implementationen.

Hinweis

- gint g_vsnprintf(*zkette*, *n*, *format*, *liste*)** Dito mit einer `va_list`
- gchar* g_strreverse(*zkette*)** Kehrt die Zeichenkette *zkette* um `g_strreverse()`
- gchar* g_strchug(*zkette*)** Trennt Leerraum am Anfang von *zkette* ab `g_strchug()`
- gchar* g_strchomp(*zkette*)** Trennt Leerraum am Ende von *zkette* ab `g_strchomp()`
- gchar* g_strstrip(*zkette*)** Trennt Leerraum am Anfang und am Ende von *zkette* ab `g_strstrip()`
- gchar* g_strdelimit(*zkette*, *trenner*, *gchar neuer_trenner*)** Ersetzt alle in *trenner* vorkommenden Zeichen in *zkette* durch das Zeichen *neuer_trenner*. Ist *trenner* NULL, wird eine Standardzeichenkette (`G_STR_DELIMITERS = "_-|> <."`) verwendet. `g_strdelimit()`
- gchar* g_strcanon(*zkette*, *const gchar *gueltige_zeichen*, *gchar ersatz*)** Ersetzt alle nicht in *gueltige_zeichen* vorkommenden Zeichen in *zkette* durch *ersatz* `g_strcanon()`
- Die folgenden Funktionen verändern keine Zeichenketten, sind aber trotzdem bei der Arbeit damit nützlich:
- gsize g_printf_string_upper_bound(*const gchar *format*, *liste*)** Gibt die Länge (in Zeichen) der Ausgabe eines `sprintf()`-Aufrufs mit dem Format *format* und der in der `va_list` *liste* übergebenen Parameterliste zurück `g_printf_string_upper_bound()`
- gdouble g_ascii_strtod(*const gchar *zkette*, *gchar **ende*)** Interpretiert *zkette* als eine Gleitkommazahl, gibt deren Wert zurück und trägt, sofern *ende* nicht NULL ist, darin einen Zeiger auf das Zeichen hinter dem letzten bei der Umwandlung ausgelesenen ein. Der Unterschied zur Standard-Bibliotheksfunktion `strtod()` ist, dass die Locale ignoriert und immer eine einheitlich formatierte Zahl im ASCII-Zeichensatz erwartet wird. `g_ascii_strtod()`
- gchar* g_ascii_dtostr(*gchar *zkette*, *gint laenge*, *gdouble zahl*)** Formatiert *zahl* als Zeichenkette und trägt sie in der *laenge* Byte langen *zkette* ein. Die Locale wird dabei ignoriert. Die Länge der erzeugten Zeichenkette in Bytes ist niemals größer als die Konstante `G_ASCII_DTOSTR_BUF_SIZE`; benutzen Sie diese Konstante daher gegebenenfalls, um die Zeichenkette zu allozieren. `g_ascii_dtostr()`
- `G_ASCII_DTOSTR_BUF_SIZE`

Hinweis

Verwenden Sie `g_ascii_strtod()` und `g_ascii_dtostr()`, um Werte in Dateien zu lesen und zu schreiben, die nicht für Endbenutzer gedacht sind. Da ein einheitliches, Locale-unabhängiges Zahlenformat benutzt wird, sind Sie vor Fehlern geschützt, die zum Beispiel durch den lokalen Gebrauch von Dezimalkomma beziehungsweise -punkt auftreten könnten.

`g_str_has_prefix/suffix()`

gboolean g_str_has_prefix(const gchar *z_kette, const gchar *prae-fix) Gibt TRUE zurück, wenn *z_kette* mit *prae-fix* beginnt, sonst FALSE.

gboolean g_str_has_suffix(z_kette, const gchar *suffix) Gibt TRUE zurück, wenn *z_kette* mit *suffix* endet, sonst FALSE.

`GStrV`

Zum Schluss noch einige Funktionen, die mit Feldern von Zeichenketten (`GStrV`, entspricht `gchar**`) arbeiten:

`g_strsplit(_set)()`

gchar g_strsplit(z_kette, const gchar *trenner, gint max_felder)**

Trennt maximal *max_felder* durch *trenner* voneinander getrennte Felder aus *z_kette* heraus und gibt sie als ein Feld von Zeichenketten zurück. Hat die Eingabezeichenkette mehr als *max_felder* Felder, wird ihr Rest an das letzte Feld angehängt. Ist *max_felder* kleiner oder gleich 0, wird die Anzahl der Ausgabefelder nicht begrenzt. Ist die Eingabezeichenkette leer, wird ein leeres Feld zurückgegeben.

gchar g_strsplit_set(z_kette, trenner, gint max_felder)** Dito, betrachtet *trenner* jedoch nicht als eine Trennzeichenkette, sondern als eine Menge möglicher ASCII-Trennzeichen, so dass hinterher (abgesehen von eventuellen Überschüssen, die im letzten Feld landen) keines der resultierenden Felder irgendeines der Zeichen in *trenner* enthält.

`g_strjoinv()`

gchar* g_strjoinv(trenner, gchar **feld) Fügt die Zeichenketten im Feld *feld* zu einer neu angelegten Zeichenkette zusammen und gibt sie zurück. Ist *trenner* nicht NULL, wird sein Inhalt als Feldtrenner zwischen die einzelnen Teilzeichenketten gestellt.

`g_strdupv()`

gchar g_strdupv(feld)** Gibt eine neu angelegte Kopie des Zeichenkettenfeldes *feld* zurück.

`g_strfreev()`

void g_strfreev(feld) Gibt die Zeichenketten in *feld* und das Feld selber frei.

Warnung

Verwenden Sie keine andere Funktion, um ein von `g_strsplit()` oder `g_strdupv()` zurückgegebenes Feld freizugeben.

g_uni char_...	Art	G_UNICODE_...
validate()	gültig	
isdefined()	im Unicode-Standard definiert nicht zugewiesen/intern reserviert	UNASSIGNED/Private_USE
isalnum()	Buchstabe oder Ziffer	
isalpha()	Buchstabe	
islower()	kleingeschrieben	LOWERCASE_LETTER
isupper()	großgeschrieben	UPPERCASE_LETTER
istitle()	Titelbuchstabe (siehe Anmerkung)	TITLECASE_LETTER
	Modifikatorbuchstabe/sonstiger Buchstabe	MODIFIER/OTHER_LETTER
isdigit()	Dezimalziffer	DECIMAL_NUMBER
isxdigit()	Hexadezimalziffer	
	Ziffer aus Buchstaben/andere Ziffer	LETTER/OTHER_NUMBER
ispunct()	Satzzeichen oder Symbol	
	verbindendes/strichartiges Satzzeichen	CONNECT/DASH_PUNCTUATION
	öffnendes/schließendes Satzzeichen	OPEN/CLOSE_PUNCTUATION
	einleitendes/abschließendes Satzzeichen	INITIAL/FINAL_PUNCTUATION
	sonstiges Satzzeichen	OTHER_PUNCTUATION
	Währungssymbol	CURRENCY_SYMBOL
	mathematisches Zeichen	MATH_SYMBOL
	Modifikatorsymbol/sonstiges Symbol	MODIFIER/OTHER_SYMBOL
ismark()	Marke (bspw. kombinierendes diakritisches Zeichen)	
	kann zu einem Basisbuchstaben hinzukommen	COMBINING_MARK
	kann einen Basisbuchstaben umschließen	ENCLOSING_MARK
	Surrogat	SURROGATE
isspace()	Leerraum (Leerzeichen, Tabulator, etc.)	
	Zeilentrenner/Absatztrenner	LINE/PARAGRAPH_SEPARATOR
	Leerzeichen	SPACE_SEPARATOR
iswide()	doppelt breites Zeichen	
iscntrl()	Steuerzeichen	CONTROL
	Formatzeichen	FORMAT
isgraph()	grafisch (weder unsichtbar noch Leerraum)	
isprint()	druckbar (grafisch oder Leerraum)	

Tabelle 2-3
Funktionen und
Konstanten zur
Kategorisierung von
Unicode-Zeichen

2.3.4 Unicode und Zeichencodierungen

Die klassischen C-Zeichenkettenfunktionen und auch die im vorigen Abschnitt vorgestellten der GLib befassen sich mit etwas, was man ebenso gut ›Bytekette‹ wie Zeichenkette nennen könnte. Wie viele Bytes ein Zeichen hat, ist für die Funktionen, die nicht mit einzelnen Zeichen vom Typ `gchar` arbeiten, mehr oder weniger egal.

Anders dagegen die in diesem Abschnitt besprochenen Funktionen. Sie arbeiten mit *Unicode*-Zeichen und Unicode-Zeichenketten. Unicode ist auf drei verschiedene Arten und Weisen codierbar:

UCS-4 Eine mit dem 32-Bit-Zeichensatz UCS kompatible Codierung: Jedes Zeichen ist vier Bytes breit, das Unicode-Zeichen nimmt die beiden niederwertigen Bytes ein, die anderen beiden sind null.

gunichar(2)

Der GLib-Datentyp für Zeichen in UCS-4-Codierung ist `gunichar`. Er ist 32 Bit breit und der Standardtyp für Unicode-Zeichen. UCS-4-codierte Zeichenketten (`gunichar*`) werden von einigen Funktionen verwendet.

UTF-16 Die native Codierung: Jedes Zeichen ist zwei Bytes breit.

Die GLib kennt den Datentyp `gunichar2` für UTF-16-codierte Unicode-Zeichen. Er ist nur 16 Bit breit. Auch UTF-16-Zeichenketten (`gunichar2*`) werden an einigen Stellen verwendet.

UTF-8 Diese Codierung ist die in der Praxis wohl wichtigste, da sie kompatibel zu ASCII ist: ASCII-Zeichen werden ein Byte breit dargestellt, andere Zeichen zwei oder mehr Bytes breit. Jeder ASCII-Text ist gleichzeitig ein gültiger UTF-8-Text. Allerdings besteht der Nachteil, dass ein UTF-8-Text keine Kette von gleich breiten Zeichen ist und deswegen durch zeichenweises Iterieren zerlegt werden muss; einfach mitten hineinzugreifen, ist nicht möglich.

UTF-8-codierter Text kann in normalen C-Zeichenketten (`gchar*`) abgelegt werden. Einzelne UTF-8-Zeichen sind nie breiter als 32 Bit und passen daher, wie UCS-4-Zeichen auch, in `gunichar`.

Eine ganze Reihe von Funktionen überprüfen ein einzelnes Unicode-Zeichen (`gunichar`) auf verschiedene Eigenschaften. Die meisten von ihnen beginnen auf `gunichar_is` und Sie finden Sie in Tabelle 2-3; sie nehmen ein einzelnes Unicode-Zeichen als Parameter und geben einen Wahrheitswert zurück. Ähnlich, nur noch detaillierter, arbeitet die Funktion `gunichar_type()`. Sie liefert als Typ des Zeichens eine Reihe von Konstanten auf `G_UNICODE_` zurück, die Sie in derselben Tabelle finden.²

gunichar_is()*
gunichar_
validate()
gunichar_type()

²Anmerkung zur Tabelle: Titelbuchstaben sind Großbuchstaben, die am Anfang eines Wortes stehen, dessen Rest in Kleinbuchstaben geschrieben ist. Das in der Unicode-Dokumentation angegebene Beispiel ist die Ligatur von D und

Noch einige weitere Funktionen beschäftigen sich mit `gunichars`:

- gunichar g_unichar_toupper(gunichar z)** Gibt das Unicode-Zeichen `z` als Großbuchstaben zurück, sofern möglich; andernfalls wird das Zeichen unverändert zurückgegeben. *g_unichar_to_lower/upper/title()*
- gunichar g_unichar_tolower(z)** Dito, wandelt aber in Kleinbuchstaben um.
- gunichar g_unichar_totitle(z)** Dito, wandelt aber in Titelbuchstaben (siehe Fußnote S. 18) um.
- gint g_unichar_digit_value(z)** Gibt den numerischen Wert der Dezimalziffer `z` zurück. Ist `z` keine Dezimalziffer, wird `-1` zurückgegeben. *g_unichar_(x)digit_value()*
- gint g_unichar_xdigit_value(z)** Dito für Hexadezimalziffern.

Nun kennen Sie bereits viele schöne Dinge, die Sie mit `gunichars` tun können. Zu beantworten bleibt die Frage, wo Sie überhaupt solche Zeichen herbekommen. Meistens werden Sie Unicode-Zeichen aus einer UTF-8-codierten Zeichenkette entnehmen wollen, und eben damit, solche Zeichenketten auf Gültigkeit zu überprüfen, darüber zu iterieren und Zeichen daraus auszulesen, befassen sich die nächsten Funktionen.

Hierbei gilt allgemein, dass die übergebene Zeichenkette in einem NUL-Zeichen enden kann, aber nicht muss. Nimmt eine Funktion einen Längenparameter vom Typ `gssize` entgegen, kann darin die Anzahl der auszuwertenden Bytes übergeben werden. Soll eine NUL-terminierte Zeichenkette komplett ausgelesen werden, übergeben Sie `-1` statt der Länge.

Hinweis

- gboolean g_utf8_validate(const gchar *z_kette, gssize laenge, const gchar **ende)** Prüft die UTF-8-Zeichenkette `z_kette` auf Gültigkeit. Ist sie vollständig gültig, wird `TRUE` zurückgegeben. Andernfalls ist das Ergebnis `FALSE`. Falls `ende` nicht `NULL` ist, wird das Ende des gültigen Teils von `z_kette` dort hineingeschrieben, das heißt, wenn ein Fehler auftrat, zeigt `ende` nach dem Aufruf auf das erste ungültige Zeichen, andernfalls auf das Ende der Zeichenkette. *g_utf8_validate()*
- gunichar g_utf8_get_char_validated(const gchar *z, gssize laenge)** Diese Funktion versucht, aus der Bytesequenz, auf die `z` zeigt, ein UTF-8-codiertes Unicode-Zeichen auszulesen, und überprüft diese Sequenz vorher in jeder Hinsicht auf Gültigkeit als UTF-8. Konnte ein gültiges Zeichen ermittelt werden, wird es zurückgegeben. Zeigt `z` auf den Anfang eines unvollständigen Zeichens am Ende *g_utf8_get_char_validated()*

Z. Als Kleinbuchstabe sieht sie aus wie »dz«, als Großbuchstabe wie »DZ«, als Titelbuchstabe jedoch wie »Dz«.

einer Zeichenkette, wird `(gunichar)-2` zurückgegeben; ist die Bytesequenz völlig ungültig, `(gunichar)-1`.

`g_utf8_get_char()` **gunichar g_utf8_get_char(z)** Wandelt die UTF-8-Bytesequenz, auf die `z` zeigt, in ein Unicode-Zeichen und gibt es zurück.

Warnung

Es finden keinerlei Überprüfungen statt. Also *muss* `z` auf ein gültiges UTF-8-codiertes Zeichen zeigen. Verwenden Sie diese Funktion in Zeichenketten, die Sie vorher mit `g_utf8_validate()` geprüft haben; das ist schneller, als jedes einzelne Zeichen mit `g_utf8_get_char_validated()` auszulesen.

`g_utf8_next/prev_char()` **gchar* g_utf8_next_char(z)** Gibt einen Zeiger auf den Anfang des nächsten UTF-8-Zeichens hinter `z` in der Zeichenkette, in die `z` hinzeigt, zurück. `z = g_utf8_next_char(z)`; schiebt den Zeiger also um ein Zeichen in der Zeichenkette weiter. Der Zeiger darf noch nicht am Ende der Zeichenkette stehen.

Hinweis

Es finden keine Überprüfungen statt; die Zeichenkette muss also gültig sein. Dies gilt auch für alle folgenden Funktionen, die auf UTF-8-Zeichenketten arbeiten.

gchar* g_utf8_prev_char(z) Dito, nur rückwärts. Der Zeiger darf hierbei nicht bereits am Anfang der Zeichenkette stehen.

`g_utf8_find_next/prev_char()` **gchar* g_utf8_find_next_char(z, const gchar *ende)** Wie `g_utf8_next_char()`, wobei `ende` auf das Ende der Zeichenkette zeigt. Ist `ende` `NULL`, wird davon ausgegangen, dass die Zeichenkette mit `NUL` abschließt. Die Funktion gibt `NULL` zurück, wenn `z` bereits am Ende der Zeichenkette steht.

gchar* g_utf8_find_prev_char(zkette, z) Dito rückwärts. Die Funktion gibt `NULL` zurück, wenn `z` bereits am Anfang von `zkette` steht.

`g_utf8_pointer_to_offset()` **glong g_utf8_pointer_to_offset(zkette, const gchar *pos)** Gibt den Offset (die Zeichennummer) des UTF-8-Zeichens innerhalb der Zeichenkette `zkette` an, auf das `pos` zeigt.

`g_utf8_offset_to_pointer()` **gchar* g_utf8_offset_to_pointer(zkette, glong offset)** Gibt einen Zeiger auf das Zeichen mit dem Offset `offset` der Zeichenkette `zkette` zurück.

Einige weitere Funktionen befassen sich noch mit ganz normalen Zeichenkettenoperationen auf UTF-8-Zeichenketten, sofern die klassischen C->Byteketten-<-Funktionen hierfür nicht ausreichen.

`g_utf8_strlen()` **glong g_utf8_strlen(zkette, gssize laenge)** Gibt die Länge der UTF-8-Zeichenkette `zkette` in UTF-8-Zeichen zurück.

`g_utf8_strncpy()` **gchar* g_utf8_strncpy(gchar *ziel, const gchar *quelle, gsize anzahl)** Kopiert wie `strncpy()` einen Teil der Zeichenkette `quelle`

nach *ziel*, zählt aber nicht Bytes, sondern *anzahl* UTF-8-Zeichen. Gibt *ziel* zurück.

gchar* g_utf8_strreverse(*zkette*, *laenge*) Gibt eine neu angelegte UTF-8-Zeichenkette zurück, die die Umkehrung von *zkette* enthält.

g_utf8_strreverse()

Denken Sie daran, die zurückgegebene Zeichenkette anders als bei *g_utf8_strreverse()* freizugeben. Beachten Sie außerdem, dass das Ergebnis der Funktion nicht zur Anzeige gedacht ist, da zum Beispiel als Sequenz dargestellte zusammengesetzte Zeichen nicht in der richtigen Reihenfolge erhalten bleiben.

Warnung

gchar* g_utf8_strchr(*zkette*, *laenge*, *gunichar z*) Gibt einen Zeiger auf das erste Vorkommen des UTF-8-Zeichens *z* in der Zeichenkette *zkette* zurück, oder NULL, falls es nicht gefunden wurde.

g_utf8_str(r)chr()

gchar* g_utf8_strrchr(*zkette*, *laenge*, *gunichar z*) Dito, sucht aber von hinten und findet so das letzte Vorkommen des Zeichens.

gchar* g_utf8_strup(*zkette*, *laenge*) Legt eine neue, in Großbuchstaben gewandelte Kopie von *zkette* an und gibt sie zurück. Da zum Beispiel das deutsche ‘ß’ dabei in ›SS‹ umgewandelt wird, kann die Länge sich dabei ändern.

g_utf8_strup/down()

gchar* g_utf8_strdown(*zkette*, *laenge*) Dito, wandelt aber in Kleinbuchstaben um.

gchar* g_utf8_casefold(*zkette*, *laenge*) Gibt eine neu angelegte Kopie von *zkette* zurück, die in eine von Groß- oder Kleinschreibung unabhängige Darstellung umgewandelt wurde. Diese ist zur Ausgabe ungeeignet, kann aber zum Vergleichen oder Ordnen verwendet werden.

g_utf8_casefold()

gchar* g_utf8_normalize(*zkette*, *laenge*, *GNormalizeMode modus*) Gibt eine neu angelegte Kopie von *zkette* zurück, die normalisiert wurde. Dies bedeutet, dass Schreibweisen, für die es im Unicode mehrere Darstellungsformen gibt, in die Standardform gebracht werden. Somit wird beispielsweise entschieden, ob ein Zeichen mit Akzent als ein einziges Zeichen oder als eine Kombination eines akzentlosen Zeichens mit einem Zusatzakzent dargestellt wird. Nach welcher Methode die Normalisierung geschieht, entscheidet *modus* mit folgenden möglichen Werten:

g_utf8_normalize()

G_NORMALIZE_DEFAULT Normalisiert alles, was normalisiert werden kann, ohne den Textinhalt zu beeinflussen.

G_NORMALIZE_DEFAULT_COMPOSE Dito, versucht aber, das Resultat durch Verwenden möglichst kombinierter Zeichen so kompakt wie möglich zu halten.

G_NORMALIZE_ALL Normalisiert alles, auch wenn dies den Textinhalt beeinflusst, indem zum Beispiel eine hochgestellte Ziffer in eine normale Ziffer umgewandelt wird.

G_NORMALIZE_ALL_COMPOSE Dito mit möglichst kompaktem Resultat.

Hinweis

Sie sollten, bevor Sie zwei UTF-8-Zeichenketten vergleichen, stets beide mit derselben Methode normalisieren, da sonst Unterschiede zwischen zwei gleichwertigen Zeichenketten gefunden werden könnten.

g_utf8_collate(_key)()

gint g_utf8_collate(const gchar *zkette1, const gchar *zkette2) Vergleicht die beiden Zeichenketten *zkette1* und *zkette2* auf sprachlich möglichst korrekte Art und Weise. Zurückgegeben wird -1, wenn *zkette1* in der Sortierordnung vor *zkette2* stehen müsste, 0, wenn beide Zeichenketten gleichwertig sind, und sonst +1.

gchar* g_utf8_collate_key(zkette, laenge) Gibt einen Sortierschlüssel für *zkette* zurück. Der Vergleich zweier auf diese Weise berechneter Sortierschlüssel mit `strcmp()` erbringt dasselbe Ergebnis wie der Vergleich der UTF-8-Zeichenketten, aus denen die Schlüssel erzeugt wurden, mit `g_utf8_collate()`.

Hinweis

Wenn Sie eine Menge UTF-8-Zeichenketten oft untereinander vergleichen müssen, zum Beispiel beim Sortieren, sollten Sie vorher Sortierschlüssel für sie berechnen und dann nur noch die Schlüssel mit `strcmp()` vergleichen. Dies ist *erheblich* schneller, als jedes Mal `g_utf8_collate()` aufzurufen, was die Sortierschlüssel bei jedem Vergleich neu berechnen würde.

Der Konvertierung zwischen den verschiedenen Codierungen dienen einige Umwandlungsfunktionen. Sie lesen generell eine Zeichenkette *zkette* in einem Format und geben eine neu angelegte, durch ein Nullzeichen abgeschlossene Zeichenkette mit gleichem Inhalt in einem anderen Format zurück. In der als *gelesen* übergebenen Variable wird die Anzahl eingelesener Zeichen (das heißt bei UTF-8: Bytes; bei UTF-16: 16-Bit-Wörter; bei UCS-4: 32-Bit-Wörter) gespeichert, was dazu dienen kann, einen Fehler zu orten; *geschrieben* nimmt die Anzahl der im Ergebnis niedergeschriebenen Zeichen auf. An Stelle von *gelesen* beziehungsweise *geschrieben* kann jeweils auch NULL übergeben werden. Ist die Eingabezeichenkette im UTF-8-Format und *gelesen* gleich NULL, führt das Auftreten eines unvollständigen Zeichens am Ende der Eingabezeichenkette zu einem Fehler. In *fehler* wird gegebenenfalls ein Fehler der Klasse `G_CONVERT_ERROR` zurückgegeben; der Rückgabewert ist dann NULL.

- gunichar2* g_utf8_to_utf16(const gchar *z_kette, glong laenge, glong *gelesen, glong *geschrieben, GError **fehler)** Diese Funktion liest die UTF-8-Zeichenkette *z_kette* aus und gibt ihren Inhalt als eine neu angelegte nullterminierte UTF-16-Zeichenkette zurück. *g_utf8_to*()*
- gunichar* g_utf8_to_ucs4(z_kette, laenge, gelesen, geschrieben, fehler)** Dito, wandelt aber aus UTF-8 in UCS-4 um.
- gunichar* g_utf8_to_ucs4_fast(z_kette, laenge, geschrieben)** Wie *g_utf8_to_ucs4()*, ist aber etwa doppelt so schnell, da keinerlei Fehlerüberprüfung stattfindet. Die Parameter *gelesen* und *fehler* entfallen daher.
- gunichar* g_utf16_to_ucs4(const gunichar2 *z_kette, laenge, gelesen, geschrieben, fehler)** Wie *g_utf8_to_ucs4()*, wandelt aber aus UTF-16 in UCS-4 um. *g_utf16_to_**
- gchar* g_utf16_to_utf8(const gunichar2 *z_kette, laenge, gelesen, geschrieben, fehler)** Dito, wandelt aber aus UTF-16 in UTF-8 um.
- gunichar2* g_ucs4_to_utf16(const gunichar *z_kette, laenge, gelesen, geschrieben, fehler)** Dito, wandelt aber aus UCS-4 in UTF-16 um. *g_ucs4_to_**
- gchar* g_ucs4_to_utf8(z_kette, laenge, gelesen, geschrieben, fehler)** Dito, wandelt aber aus UCS-4 in UTF-8 um.
- gint g_unichar_to_utf8(gunichar z, gchar *puffer)** Wandelt das einzelne Unicode-Zeichen *z* in UTF-8-Darstellung um und schreibt diese in den Puffer *puffer*; dieser muss Platz für mindestens sechs Bytes aufweisen. Rückgabewert ist die Breite des in den Puffer geschriebenen Zeichens in Byte. Ist *puffer* NULL, geschieht keine Umwandlung und es wird nur die Breite zurückgegeben. *g_unichar_to_utf8()*

Mit diesen Funktionen können Sie nunmehr zwischen den verschiedenen Unicode-Codierungen hin und her konvertieren. Abschließend sei noch eine Reihe von Funktionen aufgezählt, die sich darum kümmern, Unicode aus ihrem und in ihren lokalen Zeichensatz umzuwandeln.

- gchar* g_locale_to_utf8(const gchar *z_kette, gssize laenge, gsize *gelesen, gsize *geschrieben, GError **fehler)** Arbeitet wie die oben genannten Zeichenketten-Umwandlungsfunktionen; wandelt *z_kette* aus der zur Zeit gültigen lokalen Zeichencodierung in UTF-8 um. *g_locale_to_utf8()*
- gchar* g_filename_to_utf8(z_kette, laenge, gelesen, geschrieben, fehler)** Dito, verwendet aber die lokale Codierung für Dateinamen *g_filename_to_utf8/uri()*
- gchar* g_filename_to_uri(const char *dateiname, const char *rechner, fehler)** Wandelt den Dateinamen *dateiname* in einen UTF-8-codierten URI mit Umschreibungssequenzen um. Gegebenenfalls wird darin der Rechnername *rechner* verwendet (sofern nicht NULL). Der übergebene Dateiname muss ein vollständiger Pfadname sein.

`g_utf8_to_locale()` **gchar* g_utf8_to_locale(*zkette*, *laenge*, *gelesen*, *geschrieben*, *fehler*)** Die Umkehrfunktion zu `g_locale_to_utf8()`: wandelt die UTF-8-Zeichenkette *zkette* in die lokal gültige Zeichencodierung um.

`g_filename_from_utf8/uri()` **gchar* g_filename_from_utf8(*zkette*, *laenge*, *gelesen*, *geschrieben*, *fehler*)** Dito für Dateinamen.

gchar* g_filename_from_uri(const char *uri, char **rechner, fehler) Wandelt den UTF-8-codierten und mit Umschreibungssequenzen dargestellten URI *uri* in einen lokalen Dateinamen um. Falls ein Rechnername im URI enthalten war, wird dieser in *rechner* abgelegt, ansonsten NULL. Interessiert der Rechnername nicht, kann auch NULL an Stelle von *rechner* übergeben werden.

Mögliche Fehler

In der Fehlerklasse `G_CONVERT_ERROR` sind folgende Fehler definiert:

G_CONVERT_ERROR_NO_CONVERSION Die gewünschte Konvertierung ist gar nicht möglich.

G_CONVERT_ERROR_ILLEGAL_SEQUENCE In der Eingabezeichenkette ist eine ungültige Bytefolge aufgetaucht.

G_CONVERT_ERROR_FAILED Fehlschlag aus unbekanntem Grund.

G_CONVERT_ERROR_PARTIAL_INPUT Die Eingabezeichenkette enthielt eine bruchstückhafte Zeichenfolge.

G_CONVERT_ERROR_BAD_URI Ein URI war aus irgendeinem Grund ungültig.

G_CONVERT_ERROR_NOT_ABSOLUTE_PATH Ein Pfad, der eigentlich absolut sein sollte, war es nicht.

2.3.5 Timer

`GTimer` Ein `GTimer` ist nichts anderes als eine Stoppuhr, die so genau arbeitet, wie es die Systemuhr erlaubt.

Listing 2.3
`gtimerdemo.c`

```
int main(int argc, char **argv)
{
    GTimer *uhr = NULL;
    gint i;
    gdouble vergangene_zeit;
    gulong us;

    uhr = g_timer_new();
    g_timer_start(uhr);
    g_print("Uhr läuft\n");
```

```

g_print("Warteschleife wird begonnen\n");
for (i=0; i<DURCHLAEUFE; i++)
    /* nur Programmierbeispiele dürfen Zeit und
       Rechenleistung so stilvoll vergeuden */;

g_print("Warteschleife beendet\n");
g_timer_stop(uhr);

vergangene_zeit = g_timer_elapsed(uhr, &us);
g_print("Vergangen: %g s\n", vergangene_zeit);
g_print("          %ld us\n", us);

g_timer_destroy(uhr);

return 0;
}

```

Das Beispielprogramm veranschaulicht eigentlich bereits alles, was man über `GTimer` wissen muss: Die Variable `uhr`, die den `GTimer` enthalten soll, ist wie üblich ein Zeiger, der zunächst auf `NULL` initialisiert wird. Durch Zuweisen des Rückgabewertes von `g_timer_new()` wird eine neue `GTimer`-Struktur angelegt, auf die `uhr` zeigt.

`g_timer_start()` stößt die Zeitmessung an. Das Programm durchläuft danach eine Warteschleife, die absolut nichts tut, außer Zeit und Leistung zu verbrauchen³; anschließend stoppt `g_timer_stop()` die Uhr. `g_timer_continue()` lässt einen gestoppten Timer weiterlaufen.

Der Stand der Stoppuhr kann während oder nach der Laufzeit mit `g_timer_elapsed()` ausgelesen werden; der Rückgabewert ist eine Sekundenzahl als Fließkommawert mit doppelter Genauigkeit. Dazu nimmt `g_timer_elapsed()` neben dem Zeiger auf das Timer-Objekt auch noch einen Zeiger auf eine lange vorzeichenlose Ganzzahlvariable vom Typ `gulong` entgegen. Falls dieser Zeiger (wie im Beispiel) ungleich `NULL` ist, wird in diese Variable der Nachkommanteil der vergangenen Zeit in Mikrosekunden geschrieben; andernfalls wird sie ignoriert.

`g_timer_new()`
`g_timer_start()`
`g_timer_stop()`
`g_timer_continue()`
`g_timer_elapsed()`

Brauchen Sie nur die gebrochene Sekundenzahl, übergeben Sie einfach `NULL` als zweiten Parameter.

Hinweis

³Kompilieren Sie das Programm, wenn Sie es ausprobieren, bitte ohne Optimierung, damit die Schleife nicht wegoptimiert wird.

2.3.6 Systeminteraktion

Dieser Abschnitt beschreibt zahlreiche Funktionen, die der Interaktion mit dem das Programm umgebenden System, gleich welcher Plattform, dienen. In erster Linie geht es dabei um den Umgang mit Dateinamen und den Kontext von Programmen, wie Benutzernamen oder Standardverzeichnisstrukturen für verschiedene Zwecke.

Pfade und Dateinamen

Die folgenden Funktionen dienen dem portablen Umgang mit Verzeichnispfaden und Dateinamen; sie sind im Besonderen deshalb notwendig, weil unterschiedliche Plattformen unterschiedliche Trennzeichen und Zeichencodierungen für Pfade verwenden. Die Funktionen arbeiten daher je nach Plattform mit unterschiedlichen Zeichenformaten, akzeptieren diese aber jeweils untereinander.

<code>g_get_current_dir()</code>	gchar *g_get_current_dir() Gibt das aktuelle Arbeitsverzeichnis zurück.
<code>g_path_is_absolute()</code>	gboolean g_path_is_absolute(const gchar *dateiname) Gibt genau dann TRUE zurück, wenn <i>dateiname</i> ein absoluter Pfad ist, also mit einem Pfadtrenner und gegebenenfalls mit einem Laufwerksbuchstaben beginnt.
<code>g_path_skip_root()</code>	const gchar* g_path_skip_root(dateiname) Gibt einen Zeiger auf das erste Zeichen in <i>dateiname</i> hinter dem Wurzelteil (Pfadtrenner und eventueller Laufwerksbuchstabe) zurück; falls es sich nicht um einen absoluten Pfad handelt, ist das Ergebnis NULL.
<code>g_path_get_dirname/basename()</code>	gchar* g_path_get_dirname(dateiname) Gibt den in <i>dateiname</i> enthaltenen Pfad zurück. Enthält der Dateiname keinen Pfad, ist das Ergebnis ".". Die zurückgegebene Zeichenkette muss freigegeben werden. gchar* g_path_get_basename(dateiname) Dito, gibt aber den Basisnamen, also den letzten Teil von <i>dateiname</i> zurück: Dies ist entweder ein Dateiname, falls <i>dateiname</i> mit einem Pfadtrenner endet, ein Verzeichnisname, oder falls <i>dateiname</i> ein Wurzelverzeichnis (gegebenenfalls mit Laufwerksbuchstaben) bezeichnet, ein einzelner Pfadtrenner. Ist <i>dateiname</i> leer, wird "." zurückgegeben.
<code>g_build_filename(v)</code>	gchar* g_build_filename(const gchar *teill, ...) Nimmt eine mit NULL abgeschlossene Reihe von Zeichenketten entgegen und setzt sie mit dem auf der gegenwärtigen Plattform gültigen Pfadtrenner zu einem Dateinamen zusammen. Leere Zeichenketten werden ignoriert, Dopplungen von Pfadtrennern, die durch das Zusammensetzen entstehen, beseitigt. Das Resultat muss freigegeben werden. gchar* g_build_filenamev(gchar **teile) Dito, verwendet allerdings ein mit NULL abgeschlossenes Feld von Zeichenketten als Eingabe.

Benutzerkontext

Die in Tabelle 2-4 aufgelisteten Funktionen haben keine Argumente und geben Zeichenketten (`const gchar*`) zurück. Mit ihnen können Sie Daten über den Benutzer ermitteln, vor allem, welche Verzeichnisse für welche Zwecke angebracht sind.

Funktion	Zurückgegebene Zeichenkette
<code>g_get_host_name()</code>	Rechnername, sofern ermittelbar, sonst "localhost"
<code>g_get_user_name()</code>	System-Benutzername
<code>g_get_real_name()</code>	Voller Name des Benutzers, sofern ermittelbar, sonst "Unknown"
<code>g_get_home_dir()</code>	Heimatverzeichnis des Benutzers
<code>g_get_tmp_dir()</code>	Standardverzeichnis für temporäre Dateien
<code>g_get_user_data_dir()</code>	Standard-Wurzelverzeichnis für allgemeine benutzerspezifische Daten
<code>g_get_user_config_dir()</code>	Standard-Wurzelverzeichnis für benutzerspezifische Konfigurationsdaten
<code>g_get_user_cache_dir()</code>	Standard-Wurzelverzeichnis für benutzerspezifische, zwischengespeicherte Daten (können jederzeit von dritten Anwendungen gelöscht werden)

Tabelle 2-4

Benutzerkontext

Dieser Mechanismus ist der einzige, den Sie verwenden sollten, um solche Verzeichnisse festzustellen, da er den XDG-Standard für Standard-Wurzelverzeichnisse [3] implementiert und unter Windows versucht, ebenfalls einigermaßen das Richtige zu tun. Dies bedeutet insbesondere, dass Sie darauf verzichten sollten, auf eigene Faust anderswo »Dotfiles« anzulegen. Beachten Sie, dass die Funktionen Wurzelverzeichnisse zurückgeben und Sie ein Unterverzeichnis mit dem Namen Ihres Programms darin anlegen müssen.

Hinweis

Um die im Standard definierten weiteren Spezialverzeichnisse des Benutzers zu ermitteln, müssen Sie einen der in Tabelle 2-5 erwähnten Werte an `g_get_user_special_dir()` übergeben (Rückgabewert ebenfalls `const char*`).

Die Speicherorte für systemweite, also nicht benutzerspezifische, Daten ermitteln Sie mit den folgenden Funktionen:

`g_get_user_special_dir()`

Tabelle 2-5
Kennungen der
Benutzer-
Spezialverzeichnisse

<code>g_get_user_special_dir(...)</code>	Zurückgegebene Zeichenkette
<code>G_USER_DESKTOP</code>	Desktop-Verzeichnis des Benutzers
<code>G_USER_DOCUMENTS</code>	Dokumente-Verzeichnis
<code>G_USER_DOWNLOAD</code>	Download-Verzeichnis
<code>G_USER_MUSIC</code>	Musik-Verzeichnis
<code>G_USER_PICTURES</code>	Bilder-Verzeichnis
<code>G_USER_PUBLIC_SHARE</code>	Öffentlich freigegebenes Verzeichnis
<code>G_USER_TEMPLATES</code>	Vorlagen-Verzeichnis
<code>G_USER_VIDEOS</code>	Video-Verzeichnis

`g_get_system_data/config_dirs()`

const gchar*const* g_get_system_data_dirs() Das zurückgegebene Feld von Zeichenketten enthält, in der Reihenfolge ihres Auftretens im Suchpfad, die Standard-Wurzelverzeichnisse für systemweite Anwendungsdaten (üblicherweise `/usr/local/share`, `/usr/share` etc.).

const gchar*const* g_get_system_config_dirs() Dito, allerdings für systemweite Konfigurationsdaten (üblicherweise `/etc/xdg`).

Hinweis

Rechnen Sie nicht damit, in diese Pfade schreiben zu können, und denken Sie auch hier daran, dass Sie den Namen Ihrer Anwendung als Unterverzeichnis anhängen müssen.

Anwendungsnamen

`g_get/set_application_name()`

Ihre Anwendungen sollten, um zum Beispiel in Fehlermeldungen oder in der Fensterliste einen aussagekräftigen Namen anzuzeigen, diesen der GLib mitteilen. Hierzu dient `g_set_application_name()`. Das einzige Argument ist der Anwendungsname als C-Zeichenkette (`const gchar*`). Da es sich um einen für Benutzer lesbaren Namen handeln soll, sollten Sie diese Zeichenkette lokalisieren (siehe auch Abschnitt 6.5).

`g_get/set_prname()`

Reine Befehlszeilenprogramme sollten außerdem ihren Programmnamen (also normalerweise den Wert von `argv[0]`) auf dieselbe Weise mit `g_set_prname()` einstellen. Bei grafischen Anwendungen ist dies unnötig, weil es von den Initialisierungsfunktionen übernommen wird.

Warnung

Beide Funktionen dürfen je nur ein einziges Mal aufgerufen werden.

Die so eingestellte konstante Zeichenkette kann danach durch einen Aufruf von `g_get_application_name()` beziehungsweise `g_get_prpname()` abgefragt werden.

Umgebungsvariablen

Nicht fehlen darf natürlich eine API für Umgebungsvariablen. Sie besteht aus folgenden Funktionen, die im Gegensatz zur traditionellen C-API die Portabilität nach Windows gewährleisten. Für Variablennamen und -werte verwendet GLib auf jeder Plattform die jeweils auch für Dateinamen verwendete Zeichencodierung. Variablennamen dürfen nicht das Zeichen '=' enthalten.

- gchar** g_listenv()** liefert ein durch NULL abgeschlossenes Feld mit den Namen aller definierten Umgebungsvariablen zurück, das nach Gebrauch mit `g_strfreev()` freigegeben werden muss. *g_listenv()*
- const gchar* g_getenv(const gchar* name)** liefert den Wert der Umgebungsvariablen *name* zurück. *g_get/setenv()*
- gboolean g_setenv(name, const gchar *wert, gboolean ersetzen)** setzt die Variable *name* auf den Wert *wert*. Falls sie bereits definiert ist, muss *ersetzen* TRUE sein, damit der neue Wert den alten überschreibt. Der Rückgabewert ist genau dann TRUE, wenn die Zuweisung erfolgreich ausgeführt wurde.
- void g_unsetenv(name)** entfernt die Variable *name* aus der Umgebung. *g_unsetenv()*

Die Funktion `g_find_program_in_path()` nimmt einen Programmnamen als Zeichenkette entgegen und gibt den Pfad zum Programm zurück, wenn es im aktuellen Ausführungspfad (PATH) gefunden werden kann, sonst NULL. Ein etwaiges Ergebnis muss freigegeben werden. *g_find_program_in_path()*

2.3.7 Meldungen

Zum Absetzen von Protokollmeldungen auf der Konsole, was insbesondere zur Fehlerdiagnose nützlich ist, bietet die GLib verschiedene Funktionen. Dazu gehören in der Reihenfolge der Dringlichkeit des Anzeigtenen:

- **g_debug()** für Diagnosemeldungen zur Fehlersuche *g_debug*
- **g_message()** für informative Meldungen, mit denen sich kein Problem verbindet *g_message()*
- **g_warning()** für Warnungen über Probleme, die den allgemeinen Ablauf des Programmes jedoch nicht gefährden *g_warning()*
- **g_critical()** für Warnungen zu kritischen Zuständen *g_critical()*
- **g_error()** für fatale Fehler; führt zum Programmabbruch *g_error()*

`G_LOG_DOMAIN`

Sie nehmen auf gleiche Weise wie `printf()` eine Formatzeichenkette und eine Liste von weiteren Parametern entgegen. Damit Sie oder spätere Benutzer identifizieren können, dass eine Meldung von Ihrer Software stammt, sollten Sie zudem das Makro `G_LOG_DOMAIN` definieren, und zwar als eine kurze Zeichenkette, die die Anwendung oder die Bibliothek identifiziert. Die meisten Pakete in der G-Welt tun dies auf der Compilerbefehlszeile (... `-DG_LOG_DOMAIN=\"Name\"` ...).

`messagedemo.c`

Ein Demonstrationsprogramm für die verschiedenen Funktionen finden Sie in `messagedemo.c`.

Dringlichkeitsstufen als fatal markieren

`g_log_set_always_fatal()`

`G_LOG_LEVEL_*`

Nach dessen Ausführung sollten Sie, `g_error()` sei Dank, eine Meldung darüber sehen, dass der Prozess abgebrochen und eventuell ein CoreDump erzeugt wurde. Es lässt sich übrigens auch konfigurieren, dass Meldungen anderer Dringlichkeitsstufen als die von `g_error()` ebenfalls als fatal eingestuft werden und so zum Programmabbruch führen. `g_log_set_always_fatal(G_LOG_LEVEL_WARNING | G_LOG_LEVEL_CRITICAL)` tut genau das, wonach es aussieht: Das Argument ist eine Bitmaske, die sich durch das Verknüpfen einer oder mehrerer der folgenden Konstanten `G_LOG_LEVEL_CRITICAL`, `G_LOG_LEVEL_WARNING`, `G_LOG_LEVEL_MESSAGE`, `G_LOG_LEVEL_INFO` und `G_LOG_LEVEL_DEBUG` durch ein bitweises ODER ergibt. Meldungen der entsprechenden Dringlichkeitsstufen sind danach fatal.

Hinweis

Sofern Ihre Anwendung GTK+ verwendet, können Sie durch Übergeben des Parameters `--g-fatal-warnings` auf der Befehlszeile den Effekt erreichen, dass Meldungen sämtlicher Dringlichkeitsstufen als fatal gewertet werden.

`g_print(err)()`

Wenn Sie eine Ausgabe machen wollen, die nicht die Form einer vorformatierten Meldung haben soll, können Sie dies mit `g_print()` und `g_printerr()` tun. Aufgerufen werden beide Funktionen wie `printf()`; `g_print()` schreibt dabei auf die Standardausgabe `stdout` und `g_printerr()` auf den Standard-Fehlerstrom `stderr`.

Hinweis

Im Gegensatz zu den weiter oben besprochenen Meldefunktionen wie `g_message()` hängen `g_print()` und `g_printerr()` keinen Zeilenumbruch an ihr Argument an.

`printhandlerdemo.c`

Sie werden sich jetzt wahrscheinlich fragen, warum Sie nicht einfach weiterhin mit `fprintf()` auf die Ströme schreiben sollen. Ein Grund wird in `printhandlerdemo.c` demonstriert (falls es Sie wundert, was die Funktion `mein_printerr_handler()` dort genau tut, finden Sie Nä-

heres weiter unten in Abschnitt 2.4.1): Sie können für `g_print()` und `g_printerr()` getrennt eigene Verarbeitungsfunktionen für die mit diesen Funktionen abgesetzten Ausgaben definieren.

Wie Sie sehen, passiert dies mit den Funktionen `g_set_print_handler()` beziehungsweise `g_set_printerr_handler()`. Ihr einziges Argument ist eine Funktion vom Typ `GPrintFunc`, das heißt, eine `void`-Funktion, die eine C-Zeichenkette als einziges Argument hat:

```
/* aus $(PREFIX)/include/glib-2.0/glib/gmessages.h */
typedef void (*GPrintFunc) (const gchar *string);
```

*`g_set_print*_
handler()`
`GPrintFunc`*

Eine solche Funktion kann nur eine Zeichenkette als Parameter entgegennehmen. `g_print()` beziehungsweise `g_printerr()` verlieren also mit der Definition eines Handlers ihre Fähigkeit, wie `printf()` ein Format und eine Parameterliste zu verarbeiten. Sie sollten daher generell nur fertig formatierte Zeichenketten an `g_print()` oder `g_printerr()` übergeben, damit es nicht zu bösen Überraschungen kommt, falls die Handler umdefiniert werden.

Warnung

Eine mögliche Anwendung wäre es beispielsweise, entscheiden zu können, ob Meldungen in einem mitlaufenden Protokollfenster oder jedes Mal in einem neuen Dialogfenster erscheinen.

An dieser Stelle sinnvoll zu erwähnen sind noch zwei Funktionen, die im weitesten Sinne auch mit Fehlermeldungen zu tun haben. Es handelt sich um `g_strerror()` und `g_strsignal()`, plattformunabhängige Implementationen von `strerror()` und `strsignal()`. Diese Funktionen geben zu Fehlercodes wie `EBADF` oder `EINVAL` (im Falle von `g_strerror()`) beziehungsweise zu Signalcodes wie `SIGINT` oder `SIGPIPE` (im Falle von `g_strsignal()`) die entsprechende Meldung als Zeichenkette zurück. Die zurückgegebenen Zeichenketten sind intern und dürfen weder freigegeben noch geändert werden.

*`g_strerror()`
`g_strsignal()`*

Der Vorteil von `g_strerror()` und `g_strsignal()` gegenüber von `strerror()` und `strsignal()` ist, dass die GLib-Funktionen auf allen unterstützten Plattformen sicher verfügbar sind; außerdem sind die zurückgegebenen Meldungen in UTF-8 gehalten und können daher zum Beispiel ohne Konvertierung problemlos mit GTK+ dargestellt werden.

2.3.8 Fehlerdiagnose

Diverse Funktionen der GLib sollen Ihnen die Fehlerjagd erleichtern. Zwei davon tun im Prinzip nichts anderes als `return`. Der Unterschied ist, dass sie zusätzlich noch eine Meldung der Dringlichkeitsstufe ›kritisch‹ absetzen. Sie können sie also verwenden, um Programmstellen, die im Kontrollfluss nie erreicht werden sollen, abzusichern.

`g_return(_val)_if_reached()` entspricht `return` in void-Funktionen; `g_return_val_if_reached(wert)` entspricht `return wert` in jeder anderen Funktion. `g_return_if_fail()` und `g_return_val_if_fail()` funktionieren genauso, nehmen aber zusätzlich noch einen Testausdruck entgegen. Der Rücksprung aus der Funktion und die kritische Meldung erfolgen nur, wenn dieser Ausdruck `FALSE` ergibt. Mit `g_return*_if_*`() wird in GNOME häufig am Kopf von Funktionen überprüft, ob die als Argumente übergebenen Objekte die richtigen Typen haben.

Ebenso wie diese beiden Funktionen implementieren auch die nächsten eine Art von rudimentärem Vertragskonzept, in diesem Fall den Klassiker, die Zusicherung – eine Bedingung, die wahr sein muss, damit es Sinn hat, das Programm weiter auszuführen.

`g_assert()` **g_assert()** bricht das Programm mit `g_error()` ab, falls der als Parameter angegebene Ausdruck `FALSE` ergibt.
`g_assert_not_reached()` **g_assert_not_reached()** bricht in jedem Fall ab und ist sozusagen die härtere Version von `g_return_if_reached()`.

Beispiele für Zusicherungen finden sich an einigen Stellen in diesem Buch sowie in den meisten GNOME-Anwendungen. Auch intern sind die Funktionen der GNOME-Plattformbibliotheken größtenteils mit solchen Zusicherungen gegen grob falsche Parameter gesichert.

Falls Ihr Programm so weit ist, dass Sie sicher sind, dass keine Zusicherungen mehr gebrochen werden, können Sie beim Kompilieren das Makro `G_DISABLE_ASSERT` definieren (zum Beispiel, indem Sie auf der Compilerbefehlszeile die Option `»-DG_DISABLE_ASSERT«` angeben). Dies führt dazu, dass alle Zusicherungen ignoriert werden; die für die Prüfungen benötigte Rechenzeit wird eingespart.

2.3.9 Ausnahmebehandlung

Die genannten Routinen zur Fehlerdiagnose dienen dazu, Programmierfehler, die zur Laufzeit auftreten, zu finden und zu beheben. Es gibt jedoch noch eine andere Art von Fehlern, die *überlebba*ren Fehler oder Ausnahmen, beispielsweise, wenn ein Programm versucht, eine Datei zu öffnen, und dies aus irgendeinem Grunde fehlschlägt.

Der klassische Weg der Ausnahmebehandlung in C ist es, Funktionen einen Fehlercode zurückgeben zu lassen, der dann abgefragt werden kann; eventuell wird auch ein Fehlercode in einer globalen Variable wie `errno` hinterlassen. In höheren Sprachen gibt es ganze Ausnahmebehandlungssysteme, wie zum Beispiel der `try{}/throw()/catch(){}-Mechanismus` in C++ oder sein Äquivalent in Java.

In der GLib gibt es keine so komplexe Einrichtung, aber immerhin etwas Komfortableres als das, was in reinem C normalerweise geübt wird; das zuständige Subsystem heißt GError, und die gleichnamige Datenstruktur GError bildet seinen Kern. Ebenso wichtig wie dieser Datentyp sind jedoch auch die Konventionen für den Umgang damit.

GError

Ausnahmen behandeln

Funktionen, die ihre Ausnahmebehandlung über GError abwickeln, erhalten als letztes Argument einen Zeiger auf eine Variable vom Typ GError*. Dieser Zeiger kann NULL sein, in welchem Fall die GError-Ausnahmebehandlung gar nicht erst wirksam wird. Ist der Zeiger jedoch nicht NULL, sondern zeigt auf eine NULL-initialisierte GError*-Variable, zeigt er, sofern ein Fehler aufgetreten ist, nach der Rückkehr des Funktionsaufrufs auf eine neu angelegte GError-Struktur.

Diese Struktur hat nun folgende Felder:

domain (Typ GQuark) Die sogenannte Fehlerklasse des Fehlers: eine Kennung dafür, in welchem Modul oder Subsystem der Fehler aufgetreten ist. Es muss für jede Fehlerklasse ein Makro des Namensformats *NAMENSRAUM_MODUL_ERROR* (Beispiel: *G_FILE_ERROR*) existieren, das zu einem Funktionsaufruf expandiert, der diese Konstante zurückgibt.

code (Typ gint) Der Fehlercode: die Kennung des Fehlers innerhalb der Fehlerklasse. Für jede hier mögliche Kennung muss ein Symbol des Formats *NAMENSRAUM_MODUL_ERROR_CODE* definiert sein, das einem Aufzählungstypen namens *NamensraumModulError* angehören soll (Beispiel: *G_FILE_ERROR_PIPE* in *GFileError*).

message Eine natursprachliche, ausführliche Fehlermeldung

Die einfachste Art, das Ergebnis einer ihre Fehler mit GError behandelnden Funktion abzufragen, sehen Sie in Listing 2.4.

```
GError *fehler = NULL;

/* den GError als letztes Argument übergeben */
tu_was(arg1, arg2, &fehler);

/* ist ein Fehler aufgetreten? */
if (fehler != NULL)
{
    /* eine Meldung ausgeben */
    g_printerr("Problem beim Etwas-Tun: %s", fehler->message);
}
```

Listing 2.4

Einfache
Fehlerbehandlung mit
GError

```

    /* Fehlerstruktur freigeben */
    g_error_free(fehler);
}

```

Sie sehen im Listing auch gleich die Funktion, die aufgerufen wird, um einen GError, nachdem er seine Schuldigkeit getan hat, freizugeben: Sie heißt `g_error_free()` und nimmt den GError als einziges Argument entgegen.

Wenn Sie noch etwas anderes tun möchten, als nur wie in Listing 2.4 eine Meldung auszugeben, müssen Sie natürlich abfragen, in welche Fehlerklasse der zurückgegebene Fehler fällt und welchen Code er hat. Statt dies von Hand zu prüfen, sollten Sie die mitgelieferte Vergleichsfunktion `g_error_matches()` verwenden. Diese nimmt als erstes Argument den GError entgegen, als zweites eine Fehlerklasse und als drittes einen Fehlercode. Hat der übergebene GError genau diese Klasse und diesen Code, gibt die Funktion `TRUE` zurück, andernfalls `FALSE`.

Listing 2.5
*Ursachenabhängige
 Fehlerbehandlung mit
 GError*

```

GError *fehler = NULL;
gchar *dateiname;
BluesGitarreTeuer *gitarre;
BluesGitarreBillig *ersatzgitarre;

<< ... >>

blues_improvisieren(dateiname, gitarre,
                    BLUES_A_DUR, &fehler);
if (fehler != NULL)
    if (g_error_matches(fehler, BLUES_GITARRE_ERROR,
                        BLUES_GITARRE_ERROR_KAPUTT))
        {
            g_clear_error(&fehler);
            blues_improvisieren(dateiname, ersatzgitarre,
                                BLUES_A_DUR, &fehler);
        }

/* wenn's denn gar nicht geklappt hat, auf Clapton
   zurückgreifen */
if (fehler != NULL)
{
    dateiname = g_strdup("clapton-1966.wav");
    g_error_free(fehler);
}

blues_abspielen(dateiname);

```

Im Beispiel in Listing 2.5 können Sie sehen, wie mit `g_error_matches()` gearbeitet wird. Das Beispielprogramm ruft die Funktion `blues_improvisieren()` auf; tritt dabei der Fehler `BLUES_GITARRE_ERROR_KAPUTT` in der Fehlerklasse `BLUES_GITARRE_ERROR` auf, soll die Funktion noch einmal mit anderen Argumenten aufgerufen werden. Hierzu wird zunächst der `GError` mit der Funktion `g_clear_error()` freigegeben und der Zeiger darauf auf `NULL` gesetzt.

`g_clear_error()`

Ein `GError*` muss nach jedem Gebrauch sofort ausgewertet und gereinigt werden; es ist nicht möglich, irgendwie mehrere Funktionen hintereinander in denselben `GError`-Zeiger schreiben zu lassen, da dieser ja nur auf einen Fehler gleichzeitig zeigen kann. Der Aufruf von `GError` verwendenden Funktionen hintereinander ohne zwischenzeitliches Reinigen des `GErrors` ist ein Speicherleck.

Warnung

In jedem Fall wird in diesem Beispiel, wenn selbst nach dem zweiten Aufruf noch ein Fehler in `fehler` steht, das Problem durch die Zuweisung von `"clapton-1966.wav"` behoben und der Fehler freigegeben, so dass `blues_abspielen()` schließlich mit einem gültigen Dateinamen aufgerufen werden kann.

Ausnahmen selbst definieren

Um in Ihren eigenen Funktionen Ausnahmen über `GError` bekannt zu machen, müssen Sie folgende Schritte befolgen:

1. Eine Fehlerklasse definieren, das heißt, ein Makro im genormten Format, das zu einem Funktionsaufruf expandiert, der eine eindeutige `GQuark`-Kennung zurückgibt.
2. Die Fehlercodeaufzählung mit den Fehlercodesymbolen definieren.
3. In den fraglichen Funktionen als letztes reguläres Argument (das heißt, als letztes Argument, aber vor einer eventuellen `...`-Argumentliste) einen `GError**` (in Worten: einen Zeiger auf einen Zeiger auf eine `GError`-Struktur) entgegennehmen.
4. Im Funktionsrumpf gegebenenfalls einen frischen `GError` anlegen und über diesen Zeiger zurückgeben.

```
/* Fehlerklasse definieren */
#define MAWA_ETWASTUN_ERROR (mawa_etwastun_error_quark())

/* Fehlercodes definieren */
typedef enum
{
    MAWA_ETWASTUN_ERROR_PANIK,
```

Listing 2.6

Eigene Ausnahmen
definieren

```

MAWA_ETWASTUN_ERROR_DESINTERESSE,
MAWA_ETWASTUN_ERROR_MARIENERSCHEINUNG,
MAWA_ETWASTUN_ERROR_FAILED /* Ausweichcode */
} MawaEtwastunError;

/* Funktion, die das GQuark zurückliefert */
GQuark mawa_etwastun_error_quark(void)
{
    static GQuark q = 0;
    if (0 == q)
        q = g_quark_from_static_string("mawa-etwastun-error-quark");

    return q;
}

```

Beachten Sie im vorstehenden Listing besonders die Funktion `mawa_etwastun_error_quark()`. Sie liefert das Kennungs-GQuark für die Fehlerklasse `MAWA_ETWASTUN_ERROR` zurück; damit dieses nicht bei jedem Aufruf neu berechnet werden muss, wird es in einer statischen Variable gehalten.

Listing 2.7

Ausnahmen in eigenen Funktionen verwenden

```

void einfach_was_tun(GError **fehler)
{
    gint i;
    gboolean hat_geklappt;

    << etwas tun >>

    if (!hat_geklappt)
        g_set_error(fehler, MAWA_ETWASTUN_ERROR,
                   MAWA_ETWASTUN_ERROR_PANIK,
                   "Panik in einfach_was_tun, i = %d", i);
}

void verschachtelt_was_tun(GError **fehler)
{
    gint i;
    gboolean hat_geklappt;
    GError *fehlerpuffer = NULL;

    << etwas tun >>

    if (!hat_geklappt)
    {
        g_set_error(fehler, MAWA_ETWASTUN_ERROR,
                   MAWA_ETWASTUN_ERROR_PANIK,
                   "Panik in verschachtelt_was_tun, i = %d",

```

```

        i);
    return;
}

einfach_was_tun(&fehlerpuffer);
if (NULL != fehlerpuffer)
{
    << sonstige Fehlerbehandlung >>
    g_propagate_error(fehler, fehlerpuffer);
}
}

```

Schauen Sie sich in Listing 2.7 zunächst die Funktion `einfach_was_tun()` an. Sie tut etwas, und falls es nicht klappt, ruft sie vor der Rückkehr die Funktion `g_set_error()` auf. Diese erhält `fehler` als erstes Argument und schreibt, sofern `fehler` nicht `NULL` ist, einen Zeiger auf einen frisch angelegten `GError` hinein. Zweites und drittes Argument sind Fehlerklasse und Fehlercode; dann folgt eine Formatzeichenkette à la `printf()` mit der entsprechenden Variablenliste.

`g_set_error()`

Die Funktion `verschachtelt_was_tun()` macht im Prinzip nichts anderes. Beachten Sie jedoch, dass sie `einfach_was_tun()` aufruft und dazu nicht einfach `fehler` übergibt – denn `fehler` kann ja `NULL` sein. Stattdessen wird ein eigener `GError`, `fehlerpuffer`, verwandt.

Geben Sie nie einen `GError**` einfach weiter. Es kann nie vorausgesetzt werden, dass er nicht `NULL` ist.

Warnung

Um die Fehlerinformation von `fehlerpuffer` gegebenenfalls nach außen weiterzugeben, wird `g_propagate_error()` eingesetzt. Erstes Argument ist ein `GError**`, also normalerweise das Argument, durch das der Fehler zurückgegeben wird; zweites Argument ist ein `GError*`, also normalerweise eine in der Funktion selber existierende Fehlervariable. Was `g_propagate_error()` nun macht, ist einfach, den am Zeiger hängenden `GError` entweder nach draußen weiterzureichen oder, wenn das erste Argument `NULL` ist, ihn freizugeben.

`g_propagate_error()`

Sie sehen, dass das Hauptaugenmerk bei der Entwicklung von `GError` darauf gelegen hat, Transparenz im Umgang mit `NULL` zu erreichen, so dass niemand einen `GError` an eine Funktion übergeben und anschließend freigeben muss, der gar kein Interesse an Fehlerinformationen hat; und so, dass nicht am Schluss eines Vorgangs nicht freigegebene `GErrors` herumliegen.

Wird `NULL` statt eines Fehlerzeigers in Ihre Funktion hineingereicht, können Sie dies zum Zeichen nehmen, dass der Benutzer selbst keine