Modernes C

Praktikum „C-Programmierung“

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Eugen Betke, Nathanael Hübbe,
Michael Kuhn, Jakob Lüttgau, Jannek Squar

2019-01-28

Wissenschaftliches Rechnen
Fachbereich Informatik
Universität Hamburg

Modernes C

Geschichte [2]

- Erinnerung: In den 1970ern von Dennis Ritchie entwickelt
  - Seitdem stetige Weiterentwicklung
- Funktionsumfang wird durch Standards festgelegt
  - Bis 1989: Buch “The C Programming Language” von Brian W. Kernighan und Dennis Ritchie als Quasi-Standard (K&R C)
  - Ab 1989: Standardisierung durch ANSI (ANSI C, C89)
  - Ab 1990: Internationale Norm durch ISO (C90, entspricht C89)
  - Ab 1995: Erste Erweiterung (C95)

Geschichte… [2]

- Ab 1999: Neuer Standard (C99)
  - Größtenteils mit C90 kompatibel
  - Neue Funktionen, teilweise von C++ übernommen
    - Echte Booleans, komplexe Zahlen etc.
- Ab 2011: Neuer Standard C11
  - Neue Funktionen und bessere Kompatibilität mit C++
    - Atomare Datentypen, Threads etc.
- Ab 2018: Neuer Standard C18 (entspricht C11 plus Fehlerkorrekturen)

Fehlerbehandlung

- Fehlerbehandlung kann in C recht aufwendig sein
  - Keine Unterstützung für Exceptions etc.
- Häufiges Muster:
  - Eine Funktion soll vorzeitig verlassen werden
  - Allokierte Variablen müssen aufgeräumt werden etc.
- Grundsätzlich zwei Möglichkeiten
  - Code-Duplikation in jedem Fehlerfall
  - `goto`-Marke für Fehlerfälle

Cleanup v0

```c
#include <stdlib.h>

void foo (char* bar) {
    char* foo = malloc(1024);
    if (bar == NULL) {
        free(foo);
        return;
    }
    free(foo);
}

int main (void) {
    foo(NULL);
    return 0;
}
```

- `free` muss in jeder Fehlerabfrage wiederholt werden

Cleanup v1

```c
#include <stdlib.h>

void foo (char* bar) {
    char* foo = malloc(1024);
    if (bar == NULL)
        goto error;
error:
    free(foo);
}

int main (void) {
    foo(NULL);
    return 0;
}
```

- `goto` erlaubt das Anspringen einer speziellen Fehlerbehandlung
  - In diesem Fall identisch zum normalen Funktionsende

Cleanup v2

```c
#include <stdlib.h>

void cleanup_free (void* p) { free(*((void**)p)); }

void foo (char* bar) {
    __attribute__((cleanup(cleanup_free))) char* foo = malloc(1024);
    if (bar == NULL)
        return;
}

int main (void) {
    foo(NULL);
    return 0;
}
```

- `cleanup`-Attribut erlaubt das Aufräumen am Ende des Gültigkeitbereichs
  - Nicht standardisiert, wird aber von GCC und Clang unterstützt

Cleanup v3

```c
#include <glib.h>
#include <stdlib.h>

void foo (char* bar) {
    g_autofree char* foo = malloc(1024);
    if (bar == NULL)
        return;
}

int main (void) {
    foo(NULL);
    return 0;
}
```

- GLib macht die Benutzung komfortabler
  - Außerdem weitere `g_auto`-Varianten für beliebige Datentypen

Threads v0

```c
#include <stdio.h>
#include <threads.h>

int foo = 0;
int thrd (void* bar) { for (int i = 0; i < 10000; i++) foo++; }

int main (void) {
    thrd_t thrds[10];
    for (int i = 0; i < 10; i++) thrd_create(&thrds[i], thrd, NULL);
    for (int i = 0; i < 10; i++) thrd_join(thrds[i], NULL);
    printf("%d\n", foo);
    return 0;
}
```

- Threads erlauben die parallele Abarbeitung von Aufgaben
  - Sowohl zur Leistungssteigerung als auch zur Programmstrukturierung
- In diesem Fall unkoordinierter Zugriff auf gemeinsame Variable

Threads v1

```c
#include <stdatomic.h>
#include <stdio.h>
#include <threads.h>

atomic_int foo = 0;
int thrd (void* bar) { for (int i = 0; i < 10000; i++) foo++; }

int main (void) {
    thrd_t thrds[10];
    for (int i = 0; i < 10; i++) thrd_create(&thrds[i], thrd, NULL);
    for (int i = 0; i < 10; i++) thrd_join(thrds[i], NULL);
    printf("%d\n", foo);
    return 0;
}
```

- Atomare Variablen ermöglichen koordinierten Zugriff mit geringem Overhead
  - Bisher nur mit speziellen Funktionen möglich

- Einige Funktionen sind unbequem zu benutzen
  - Z. B. das automatische Aufräumen von Variablen
- Es ist schwierig portablen Code zu schreiben
  - Eigenheiten von Linux, macOS und Windows
- Selbst implementierte Datentypen und Algorithmen sind oft fehleranfällig
  - Z. B. verkettete Listen, Bäume etc.
- GLib ist eine häufig genutzte C-Bibliothek
  - *GLib is a general-purpose utility library, which provides many **useful data types, macros, type conversions, string utilities, file utilities, a mainloop abstraction, and so on**. It works on many UNIX-like platforms, as well as Windows and OS X. GLib is released under the GNU Lesser General Public License (GNU LGPL).* [1]

GLib – Strings

```c
#include <glib.h>

int main (void) {
    g_autoptr(GString) string = g_string_new(" ");
    g_string_prepend(string, "Hallo");
    g_string_append(string, "Welt!");
    g_print("%s\n", string->str);
    return 0;
}
```

- `GString`-Datentyp erlaubt das komfortable Arbeiten mit Strings
  - Speicher wird transparent durch GLib verwaltet
  - Zugriff auf C-String weiterhin möglich

GLib – Reference Counting

```c
#include <glib.h>

void bar (int* foo) {
    g_rc_box_acquire(foo);
    *foo += 42;
    g_rc_box_release(foo);
}

int main (void) {
    int* foo = g_rc_box_new0(int);
    bar(foo);
    g_rc_box_release(foo);
    return 0;
}
```

- Erlaubt es Datentypen nachträglich mit Reference Counting auszustatten
  - Native Unterstützung mit `grefcount` und `gatomicrefcount`

GLib – Fehlerbehandlung

```c
#include <glib.h>

void foo (GError** error) {
    g_return_if_fail(error == NULL || *error == NULL);
    g_set_error(error, 1, 0, "Adresse: %p", error);
}

int main (void) {
    GError* error = NULL;
    foo(&error);
    if (error != NULL)
        g_print("Fehler: %s\n", error->message);
    return 0;
}
```

- `GError` stellt ein Framework zur Fehlerbehandlung bereit
  - Kontrolle über Rückgabe liegt bei Aufrufendem

Tooling, Tests, Logging

- GLib bietet außerdem eine Vielzahl weiterer Datentypen und Algorithmen
  - Threads, Thread-Pools, Allokatoren etc.
  - String-, Zeit-, Datei- und Parser-Funktionen
  - Listen, Bäume, Queues, Arrays etc.
- Zusätzliche Infrastruktur
  - Test- und Logging-Frameworks

Zusammenfassung

- Neuere C-Standards bieten moderne Funktionalität
  - Threads, atomare Datentypen, Unicode, statische Asserts etc.
- GLib erleichtert die Nutzung von C
  - Portabilität, Komfort und Sicherheit

Quellen

[1] The GNOME Project. **GLib Reference Manual.**
https://developer.gnome.org/glib/.

[2] Wikipedia. **C (Programmiersprache).**
https://de.wikipedia.org/wiki/C_(Programmiersprache).

Makefile

```
1 CFLAGS = -fsanitize=address $(shell pkg-config --cflags glib-2.0) -pthread
2 LDFLAGS = -fsanitize=address $(shell pkg-config --libs glib-2.0) -pthread
```