

# Bibliotheken erstellen

# Ausblick

Heute beschäftigen wir uns mit *Bibliotheken*. Insbesondere ...

- ▶ damit, was der Unterschied zwischen *statischen* und *dynamischen* Bibliotheken ist
- ▶ wie man eigene Bibliotheken beider Arten erstellt
- ▶ wie man eine saubere API definiert
- ▶ und wie man gegen sie linkt

# Begriffsdefinition

Was ist eine Bibliothek?

# Begriffsdefinition

Was ist eine Bibliothek?

*Statische Bibliothek*: Sammlung an Objektdateien

# Begriffsdefinition

Was ist eine Bibliothek?

*Statische Bibliothek*: Sammlung an Objektdateien

*Dynamische Bibliothek*: Vorgelinkter Programmteil, der zur Laufzeit mit dem Hauptprogramm verbunden wird

## Das Werkzeug `ar(1)`

Statische Bibliotheken (Dateiendung `.a`) sind Archive und werden mit `ar` erstellt.

Benutzung: `ar [keys] libfoo.a [obj.o ...]`

Schlüssel:

- `c` Archiv erstelle, falls nicht vorhanden (**c**reate)
- `d` Dateien löschen (**d**elete)
- `r` Dateien hinzufügen/ersetzen (**r**eplace)
- `s` Index erstellen (**s**ymbol table)
- `t` Archivinhalt ausgeben (**t**able)
- `x` Datei extrahieren (**e**xtract)

# Das Werkzeug ar(1)

Übliche Benutzung:

```
# Quelltext in Objekte übersetzen  
cc -c foo.c bar.c baz.c
```

```
# daraus libfoo.a bauen  
ar rcs libfoo.a foo.o bar.o baz.o
```

# Arbeiten mit statischen Bibliotheken

Beim linken:

- ▶ Der Operand `-lfoo` linkt mit der Bibliothek `libfoo.a`
- ▶ diese wird im *Linkpfad* gesucht
- ▶ Elemente können zum Linkpad mit `-L...` hinzugefügt werden
- ▶ aus einer Bibliothek wird nur genommen, was der Linker gerade braucht
- ▶ also: auf Reihenfolge achten!



# Dynamische Bibliotheken: Einführung

Was ist der Unterschied?

- ▶ für das Hauptprogramm fühlen sich dynamische Bibliotheken wie statische an
- ▶ dynamische Bibliotheken werden erst zur Laufzeit an das Programm gebunden
- ▶ sparen dadurch Speicherplatz (nur eine Kopie, auch wenn von 100 Programmen genutzt)
- ▶ und können unabhängig Aktualisierungen erhalten

# Dynamische Bibliotheken erstellen

Der Prozess ist etwas involvierter als bei statischen Bibliotheken:

- ▶ alle involvierten Objekte mit `-fpic` erzeugen
- ▶ die Objekte der Bibliothek mit `-shared` linken (keine `main`-Funktion nötig)
- ▶ Ausgabedatei `libfoo.so` nennen (auf macOS anders)
- ▶ Ggf. `-Wl,-soname=...` setzen
- ▶ mit *Versionsskripten* kann bestimmt werden, welche Symbole exportiert werden

# Arbeiten mit dynamischen Bibliotheken

So ziemlich wie bei statischen Bibliotheken:

- ▶ Der Operand `-lfoo` linkt mit der Bibliothek `libfoo.so`
- ▶ diese wird im *Linkpfad* gesucht
- ▶ Elemente können zum Linkpad mit `-L...` hinzugefügt werden
- ▶ der *Laufzeitpfad* kann mit `-Wl,-rpath=...` gesetzt werden
- ▶ dieser bestimmt, wo zur Laufzeit nach Bibliotheken gesucht wird
- ▶ nützlich, wenn Bibliotheken nicht in Standardpfaden sind

## Installation von Bibliotheken und Programmen

Lasse den Nutzer einen Präfix `PREFIX` wählen (standardmäßig `/usr/local`). Dann installieren in folgende Verzeichnisse unter diesem:

`bin` für den Nutzer ausführbare Programme

`lib` statische und dynamische Bibliotheken

`libexec` interne Programme, die der Nutzer nicht sehen soll

`include` Header-Dateien zu den Bibliotheken

`share` sonstige Dateien, die euer Code zur Laufzeit braucht

`share/doc` Dokumentation

`share/man` Handbuchseiten

→ sie **hier**(7) für Details.

# API-Design

# API-Design

Wie baut ihr eine gut benutzbare Bibliothek?

## Beispiel I (schlecht)

```
1 // Zerlegt str an Vorkommnissen der Zeichen in sep
2 // 1. Aufruf: str != NULL, weitere Aufrufe: str == NULL
3 // Funktion gibt nacheinander Token zurück, Separator
4 // wird eliminiert. NULL wenn keine Token mehr da.
5 #include <string.h>
6
7 char *strtok(char *str, const char *sep);
```

**schlecht:** Funktion hat Zustand, der intern gespeichert wird (nicht wiedereintrittsfähig [engl. reentrant])

## Beispiel I (gut)

```
1 // Zerlegt str an Vorkommnissen der Zeichen in sep
2 // 1. Aufruf: str != NULL, weitere Aufrufe: str == NULL
3 // Funktion gibt nacheinander Token zurück, Separator
4 // wird eliminiert. NULL wenn keine Token mehr da.
5 // last muss auf Variable vom Typ char * zeigen.
6 #include <string.h>
7
8 char *strtok_r(char *str, const char *sep, char **last);
```

**gut:** Speicher für Zustand wird vom Nutzer bereitgestellt.



## Beispiel II (schlecht)

```
1 // Angegebene URL wird in angegebene Datei heruntergeladen.
2 // Ein Fortschrittsbalken wird auf der Konsole angezeigt.
3 // Rückgabewert: Dateigrösse bei Erfolg, sonst negativer
4 // Fehlercode.
5
6 ssize_t get_file(const char *file, const char *url);
```

**schlecht:** Fortschrittsbalken macht die Funktion ungeeignet für komplexe Programme mit eigener Ein- und Ausgabebehandlung / GUI.

## Beispiel II (gut)

```
1 // Angegebene URL wird in angegebene Datei heruntergeladen.
2 // progress() wird in regelmäßigen Abständen aufgerufen.
3 // Rückgabewert: Dateigrösse bei Erfolg, sonst negativer
4 // Fehlercode.
5
6 typedef void progress_func(const char *file, const char *url,
7     size_t length, size_t downloaded);
8 ssize_t get_file(const char *file, const char *url,
9     progress_func *progress);
```

**gut:** Fortschritt wird über Callback angezeigt, Nutzer hat volle Kontrolle.

## Beispiel III (schlecht)

```
1 // Die GUI wird gestartet.  
2 // Wenn das nicht klappt, wird eine Fehlermeldung angezeigt  
3 // und das Programm beendet  
4  
5 void start_gui(void);
```

**schlecht:** Keine Möglichkeit für den Aufrufer, den Fall, dass die GUI nicht geht (z. B. bei Aufruf per SSH) ordentlich zu behandeln (z. B. durch Terminal-Modus). Keine Möglichkeit, eine eigene Fehlermeldung zu produzieren.

## Beispiel III (gut)

```
1 // Die GUI wird gestartet.  
2 // Bei Erfolg wird 0 zurueckgegeben.  
3 // Wenn das nicht klappt, wird ein Fehlercode zurückgegeben.  
4  
5 int start_gui(void);
```

**gut:** Aufrufer hat Kontrolle über die Fehlerbehandlung.

## Beispiel IV (schlecht)

```
1 // Starte einen neuen Prozess aus path. Schreibe PID nach
2 // *pid. Wende file_actions an. Wende Attribute aus attrp
3 // an. Uebergebe Argumente aus argv und envp.
4 #include <spawn.h>
5
6 int
7 posix_spawn(pid_t *restrict pid, const char *restrict path,
8             const posix_spawn_file_actions_t *file_actions,
9             const posix_spawnattr_t *restrict attrp,
10            char *const argv[restrict], char *const envp[restrict]);
```

**schlecht:** stark überfrachtete Funktion, die alles auf einmal machen will und dadurch sehr kompliziert ist.

## Beispiel IV (gut)

```
1 #include <unistd.h>
2
3 // Klone den aktuellen Prozess in einen neuen Kindprozess.
4 pid_t fork(void);
5
6 // Ersetze den aktuellen Prozess durch das Programm path.
7 // Uebergebe Argumente argvp und Umgebung envp.
8 int execve(const char *path, char *const argvp[],
9            char *const envp[]);
```

**gut:** die Komplexität wurde auf mehrere einfache Funktionen heruntergebrochen. Jede Funktion erfüllt genau eine Aufgabe und ist einfach zu verstehen.

## Beispiel V (schlecht)

```
1 #include <crazyimg.h>
2
3 czing *open_crazy_image(const char *path);
4 void    czing_close(czing *image);
5 int     canvas_width(czing *image);
6 int     canvas_height(czing *image);
```

**schlecht:** inkonsistentes Namensschema, Vermüllung des globalen Namensraumes.

## Beispiel V (gut)

```
1 #include <crazyimg.h>
2
3 czing *czing_open(const char *path);
4 void    czing_close(czing *image);
5 int     czing_width(czing *image);
6 int     czing_height(czing *image);
```

**gut:** konsistentes Namensschema, einheitlicher Namensraum durch Präfix.



# API-Design

Wie baut ihr eine gut benutzbare Bibliothek?

- ▶ gute Dokumentation
- ▶ entscheidet bewusst, was ihr exportiert und was nicht
- ▶ benutzt einen konsistenten Präfix für eure Funktionen
- ▶ alles was ihr exportiert, müsst ihr auch in Zukunft unterstützen
- ▶ macht gute Fehlerbehandlung
- ▶ Fehler dem Aufrufer berichten, kein Spam auf die Konsole!
- ▶ bei Updates: bleibt kompatibel zu euch selbst!
- ▶ im Zweifel: einfach denken!
- ▶ je weniger Komplexität, desto einfacher ist die Bibliothek zu nutzen