Freie Universität Berlin

Objektorientiertes Programmieren in C

Ausblick

Heuter lernen wir Techniken der objektorientierten Programmierung in C.
Insbesondere, ...

- wie man *dynamische Bindung* mit *vtables* implementiert
- wie man daraus *Klassen* baut
- wie Klassen voneinander erben können

Die Inhalte dieser Vorlesung sind angelehnt an A.-T. Schreiner: *Objekt-orientierte Programmierung mit ANSI-C*, 1994, Hanser, München.

Was ist ein Objekt?

Was ist ein Objekt?

Der C-Standard sagt (ISO/IEC 9899:2024 § 3.18):

> *object – region of data storage in the execution environment, the contents of which can represent values. […] When referenced, an object can be interpreted as having a particular type […].*

Freie Universität Berlin

Was ist ein Objekt?

Der C-Standard sagt (ISO/IEC 9899:2024 § 3.18):

> *object – region of data storage in the execution environment, the contents of which can represent values. [...] When referenced, an object can be interpreted as having a particular type [...].*

Hat nicht viel mit einem Objekt aus OOP zu tun!
Wollen wir OOP-Objekte, müssen wir diese selber bauen.

Was ist ein Objekt?

Objekte in OOP...

- sind Mitglieder von Klassen
- haben Methoden
- haben Konstruktoren und Destruktoren
- können Eigenschaften Erben
- können Schnittstellen erfüllen

Was ist ein Objekt?

Objekte in OOP...

- sind Mitglieder von Klassen
- haben Methoden
- haben Konstruktoren und Destruktoren
- können Eigenschaften Erben
- können Schnittstellen erfüllen

Wollen wir solche Objekte, müssen wir sie uns selber bauen.

Freie Universität Berlin

Dynamische Bindung

Ein `FILE` kann lesen (`fread`), schreiben (`fwrite`), spulen (`fseek`), und geschlossen werden (`fclose`). Aber je nach Herkunft, sind die Operationen verschieden implementiert (`fopen` vs. `open_memstream` vs. `fmemopen`). Wie funktioniert das?

Dynamische Bindung

Ein `FILE` kann lesen (`fread`), schreiben (`fwrite`), spulen (`fseek`), und geschlossen werden (`fclose`). Aber je nach Herkunft, sind die Operationen verschieden implementiert (`fopen` vs. `open_memstream` vs. `fmemopen`). Wie funktioniert das?

- die `FILE`-Struktur speichert Zeiger auf Funktionen, die diese Operationen für dieses `FILE` implementieren.
- die Zeiger werden beim Öffnen der Datei gesetzt
- `fread`, `fwrite`, … rufen dann einfach die Zeiger auf
- ist ein Zeiger `NULL`, so ist die Operation nicht unterstützt

Freie Universität Berlin

Dynamische Bindung

```c
#include <stdio.h>

FILE *funopen(const void *cookie,
    int (*readfn)(void *cookie, char *buf, int len),
    int (*writefn)(void *cookie, const char *buf, int len),
    int (*seekfn)(void *cookie, fpos_t pos, int whence),
    int (*closefn)(void *cookie));
```

Erzeugt ein `FILE`, dessen Operationen durch die angegebenen Funktionen implementiert sind. Das `cookie` hält nutzerdefinierte Daten.
Achtung: `funopen()` gibt es nicht auf Linux; dort stattdessen `fopencookie()` nutzen.

Freie Universität Berlin

Dynamische Bindung

```
static int dummy_read(void *cookie, char *buf, int len) {
    return (0);
}

static int dummy_write(void *cookie,
                       const char *buf, int len) {
    return (len);
}

FILE *devnull =
    funopen(NULL, dummy_read, dummy_write, NULL, NULL);
```

Mit `devnull` ließt man nichts und schreibt ins nichts. Wie bei /dev/null.

Tabellen virtueller Methoden

Soweit ganz gut! Aber: die Zeiger wollen wir nicht in jedem Objekt mitschleifen.

Tabellen virtueller Methoden

Soweit ganz gut! Aber: die Zeiger wollen wir nicht in jedem Objekt mitschleifen.

- Idee: Funktionszeiger kommen in eine separate Struktur (*vtable*)
- unser Objekt zeigt auf diese Struktur
- die vtable braucht es nur einmal
- spart Platz, wenn man viele Objekte mit den gleichen Funktionen hat.

Tabellen virtueller Methoden

```c
struct fileops {
    int (*readfn)(void *cookie, char *buf, int len);
    int (*writefn)(void *cookie, const char *buf, int len);
    int (*seekfn)(void *cookie, fpos_t pos, int whence);
    int (*closefn)(void *cookie);
};

struct FILE {
    struct fileops *ops;
    ...
};
```

So könnte man es machen.

Von der vtable zur Klasse

Was fehlt noch zur Klasse?

Von der vtable zur Klasse

Was fehlt noch zur Klasse?

- ein *Konstruktor* zur Erzeugung eines Objektes
- ein *Destruktor* zur Freigabe eines Objektes
- eine Möglichkeit, Objekte zu *klonen*
- eine Möglichkeit, die Größe eines Objektes zu bestimmen

Von der vtable zur Klasse

```
1 struct class {
2     size_t size;
3     void * (*ctor)(void *self, va_list ap);
4     void * (*dtor)(void *self);
5     void * (*clone)(const void *self);
6 };
```

Minimale Funktionen, die eine Klasse benötigt. Weitere Funktionen können am Ende eingebttet werden (dazu später mehr).

Objekte allozieren

```
1  void *new(const struct class *class, ...) {
2      void *obj = calloc(1, class->size);
3  
4      *(const struct class **)obj = class;
5      if (class->ctor) {
6          va_list ap;
7          va_start(ap, class);
8          obj = class->ctor(obj, ap);
9          va_end(ap);
10     }
11 
12     return (obj);
13 }
```

Freie Universität Berlin

Objekte freigeben

```c
void delete(void *self) {
    const struct class **cp = self;

    if (self != NULL && *cp != NULL && (*cp)->dtor != NULL)
        self = (*cp)->dtor(self);

    free(self);
}
```

Beispiel: Leinwand

```c
struct canvas {
    struct class *vtable;
    size_t x, y;
    int *pixels;
};

struct class Canvas = {
    .size = sizeof (struct canvas),
    .ctor = canvas_ctor,
    .dtor = canvas_dtor,
    .clone = canvas_clone,
};
```

Beispiel: Leinwand

```
1 void *canvas_ctor(void *selfp, va_list ap) {
2     struct canvas *self = selfp;
3
4     self->x = va_arg(ap, size_t);
5     self->y = va_arg(ap, size_t);
6     self->pixels = calloc(self->x * self->y,
7         sizeof *self->pixels);
8
9     return (self);
10 }
```

Beispiel: Leinwand

```c
void *canvas_dtor(void *selfp) {
    struct canvas *self = selfp;
    free(self->pixels);
    return (self);
}

void *canvas_clone(const void *selfp) {
    const struct canvas *self = selfp;
    struct canvas *clone = new(Canvas, self->x, self->y);
    memcpy(clone->pixels, self->pixels,
        self->x * self->y * sizeof *self->pixels);
    return (clone);
}
```

Dynamische Bindung mit Klassen

Wir wissen jetzt, wie man Vtables und Klassen baut.
Wie baut man jetzt Klassen, die dynamische Bindung haben?

- wir lassen die vtable mit der Klasse beginnen
- Zeiger auf weitere Methoden finden sich hinter `clone`
- `new` und `delete` müssen davon nichts wissen
- alle Klassen mit derselben vtable implementieren die selben Methoden (also die selbe *Schnittstelle*)

Dynamische Bindung mit Klassen

```
1 struct File_class {
2     struct class cl;
3     int (*readfn)(void *self, char *buf, int len);
4     int (*writefn)(void *self, const char *buf, int len);
5     int (*seekfn)(void *self, fpos_t pos, int whence);
6     int (*closefn)(void *self);
7 };
```

Freie Universität Berlin

Dynamische Bindung mit Klassen

```
1 struct devnull {
2     struct File_class *class;
3 };
4
5 struct File_class Devnull = {
6     .class = { .size = sizeof (struct devnull) },
7     .readfn = dummy_read,
8     .writefn = dummy_write,
9 };
```

Dynamische Bindung mit Klassen

```
1  struct realfile {
2      struct File_class *class;
3      int fd, buf_fill;
4      char buf[BUFSIZE];
5  };
6
7  struct File_class Realfile = {
8      .class = { .size = sizeof (struct realfile),
9          .ctor = realfile_ctor, .dtor = realfile_dtor },
10     .readfn = realfile_read, .writefn = realfile_write,
11     .seekfn = realfile_seek, .closefn = realfile_close
12 };
```

Dynamische Bindung mit Klassen

```
1 int file_read(void *selfp, char *buf, int len)
2 {
3     struct File_class **self = selfp;
4 
5     if ((*self) == NULL || (*self)->readfn == NULL)
6         return (-1);
7 
8     return (*self)->readfn(self, buf, len);
9 }
```

Vererbung

Wie können wir aus einer Klasse eine neue ableiten?

- wir kopieren die Struktur der Basisklasse mit adjustierter Größe
- und fügen an die Objekt-Struktur neue Felder an
- einzelne Methoden können überschrieben werden
- oder wir übernehmen sie von der Basisklasse
- neue Methoden können hinten angehangen werden

Vererbung

```c
1 struct TextFile_class {
2     struct File_class super;
3     int (*getsfn)(void *self, char *buf, int len);
4     int (*putsfn)(void *self, char *str);
5 };
6
7 struct textfile {
8     struct realfile super;
9 };
```

Vererbung

```c
struct TextFile_class Textfile = {
    .super = {
        .class = { .size = sizeof (struct textfile),
        .ctor = textfile_ctor, .dtor = textfile_dtor },
        .readfn = realfile_read, .writefn = realfile_write,
        .seekfn = realfile_seek, .closefn = realfile_close,
    },
    .getsfn = textfile_getsfn,
    .putsfn = textfile_putsfn,
};
```

Freie Universität Berlin

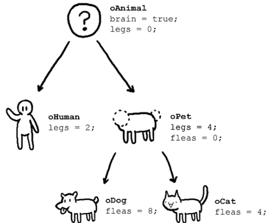
Weitere Themen

Viele weitere Dinge sind möglich:

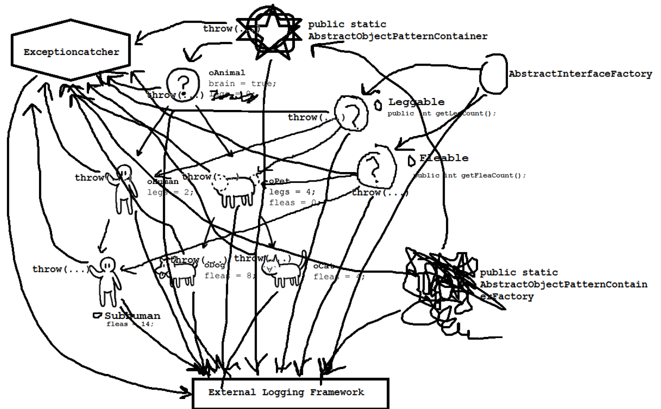
- Unterscheidung von Objekten nach Klasse zur Laufzeit
- Mehrfachvererbung
- Delegates
- Ausnahmen
- ... vieles mehr

Bei Interesse: lest das Buch!

What OOP users claim



What actually happens



OOP in der Praxis

Klassisches OOP ist super hilfreich bei GUI und Aktorenprogrammierung.

OOP in der Praxis

Klassisches OOP ist super hilfreich bei GUI und Aktorenprogrammierung.
Anderswo ist es aber oft Overkill.

- oft reichen einfache Funktionszeiger für dynamische Bindung aus
- wo das nicht reicht, helfen oft einfache vtables
- verwendet ggf. eine objektorientierte C-Variante (C++, Objective C, Vala)
- OOP nicht übertreiben, sondern so nutzen, dass es den Code einfacher macht!
- Komplexität ist der Feind!