Objektorientierte Programmierung mit C

Eugen Betke

2019-01-14

Inhaltsverzeichnis

1 Einleitung

Obwohl C keine objektorientierte Programmierung unterstützt, kann man mit der Sprache den Konzept der objektorientierten Programmierung umsetzen. Das erfordert allerdings ein sehr viel Disziplin, weil C-Sprache den Programmierer kaum dabei unterstützt. Sie eschwert es sogar, indem sie viele Umsetzungsmöglichkeiten bietet. In dieser Ausarbeitung behandeln nur eine Möglichkeit von Vielen objektorientiert zu Programmieren. Unsere Absicht ist möglichst nicht über den C-Standard hinausgehen, d.h. ohne Verwendung von exotischen Kompilererweiterungen und mit minimalen Einsatz von Präprozessor.

1.1 Debugging

Um die Funktionsweise zu erläutern schleusen wir Debugging-Code in die Programme. Es handelt sich dabei um die Ausgabe des Funktionsnamen, der Zeile und Kommentar. Das Macro ist dargestellt in Listing 1

Source Code 1: Debugging Macro

```
#define DEBUG

#ifdef DEBUG
#define DEBUGMSG(...) \
{\
printf("[DEBUG] %s:%d - ", __PRETTY_FUNCTION__, __LINE__); \
printf(__VA_ARGS__); \
printf("\n"); \
}
#else
#define DEBUGMSG(...)
#endif
```

1.2 Kompilierung

Die anonymen Strukturen haben sich als esseziell für die virtuellen Strukturen erwiesen. Um sie zu aktivieren, verwenden wir bei der Kompilierung den folgenden Kompiler-Flag.

```
gcc -fms-extensions
```

2 Einfache Klassen

Eine Klasse definiert einen Bauplan für einen neuen Datentypen. Allerdings, bietet uns die C-Programmiersprache keine echten Klassen an. Das Nächste, was Klassen nahe kommt sind Strukturen. Sie spielen deshalb eine zentrale Rolle bei der Umsetzung des OOP Konzeptes.

2.1 Klassendeklaration

In C werden, bis auf einige Ausnahmen, die Variablen, insbesondere Strukturen, nicht automatisch initialisiert. Nach Instanziierung der Strukturen werden wir deswegen üblicherweise zufällige Werte in den Membervariablen vorfinden. Es ist die Aufgabe der Programmierer die Instanzen in einen Konsistenten zustand zu bringen. Desweiteren, werden wir in statt Strukturen, von den Klassen sprechen. Die Instanzen der Strukturen nennen wir Objekte.

Bevor wir mit dem Konzept fortfahren, wollen wir einen generellen Blick auf die OOP in den anderen Programmiersprachen werfen. Die nativen OOP Programmiersparachen wie C++ und Java erzeugen automatisch eine Reihe von Default-Funktionen, die bestimmte Operationen auf Objekten ausführen, wenn der Programmier sie selber nicht erzeugt. Zu einem sind es Konstruktoren, die dafür zuständig sind den konsisteten Zustand von Objekten sicherzustellen. Jede Klasse benötigt mindestens einen Konstruktor. Abhängig von dem Funktionalität, kann eine Klasse mehrere Konstruktoren haben. Weitere wichtige Funktionen sind Kopierkonstruktor, Zuweisungsoperator und Destruktor. Um sauber das OOP-Konzept umzusetzen, benötigen wir auch diese Funktionen. Da C-Programmiersprache sie nicht automatisch erzeugt, muss der Programmier sich darum kümmern.

- Kopierkonstruktor
  - Ein Kopierkonstruktor wird auf uninitialisierte Objekte angewendet. Er weist eine Kopie des Quellobjektes einem uninitialisierten Zielobjekt zu.

```
someclass_t target;
someclass_t source;
someclass_constructor(&target, /* init params */)
someclass_copy(&target, &source);
/* ... */
```

- Zuweisungsoperator
  - Ein Zuweisungsoperator wird auf bereits initializierte Objekte angewendet. Das Zielobjekt wird zuerst bereinigt. Danach wird eine Kopie aus dem Quellobjekt erzeugt und dem Zielobjekt zugewiesen. So entstehen keine Speicherlecks.

```
someclass_t target;
someclass_t source;
someclass_constructor(&target, /* init params */)
someclass_constructor(&source, /* init params */)
someclass_assign(&target, &source);
/* ... */
```

- Destruktor
  - Ruf Destruktoren von allen Membervariablen des Objektes.

Der Destruktor ist ein besonders wichtiger Bestandteil von OOP. Insbesondere, weil auf dem Heap allozierter Speicher nach der Verwendung wieder freigegben werden muss, um Speicherlecks zu vermeiden. Für diesen Zwecke muss zu einem geeigneten Zeitpunkt der Destruktor aufgerufen werden. In C muss Destruktor manuell aufgerufen werden, da C uns keine Hilfmittel für die Automatisierung zur Verfügung stellt, z.B. kein Garbage-Kollektor oder Aufruf vom Destruktor beim Verlassen des Gültigekeitsbereiches des Objektes.

In unseren fiktiven Beispiel wollen wir den Vornamen, den Namen und den Job, der die Angestellten einer Firma ausüben speichern. In Listing 2 beinhaltet die Strukture `employee_t` die Membervariablen. Weiter unten werden der Konstruktor, Destruktor, Kopierkonstruktor und der Zuweisungsoperator deklariert. Zusätzlich deklarieren wir eine `print` Memberfunktion.

Source Code 2: Einfache Klassen: employees.h

```c
#ifndef  employee_INC
#define  employee_INC

/* class */
typedef struct employee_t {
  /* member variables */
  char *firstname;
  char *lastname;
  char *job;
} employee_t;

/* constructor and "rule of three" functions */
void employee_constructor(employee_t *this, char *firstname, char *lastname, char *job);
void employee_destructor(employee_t *this);
void employee_copy(employee_t *this, employee_t *source);
void employee_assign(employee_t *this, employee_t *source);

/* member functions */
void employee_print(employee_t *this);

#endif   /* ----- #ifndef employee_INC  ----- */
```

2.2 Klassenfunktionen

In C bietet es sich an die Implementierung der Funktionen in separate Source-Dateien auszulagern. Wie die Implementierung aussehen kann zeigt Listing 3.

Source Code 3: Einfache Klassen: employees.c

```c
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#include "debug.h"
#include "employee.h"

void employee_constructor(employee_t *this, char *firstname, char *lastname, char* job) {
  DEBUGMSG("");
  this->firstname = strdup(firstname);
  this->lastname = strdup(lastname);
  this->job = strdup(job);
}

void employee_destructor(employee_t *this) {
  DEBUGMSG("");
  free(this->firstname);
  free(this->lastname);
  free(this->job);
}

void employee_copy(employee_t *this, employee_t *source) {
  DEBUGMSG("");
  this->firstname = strdup(source->firstname);
  this->lastname = strdup(source->lastname);
  this->job = strdup(source->job);
}

void employee_assign(employee_t *this, employee_t *source) {
  DEBUGMSG("");
  free(this->firstname);
  free(this->lastname);
  free(this->job);
  this->firstname = strdup(source->firstname);
  this->lastname = strdup(source->lastname);
```

```
36    this->job = strdup(source->job);
37  }
38
39  void employee_print(employee_t *this) {
40    DEBUGMSG("");
41    printf("Employee: %s %s - %s\n", this->firstname, this->lastname, this->job);
42  }
```

2.3 Nutzung

Bei der Benutzung sollte man immer im Erinnerung behalten, dass alle Funktionen, mindestens ein Konstruktor und Destruktor manuell aufrufen werden müssen. Listing 4 zeigt anhand von Beispielen wie der Konstruktor, Kopierkonstruktor, Zuweisungsoperator und der Destruktor angewendet werden können.

Source Code 4: Einfache Klassen: main.c

```
1   #include "employee.h"
2
3   int main(int argc, char** argv) {
4     /*  constructor */
5     employee_t employee;
6     employee_constructor(&employee, "Ulrike", "Müller", "Designer");
7     employee_print(&employee);
8
9     /* copy constructor */
10    employee_t copy;
11    employee_copy(&copy, &employee);
12    employee_print(&copy);
13
14    /* assignment */
15    employee_t assignment;
16    employee_constructor(&assignment, "Some", "Body", "Employment");
17    employee_assign(&assignment, &employee);
18    employee_print(&assignment);
19
20    /* destructor */
21    employee_destructor(&assignment);
22    employee_destructor(&copy);
23    employee_destructor(&employee);
24  }
```

2.4 Einschränkungen

2.4.1 Zugriffskontrolle

C-Standard bietet kein Hilfsmittel, mit den mit den man den Zugriff auf die Klassenvariablen und Klassenfunktionen beschränken könnte. Alle Variablen und Methoden sind quasi `public`. Nachbildung von `protected` `private` nicht ohne Weiteres möglich.

2.4.2 Konstruktor und Destruktor

Wie bereits diskutiert, ist es leider so, dass beim Erzeugen eines Objektes der passende Konstruktor nicht automatisch aufgerufen wird, wie z.B. bei C++ oder Java. Darum muss sich der Programmier explizit kümmern. Das erfordert zwar viel Disziplin, bedeutet für uns keine Nachteile in der Funktionalität. Bei den Destruktoren müssen wir leider Abstriche machen. C bietet uns weder eine Garbage-Kollektor noch Ruf den Destrutkor automatisch auf, wenn das Programm den Scope verlässt. Konkret für uns heißt es wieder viel Disziplin, aber auch das wir bestimmte Programmierkonzepte wie "RAII"nicht umsetzen können.

3 Vererbung

3.1 C++ als Vorbild

Dieser Abschnitt zeigt einen Beispiel einer Vererbung in C++. In einem Unternehmen werden von den gewöhnlichen Angestellten nur der Vorname und Nachname gespeichert. Es ist möglich den Vornamen und Namen ausgeben zu lassen.

Source Code 5: C++ Beispiel: `Employee` Klasse

```
#ifndef  employee_INC
#define  employee_INC


class Employee {
  protected:
    char* firstname;
    char* lastname;
  public:
    Employee(const char* firstname, const char* lastname);
    ~Employee();
    virtual void print();
};

#endif   /* ----- #ifndef employee_INC  ----- */
```

Ein Manager ist ein Angestellter mit einem bestimmten Level. Ihm kann eine Gruppe mit einer bestimmter Anzahl der Angestellten zugeordnet werden. Es ist möglich alle diese Information über einen Manager anzeigen zu lassen.

Source Code 6: C++ Beispiel: `Manager` Klasse

```
#ifndef  manager_INC
#define  manager_INC

#include "employee.hpp"

class Manager : public Employee {
  private:
    int max_group_size;
    int group_size;
    int level;
    Employee** group;
  public:
    Manager(const char* firstname, const char* lastname, int level);
    ~Manager();
    virtual int add_member(Employee* employee);
    virtual void print();
};

#endif   /* ----- #ifndef manager_INC  ----- */
```

Dieser Zusammenhang ist abgebildet durch die Klassen `Employee` und `Manager`.

Eine mögliche Nutzung ist in Listing 7 dargestellt. Wir haben in den Konstruktor, Destruktor und Memberfunktionen Debugging-Code eingeschleusst um die Aufrufreihenfolge zu verfolgen. Als Kommentar zur jeder Ausgabe steht auch der Vorname und der Nachname der Angestellten, um die Aufrufe zuzuordnen zu können. In der Ausgabe kann man sehen, dass beim Erzeugen des Objektes stets ein passender Konstruktor aufgerufen wird. Am Ende werden die Objekte nach dem LIFO Prizip automatisch zerstört, weil der Gültigkeitsbereich der Objekte, der sich auf die `main` Funktion beschränkt, verlassen wird. Zwischen den Konstruktor- und Destruktoraufrufen sehen wir die Aufrufe der Memberfunktionen und die Ausgabe von `printf`.

Source Code 7: C++ Beispiel: Nutzung

```cpp
#include "employee.hpp"
#include "manager.hpp"

int main(int argc, char** argv) {
  Employee employee1{"Ulrike", "Müller"};
  Employee employee2{"Hans", "Meier"};

  Manager manager{"Matthias", "Gross", 10};
  manager.add_member(&employee1);
  manager.add_member(&employee2);
  Employee* foo = &manager;
  foo->print();
}
```

Source Code 8: C++ Beispiel: Ausgabe

```
[DEBUG] Employee::Employee(const char*, const char*):9 - Matthias Gross
[DEBUG] Manager::Manager(const char*, const char*, int):9 - Matthias Gross
[DEBUG] Employee::Employee(const char*, const char*):9 - Ulrike Müller
[DEBUG] int Manager::add_member(Employee*):22 - Matthias Gross
[DEBUG] Employee::Employee(const char*, const char*):9 - Hans Meier
[DEBUG] int Manager::add_member(Employee*):22 - Matthias Gross
[DEBUG] void Manager::print():34 - Matthias Gross
Manager (10) Matthias Gross
[DEBUG] void Employee::print():21 - Ulrike Müller
Employee: Ulrike Müller
[DEBUG] void Employee::print():21 - Hans Meier
Employee: Hans Meier
[DEBUG] Employee::~Employee():15 - Hans Meier
[DEBUG] Employee::~Employee():15 - Ulrike Müller
[DEBUG] Manager::~Manager():17 -
[DEBUG] Employee::~Employee():15 - Matthias Gross
```

Dieses Verhalten wollten wir in C nachbilden.

3.2 Konzepte

Um die objektorientierte Programmierung in C umzusetzen werden wir uns den Speicherlayout von Strukturen genauer ansehen und den Konzept der virtuellen Tabelle einführen.

3.2.1 Speicherlayout der Strukturen

Die Strukturen in C sind sensitiv zur Reihenfolge der Variablen. Wir nutzen diese Eigenschaft von C, um Vererbung zu implementieren. Das funktioniert folgenderweise. Angenommen wir haben eine Basisklasse `base_t` mit zwei Membervariablen `a` und `b`. Hier gilt erstmal nichts besonders zu beachten. Davon wollen wir nun eine weiter Klasse `derived_t` ableiten, die ebenfalls die Variablen `a` und `b` beinhaltet. Es reicht, wenn wir eine `base_t` Membervariable an erster Stelle plazieren. Danach kommen die anderen Membervariablen. Diese kleine Klassenhierarchie ist in Listing 9 abgebildet. Die Figure 1 zeigt wie die Daten im Speicher abgelegt werden.

Source Code 9: Klassenhierarchie

```c
typedef struct base_t {
  int a;
  int b;
} base_t;

typedef struct derived_t {
  base_t super;
  double a;
  double b;
} derived_t;
```

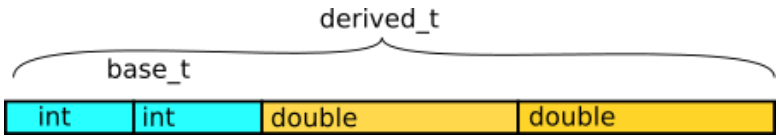

Abbildung 1: Datalayout

Wenn wir solche ein Objekt erstellen, dann können wir den Speicher mit dem Casting unterschiedlich interpretieren und je nachdem auf das Basisobjekt of auf das abgeleitete Objekt zugreifen (Listing 10).

Source Code 10: Up- und Down-Casting

```
derived_t d;
base_t *b1 = (base_t*) &d; // up-casting
base_t *b2 = &d.super; // alternative to up-casting
derived_t *d1 = (derived_t*) b1; // down-casting
derived_t *d2 = (derived_t*) b2; // down-casting
```

3.2.2 Virtuelle Tabelle

Die abgeleitete Klasse kann eine virtuelle Memberfunktion aus der Basisklasse übernehmen, sie reimplementieren oder eine neue hinzufügen. Es können auch weitere nicht virtuelle Funktionen hinzugefügt werden. Alle diese Möglichkeiten müssen wir bei unseren Konzept der virtuellen Tabelle berücksichtigen.

Listing 11 und Listing 12 zeigen nur die nötigsten Elemente für die Implementierung einer virtuellen Tabelle. In der Header-Datei sind es Deklarationen des virtuellen Tabelle, Klasse und Memberfunktionen. In der Source-Datei sind es die Instanziierung zusammen mit der Initializierung der virtuellen Tabelle und die Implementierung der Memberfunktionen.

Die virtuelle Tabelle beinhaltet einen oder mehrere Funktionspointer.

Source Code 11: Basis Klasse

```
/* virtual table (type) */
typedef struct employee_vtbl_t {
  void  (*print)(void *employee);
} employee_vtbl_t;

/*  base class */
typedef struct employee_t {
  employee_vtbl_t *vtbl;
} employee_t;

/* default constructor */
void employee_constructor(void *_this);

/* member function (declaration) */
void employee_print(void *_this);
```

```
/*  virtual table (instantiation + initialization) */
static employee_vtbl_t employee_vtbl = {
  .print = employee_print
};

void employee_constructor(void *_this) {
  ((employee_t*) this)->vtbl = &employee_vtbl;
}

/* member functions (implementation) */
void employee_print(void *_this) {
  /* do something */
}
```

Source Code 12: Abgeleitete Klasse

```c
/* virtual table (declaration) */
typedef struct manager_vtbl_t {
  struct employee_vtbl_t;
  int (*add_member)(void *_this, employee_t *employee);
} manager_vtbl_t;

/* derived class */
typedef struct manager_t {
  employee_t super;
} manager_t;

/* default constructor */
void manager_constructor(void *_this);

/* member functions (declaration) */
int manager_add_member(void *_this, employee_t *employee);
void manager_print(void *_this);
```

```c
/* virtual table (instantiation + initialization) */
static manager_vtbl_t manager_vtbl = {
  .print = manager_print,
  .add_member = manager_add_member
};

/* default constructor */
void manager_constructor(void *_this) {
  ((employee_t*) this)->vtbl = (employee_vtbl_t*) &manager_vtbl;
  /* do something */
}

/* member functions (implementation) */
int manager_add_member(void *_this, employee_t *employee) {
  /* do something */
   return 0;
}

void manager_print(void *_this) {
  /* do something */
}
```

Der native Zugriff auf die virtuelle Tabelle kann etwas umständlich sein, wenn man nur die C-Sprache verwendet. Das kann in drei Schritten gemacht werden.

1. Up-Casting vom abgeleiteten Objekten auf die Basis, um auf die virtuelle Tabelle zuzugreifen.
2. Down-Casting der virtuellen Tabelle auf benötigten Level
3. Nutzung der virtuellen Funktionen

In Listing 13 zeigt wie man von der Basisklasse und von der abgeleiteten Klasse auf die virtuelle Tabelle in zugreift.

Source Code 13: Zugriff auf die virtuellen Funktionen mit Makros

```c
/*  create and construct employee and manager */
// direct access of virtual table in base object
employee.vtbl->print(&employee);

// explicit access of virtual table in derived object
employee_t* base = (employee_t*) &manager; // 1. up-casting derived object
```

```
manager_vtbl_t* vtbl = (manager_vtbl_t*) base->vtbl; // 2. down-casting virtual table
vtbl->add_member(&manager, &employee); // 3. using a virtual function
vtbl->print(&manager); // 3. using another virtual function

// short version
((manager_vtbl_t*) ((employee_t*) &manager)->vtbl)->add_member(&manager, &employee);
((manager_vtbl_t*) ((employee_t*) &manager)->vtbl)->print(&manager);
/* destroy manager and employee*/
```

Dieses umständliche Zugreifen kann mit Makros versteckt werden (Listing 14). Der Zugriff kann sogar so maskiert werden, dass der Eindruck entsteht, dass man eine gewöhnliche Funktion benutzt.

Source Code 14: Zugriff auf die virtuellen Funktionen mit Makros

```
#define M_employee_print(me) ((employee_t*)(me))->vtbl->print(me)
#define M_manager_add_member(me, employee) \
  ( ((manager_vtbl_t*) ((employee_t*) me)->vtbl)->add_member((me), (employee)))
#define M_manager_print(me) ( ((manager_vtbl_t*) ((employee_t*) me)->vtbl)->print(me))

/*  create and construct employee and manager */
M_employee_print(&employee);
M_manager_add_member(&manager, &employee);
M_manager_print(&manager);
/* destroy manager and employee*/
```

Man kann natürlich auch die `super`-Membervariable benutzen, um ein Up-Casting zu ersparen (Listing 15).

Source Code 15: Zugriff auf die virtuellen Tabelle über die `super`-Variable

```
/*  create and construct employee and manager */
employee.vtbl->print(&employee);
((manager_vtbl_t*) manager.super.vtbl)->add_member(&manager, &employee);
((manager_vtbl_t*) manager.super.vtbl)->print(&manager);
/* destroy manager and employee*/
```

3.3 Umsetzung in C

Den Preis, den wir bei jedem Objekt für eine virtuelle Tabelle bezahlen, ist typischerweise der Speicher für einen Zeiger (8 bytes in der 64bit-Architektur). Das ist implementierungsabhängig und kann je nach Implementierung abweichen. Der C++-Beispiel aus der vorherigen Abschnitt wird im unteren Abschnitt umgesetzt. Als erstes erstellen wir eine virtuelle Tabelle `employee_vtbl_t`. Sie beinhaltet eine einzige Funktion `print`.

Dann erstellen wir eine Klasse `employee_t`. Neben den Membervariablen, beinhaltet die Klasse auch einen Zeiger auf die virtuelle Tabelle. Dieser Zeiger ist nur in der Basisklasse notwendig und verbraucht deshalb nur eine konstante Menge an Speicher, unabhängig davon wie unsere Klassenhierarchie aussieht.

Achtet darauf, dass die Memberfunktionen statt einen Zeiger auf einen konkreten Typen einen `void*` Zeiger als Parameter bekommen. Damit machen wir zwar die Klasse etwas typ-unsicherer, aber dieses Vorgehen erleichter später die Nutzung der abgeleiteten Klassen.

```
#ifndef  employee_INC
#define  employee_INC

#define M_employee_print(me) ((employee_t*)(me))->vtbl->print(me)

typedef struct employee_vtbl_t {
  void  (*print)(void *_this);
} employee_vtbl_t;

typedef struct employee_t {
  employee_vtbl_t *vtbl;
  char *firstname;
```

```
  char *lastname;
} employee_t;

void employee_constructor(void *_this, char* firstname, char* lastname);
void employee_destructor(void *_this);
void employee_print(void *_this);


#endif   /* ----- #ifndef employee_INC  ----- */
```

Wie in Listing 16 gezeigt, muss die virtuelle Tabelle zuerst einmal im globalen Namespace angelegt werden. An dieser Stelle werden die Funktionen den Funktionszeiger zugewiesen.

Im Konstruktor casten wir als erstes den `void*` auf den richtigen Typen und haben somit einen bequemen Zugang auf die Membervariablen. Dann weisen wir den Zeiger auf die virtuelle Tabelle die im vorherigen Schritt angelegte Tabelle zu. Schließlich, initializieren wir die restlichen Membervariablen.

Source Code 16: Manager-Klasse Deklaration

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

#include "debug.h"
#include "employee.h"

static employee_vtbl_t employee_vtbl = {
  .print = employee_print
};

void employee_constructor(void *_this, char *firstname, char *lastname) {
  employee_t *this = _this;
  DEBUGMSG("%s %s", firstname, lastname);
  this->vtbl = (employee_vtbl_t*) &employee_vtbl;
  this->firstname = strdup(firstname);
  this->lastname = strdup(lastname);
}

void employee_destructor(void *_this) {
  employee_t *this = _this;
  DEBUGMSG("%s %s", this->firstname, this->lastname);
  free(this->firstname);
  free(this->lastname);
}

void employee_print(void *_this) {
  employee_t *this = _this;
  DEBUGMSG("%s %s", this->firstname, this->lastname);
  printf("Employee: %s %s\n", this->firstname, this->lastname);
}
```

Die virtuelle Tabelle `manager_vtbl_t` erbt die `employee_vtbl_t` und erhält somit automatisch die bereits definierten Funktionszeiger. Nun fügen wir einen weiteren Funktionszeiger auf die `add_member()` hinzu. Die `manager_t` Klasse leitet die `employee_t` Klasse ab und erbt somit automatisch den Zeiger auf die virtuelle Tabelle. Zum Schluss deklararieren wir die benötigten Memberfunktionen.

```
#ifndef  manager_INC
#define  manager_INC

#include "employee.h"

#define M_manager_add_member(me, employee) \
  (((manager_vtbl_t*) ((employee_t*) me)->vtbl)->add_member((me), (employee)))
#define M_manager_print(me) ( ((manager_vtbl_t*) ((employee_t*) me)->vtbl)->print(me))
```

```
9
10  typedef struct manager_vtbl_t {
11    struct employee_vtbl_t;
12    int (*add_member)(void *_this, employee_t *employee);
13  } manager_vtbl_t;
14
15  typedef struct manager_t {
16    employee_t super;
17    int level;
18    unsigned int max_group_size;
19    unsigned int group_size;
20    employee_t **group;
21  } manager_t;
22
23  void manager_constructor(void *_this, char *firstname, char *lastname, int level);
24  void manager_destructor(void *_this);
25  int manager_add_member(void *_this, employee_t *employee);
26  void manager_print(void *_this);
27
28
29  #endif   /* ----- #ifndef manager_INC  ----- */
```

Bei der Implementierung (Listing 17) verfahren wir so ähnlich wie bei der Basisklasse.

Source Code 17: Manager-Klasse Implementierung

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   #include "manager.h"
5   #include "debug.h"
6
7   static manager_vtbl_t manager_vtbl = {
8     .print = manager_print,
9     .add_member = manager_add_member
10  };
11
12  void manager_constructor(void *_this, char *firstname, char *lastname, int level) {
13    manager_t *this = _this;
14    employee_constructor(&this->super, firstname, lastname);
15    DEBUGMSG("%s %s", firstname, lastname);
16    this->super.vtbl = (employee_vtbl_t*) &manager_vtbl;
17    this->max_group_size = 10;
18    this->group_size = 0;
19    this->group = malloc(sizeof(*(this->group)) * this->max_group_size);
20    this->level = level;
21  }
22
23  void manager_destructor(void *_this) {
24    manager_t *this = _this;
25    DEBUGMSG("%s %s", this->super.firstname, this->super.lastname);
26    free(this->group);
27    employee_destructor((employee_t*) this);
28  }
29
30  int manager_add_member(void *_this, employee_t *employee) {
31    manager_t *this = _this;
32    DEBUGMSG("%s %s", this->super.firstname, this->super.lastname);
33    if (this->group_size < this->max_group_size) {
34      this->group[this->group_size] = employee;
35      this->group_size +=1;
36      return 0;
37    }
38    else {
39      return 1;
```

```
  }
}

void manager_print(void *_this) {
  manager_t *this = _this;
  DEBUGMSG("%s %s", this->super.firstname, this->super.lastname);
  printf("Manager (%d) %s %s\n", this->level, this->super.firstname, this->super.lastname);
  for (int i = 0; i < this->group_size; i++) {
    printf("\t Group member: %s %s\n", this->group[i]->firstname, this->group[i]->lastname);
  }
}
```

```
#include "employee.h"
#include "manager.h"

int main(int argc, char** argv) {
  /* create employees and manager */

  employee_t* staff[3];
  staff[0] = &employee1;
  staff[1] = &employee2;
  staff[2] = &manager.super;
  for (int i = 0; i < 3; ++i) {
    M_employee_print(staff[i]);
  }

  /* clean up */
}
```

```
Employee: Ulrike Müller
Employee: Hans Meier
Manager (1) Matthias Gross
   Group member: Ulrike Müller
   Group member: Hans Meier
```

4 Abstrakte Klassen

TODO