Objektorientierte Programmierung

Praktikum "C-Programmierung"

Eugen Betke

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

2019-01-21

Gliederung (Agenda)

Table Of Content

Umsetzung

- Die Daten werden in einer Struktur gekapselt
- Funktionen werden extern deklariert und implementiert
- Typischerweise, bekommen die Klassen einen oder mehrere Konstruktoren und genau einen Destruktor

Beispiel einer Klasse

```
1   #ifndef  employee_INC
2   #define  employee_INC
3
4   typedef struct employee_t {
5     char *firstname;
6     char *lastname;
7     char *job;
8   } employee_t;
9
10  void employee_constructor(employee_t *this, char *firstname, char *lastname, char *job);
11  void employee_destructor(employee_t *this);
12  void employee_print(employee_t *this);
13
14  #endif   /* ----- #ifndef employee_INC  ----- */
```

Typische Nutzung einer Klasse

```
1  #include "employee.h"
2
3  int main(int argc, char** argv) {
4    employee_t employee;
5    employee_constructor(&employee, "Ulrike", "Müller", "Designer");
6    employee_print(&employee);
7    employee_destructor(&employee);
8  }
```

Funktionsaufrufe:

```
1  [DEBUG] employee_constructor:9 -
2  [DEBUG] employee_print:23 -
3  Employee: Ulrike Müller - Designer
4  [DEBUG] employee_destructor:16 -
```

- Konstruktor und Destruktor müssen manuell aufgerufen werden
- Die Memberfunktionen können nicht über die Punktnotation aufgerufen werden

Einschränkungen

- Keine Zugriffskontrolle
  - Alle Variablen und Methoden sind `public`
  - Nachbildung von `protected` und `private` nicht ohne weiteres möglich
- Methoden müssen extern deklariert und implementiert werden.
  - Namenskonventionen sind hilfreich
- Manueller Speichermanagement
  - Keine automatischer Konstruktor-Aufruf
  - Es gibt kein Garbage-Collector wie in Java
  - Destruktor wird nicht automatisch aufgerufen wie in C++
- Objektorientierte Programmierung in C ist nicht standardisiert
  - Viele unterschiedliche Implementationen möglich

Table Of Content

Aufbau von Strukturen im Speicher

```
1    typedef struct base_t {
2      int a;
3      int b;
4    } base_t;
```

```
1    typedef struct derived_t {
2      base_t super;
3      double a;
4      double b;
5    } derived_t;
```

```
1    derived_t d;
2    base_t *b1 = (base_t*) &d;
3    base_t *b2 = &d.super;
```

- Die Variablen werden im Speicher in der gleichen Reihenfolgen abgelegt wie im Source Code
- Down-Cast ist möglich, wenn Basisstruktur vorne steht

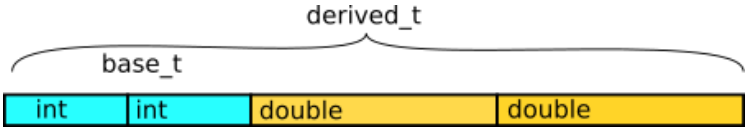


Abbildung: Speicherlayout von `derived_t`

Namenskonflikte

Basisklasse steht auf der ersten Stelle in der abgeleiten Klasse

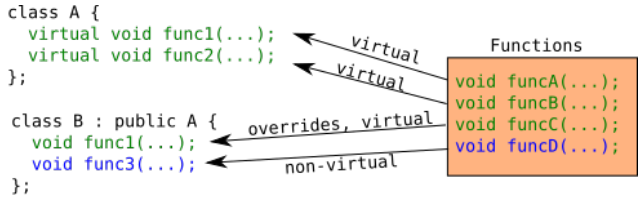
- Bei anonymen Struktur
  - Kürzerer Zugriffsname
  - Namenskonflikte wahrscheinlicher
- Bei benannte Struktur
  - Längerer Zugriffsname
  - Namenskonflikte werden vermieden

```
1  struct A {
2    int p;
3  };
4
5  struct B {
6    A super;
7    int q;
8    int p; // Kein Namenskonflikt
9  };
10
11 B b;
12 b.super.p = 5;
13 b.q = 6;
14 b.p = 7;
```

```
1  struct A {
2    int p;
3  };
4
5  struct B {
6    A;
7    int q;
8    // int p; // Name vergeben
9  };
10
11 B b;
12 b.p = 5;
13 b.q = 6;
```

Table Of Content

Konzept der Tabelle der virtuellen Funktionen



C++-Beispiell [1]

```
1   #ifndef  employee_INC
2   #define  employee_INC
3
4   class Employee {
5     protected:
6       char* firstname;
7       char* lastname;
8     public:
9       Employee(const char* firstname, const char* lastname);
10      ~Employee();
11      virtual void print();
12  };
13
14  #endif   /* ----- #ifndef employee_INC  ----- */
```

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include <string.h>
4
5   #include "debug.hpp"
6   #include "employee.hpp"
7
8   Employee::Employee(const char* firstname, const char* lastname) {
9     /*  do something */
10  }
11
12  Employee::~Employee() {
13    /*  do something */
14  }
15
16  void Employee::print() {
17    /*  do something */
18  }
```

C++-Beispiel [2]

```
1   #ifndef  manager_INC
2   #define  manager_INC
3
4   #include "employee.hpp"
5
6   class Manager : public Employee {
7     private:
8       int max_group_size;
9       int group_size;
10      int level;
11      Employee** group;
12    public:
13      Manager(const char* fn, const char* ln, int level);
14      ~Manager();
15      virtual int add_member(Employee* employee);
16      virtual void print();
17  };
18
19  #endif   /* ----- #ifndef manager_INC  ----- */
```

```
1   #include <stdlib.h>
2   #include <stdio.h>
3   #include <string.h>
4
5   #include "debug.hpp"
6   #include "manager.hpp"
7
8   Manager::Manager(const char* fn, const char* ln, int level)
9     : Employee(fn, ln) {
10    /* do something */
11  }
12
13  Manager::~Manager() {
14    /* do something */
15  }
16
17  int Manager::add_member(Employee* employee) {
18    /* do something */
19    return 0;
20  }
21
22  void Manager::print() {
23    /* do something */
24  }
```

Umsetzung in C

- Ein Konzept für die Verwaltung der virtuellen Funktionen ist erforderlich
- Typ der virtuellen Tabelle kann eine Struktur mit Funktionspointern sein
  - Wird für jede virtuelle Klasse erstellt und einmal im globalen Namespace instanziiert
- Virtuelle Tabelle muss manuell gepflegt werden
- Nur die Basisklasse speichert den Pointer auf die virtuelle Tabelle
  - Null Pointer kann eine abstrakte Klasse repräsentieren

Basisklasse

Source Code 1: employee.h

```
1   /* virtual table (forward declaration) */
2   typedef struct employee_vtbl_t employee_vtbl_t;
3
4   /*  base class */
5   typedef struct employee_t {
6     employee_vtbl_t *vtbl;
7   } employee_t;
8
9   /* member function (declaration) */
10  void employee_print(void *_this);
11
12  /* virtual table (type) */
13  struct employee_vtbl_t {
14    void  (*print)(void *employee);
15  };
```

Source Code 2: employee.c

```
1   /*  virtual table (instantiation) */
2   static employee_vtbl_t employee_vtbl = {
3     .print = employee_print
4   };
5
6   /* member functions (implementation) */
7   void employee_print(void *_this) {
8     employee_t *this = (employee_t*) _this;
9     /* do something */
10  }
```

Abgeleitete Klasse

Source Code 3: manager.h

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

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

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

Source Code 4: manager.c

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

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

void manager_print(void *_this) {
  manager_t *this = (manager_t*) _this;
  /* do something */
}
```

Funktionsaufruf mit Casting

```
1    /*  create and construct employee and manager */
2    employee.vtbl->print(&employee);
3    ((manager_vtbl_t*) ((employee_t*) &manager)->vtbl)->add_member(&manager, &employee);
4    ((manager_vtbl_t*) ((employee_t*) &manager)->vtbl)->print(&manager);
5    /* destroy manager and employee*/
```

- Vorteile
  - Nativer C-Zugriff auf die virtuelle Funktionen
  - Das Zugriffsweise ist unabhängig von der Tiefe der Verschachtelung
- Nachteile
  - Die Zugriffsweise ist relativ kompliziert und fehleranfällig

Funktionsaufruf mit Makro

```
1  #define M_employee_print(me) ((employee_t*)(me))->vtbl->print(me)
2  #define M_manager_add_member(me, employee) ( ((manager_vtbl_t*) ((employee_t*) me)->vtbl)->add_member((me), (employee)))
3  #define M_manager_print(me) ( ((manager_vtbl_t*) ((employee_t*) me)->vtbl)->print(me))
4
5  /*  create and construct employee and manager */
6  M_employee_print(&employee);
7  M_manager_add_member(&manager, &employee);
8  M_manager_print(&manager);
9  /* destroy manager and employee*/
```

- Vorteile
  - Die Makros verstecken den relativ komplexen Ausdruck
  - Intuitive Benutzung
- Nachteile
  - Die Makros können das Debugging erschweren

Funktionsaufruf über die Basis-Variable

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

- Vorteile
  - Ein Cast wird erspart
- Nachteile
  - Bei tieferen Ableitungen muss das `super` mehrmals verwendet werden.
    - Potenzielle Fehlerquelle bei tiefen Ableitungshierarchien

Table Of Content

Speicherung der basisgleicher Objekte in einem Container

```
1   #include "employee.h"
2   #include "manager.h"
3
4   int main(int argc, char** argv) {
5     /* create employees and manager */
6
7     employee_t* staff[3];
8     staff[0] = &employee1;
9     staff[1] = &employee2;
10    staff[2] = &manager;
11
12    for (int i = 0; i < 3; ++i) {
13      M_employee_print(staff[i]);
14    }
15
16    /* cleanup */
17  }
```

```
1   Employee: Ulrike Müller
2   Employee: Hans Meier
3   Manager (1) Matthias Gross
4     Group member 0: Ulrike Müller
5     Group member 1: Hans Meier
```

- Objekte mit der gleicher Basis können z.B. in einem Array gespeichert werden
- Es wird die richtige virtuelle Funktion aufgerufen

Table Of Content

Zusammenfassung

- Objektorientierte Programmierung ist möglich in C
- Einfache Klassen sind relativ einfach zu implementieren und zu benutzen
- Vererbung
  - Ausnutzung der fixen Reihenfolge der Membervariablen in Strukturen
  - Tabelle mit virtuellen Funktionen muss manuelle implementiert werden
- OOP in C verlangt viel Disziplin vom Programmierer

Table Of Content

Literatur

[1] Antonio Maiorano. *Virtual Functions in C*. 2014. URL: http://vgcoding.blogspot.com/2014/03/virtual-functions-in-c.html (besucht am 05. 09. 2018).

[2] Dan Saks. *Virtual Functions in C*. 2012. URL: https://www.embedded.com/electronics-blogs/programming-pointers/4391967/Virtual-functions-in-C (besucht am 05. 09. 2018).

[3] *Virtual Funktions in C*. 2010. URL: flinflon.brandonu.ca/dueck/1997/62285/virtualc.html (besucht am 21. 01. 2019).