Objekt-Orientiertes Programmieren in C

Raphael Diziol, Timo Bingmann, Jörn Heusipp

14. Juni 2005

Inhalt

OO Grundmodell

Modell: Ein Programm ist ein System kooperierender Objekte.

Objekte haben begrenzte Lebensdauer und einen lokalen Zustand.

Objekte können Nachrichten verarbeiten und dabei ihren Zustand ändern, neue Objekte erzeugt und alte löschen.

Um ein Program zu schreiben, definieren wir eine Menge von Objekten mit Daten und Verarbeitungsschritte und bauen diese sinnvoll zusammen.

OO Konkret

Wir wollen insbesondere die Umsetzung des OO Modells in C++ betrachten. Andere Programmiersprachen setzen andere Aspekte des OO Modells zum Teil besser um.

In C++ bildet man Klassen (eine Art Blaupause) von Objekten bestehend aus Attributen und Methoden.

Auf Java werden wir nur am Rande (gar nicht) eingehen.

OO Konzepte

Dazu eine Übersicht über die umzusetzende Prinzipen:

- Kapselung
- Geheimnisprinzip
- Vererbung
- Polymorphie
- Überladung
- Generizität

OO Konzepte: Kapselung und Geheimnisprinzip

Kapselung

- Strukturierung eines Programms in mehrere voneinander unabhängig operierende Objekte
- Verallgemeinerung des Modulbegriffs des klassischen imperativen Programmierparadigmas

Geheimnisprinzip

- Verbergen interner Daten $\Rightarrow$ semantische Konsistenz
- Zugriff ausschließlich über *wohldefinierten öffentlichen Schnittstelle*

OO Konzepte: Vererbung

Vererbung

- Wiederverwendung von Code
- *Spezialisierung* von Objektklassen
- Einordnung von Klassen in eine *Vererbungshierarchie*
- hierdurch Einsetzbarkeit eines *Subtyps* anstelle der Oberklasse

OO Konzepte: Polymorphie

Polymorphie

- Subtyping: abgeleitete Objekte an Stelle von Basistypen
- Dynamisches Binden *überschriebene "virtueller" Methoden* zur Laufzeit

OO Konzepte: Überladung

Überladung

- Gruppierung *sematisch ähnlicher Funktionen mit unterschiedlicher Signatur* unter einen Namen
- Auswahl der konkreten Funktion *zur Compilezeit* durch Typen der Parametervariablen
- “Feature” für Schreibfaule

OO Konzepte: Generizität

Generizität

- Ziel: Typinformation erhalten bei Datenhaltungsklasse
- Parametrisierung von Klassen mit abstrakten Datentypen
- In C++ : `template` classes

OO geht in C!

OO geht in C!

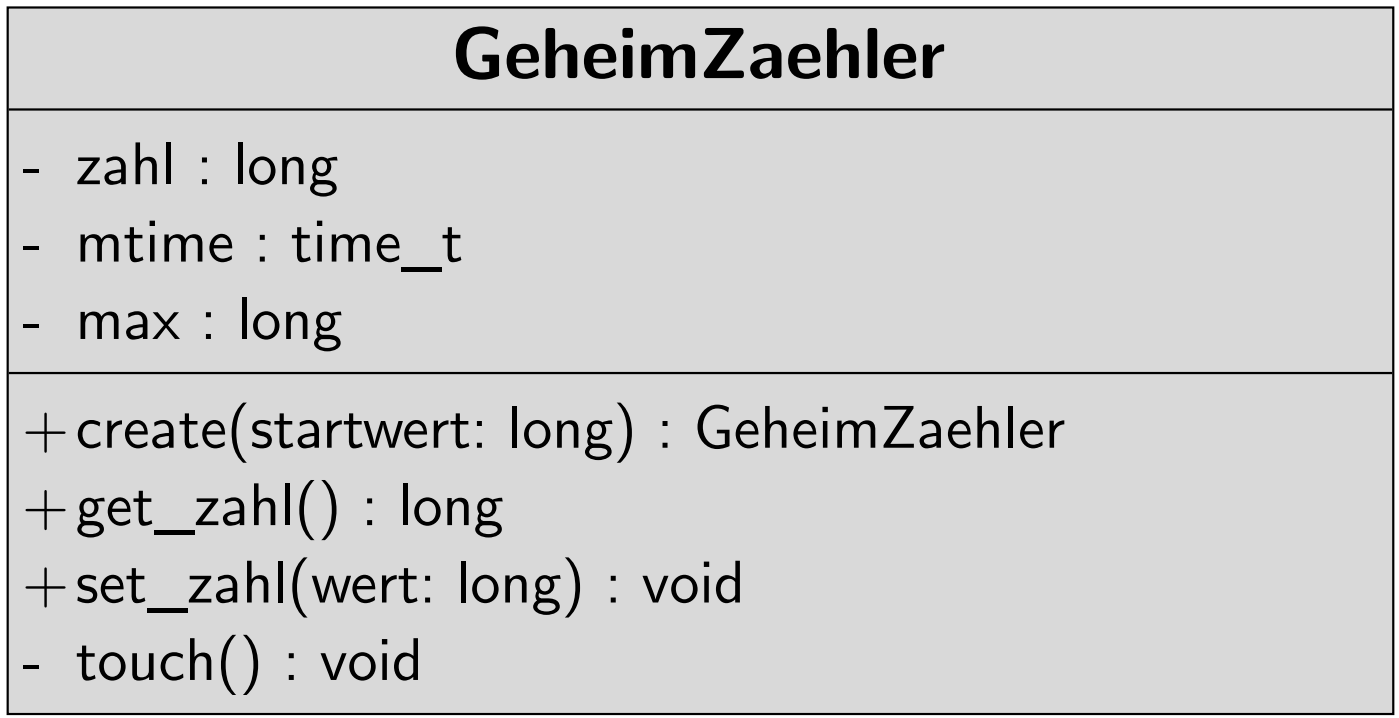
OO geht in C!

Systematische Umsetzung der Konzepte mit den Mitteln, die uns C zur Verfügung stellt.

Dazu werden wir die Möglichkeiten anhand von einigen “real-life” Beispielen ausführen und untersuchen.

Zuerst wollen wir die Konzepte der Daten-Kapselung und das Geheimnisprinzip in C abbilden.

UML Diagramm eines GeheimZaehlers



GeheimZaehler.hpp

```
/* GeheimZaehler.hpp */

class GeheimZaehler {
private:
     long zahl;
     time_t mtime;
     long max;

     void touch();
public:
     GeheimZaehler(long startwert);

     long get_zahl();
     void set_zahl(long wert);
};
```

Umbauen der C++ Klasse

```
/* GeheimZaehler.h */
```

Umbauen der C++ Klasse

```
/* GeheimZaehler.h */

struct GeheimZaehler {
     long zahl;
     time_t mtime;
     long max;
};
```

Umbauen der C++ Klasse

```
/* GeheimZaehler.h */

struct GeheimZaehler {
     long zahl;
     time_t mtime;
     long max;
};

/* Konstruktor */
GeheimZaehler* geheimzaehler_create(long startwert);
```

Umbauen der C++ Klasse

```
/* GeheimZaehler.h */

struct GeheimZaehler {
     long zahl;
     time_t mtime;
     long max;
};

/* Konstruktor */
GeheimZaehler* geheimzaehler_create(long startwert);

/* Methoden */
long geheimzaehler_get_zahl(GeheimZaehler *this);
void geheimzaehler_set_zahl(GeheimZaehler *this, long wert);
```

Umbauen der C++ Klasse

```
/* GeheimZaehler.h */

/* opaque pointer */
typedef struct _GeheimZaehler GeheimZaehler;
```

```
/* Konstruktor */
GeheimZaehler* geheimzaehler_create(long startwert);

/* Methoden */
long geheimzaehler_get_zahl(GeheimZaehler *this);
void geheimzaehler_set_zahl(GeheimZaehler *this, long wert);
```

```
/* GeheimZaehler.c */
struct _GeheimZaehler {
     long zahl;
     time_t mtime;
     long max;
};
static void geheimzaehler_touch(GeheimZaehler *this) {
    this->mtime = time(NULL);
}
GeheimZaehler* geheimzaehler_create(long startwert) {
     GeheimZaehler* n = malloc(sizeof *n);
     n->zahl = n->max = startwert;
     geheimzaehler_touch(n);
     return n;
}
long geheimzaehler_get_zahl(GeheimZaehler *this) {
     return this->zahl;
}
void geheimzaehler_set_zahl(GeheimZaehler *this, long wert) {
     this->zahl = wert;
     if (this->max < wert) this->max = wert;
     geheimzaehler_touch(this);
}
```

Vererbung und Polymorphie

Umsetzung in C:

- Keine Unterstützung der Sprache
  $\Longrightarrow$ manuelles Kopieren der Datenfelder
- Typanpassung durch explizite Casts

Wir wollen dies an dem folgenden Beispiel untersuchen.

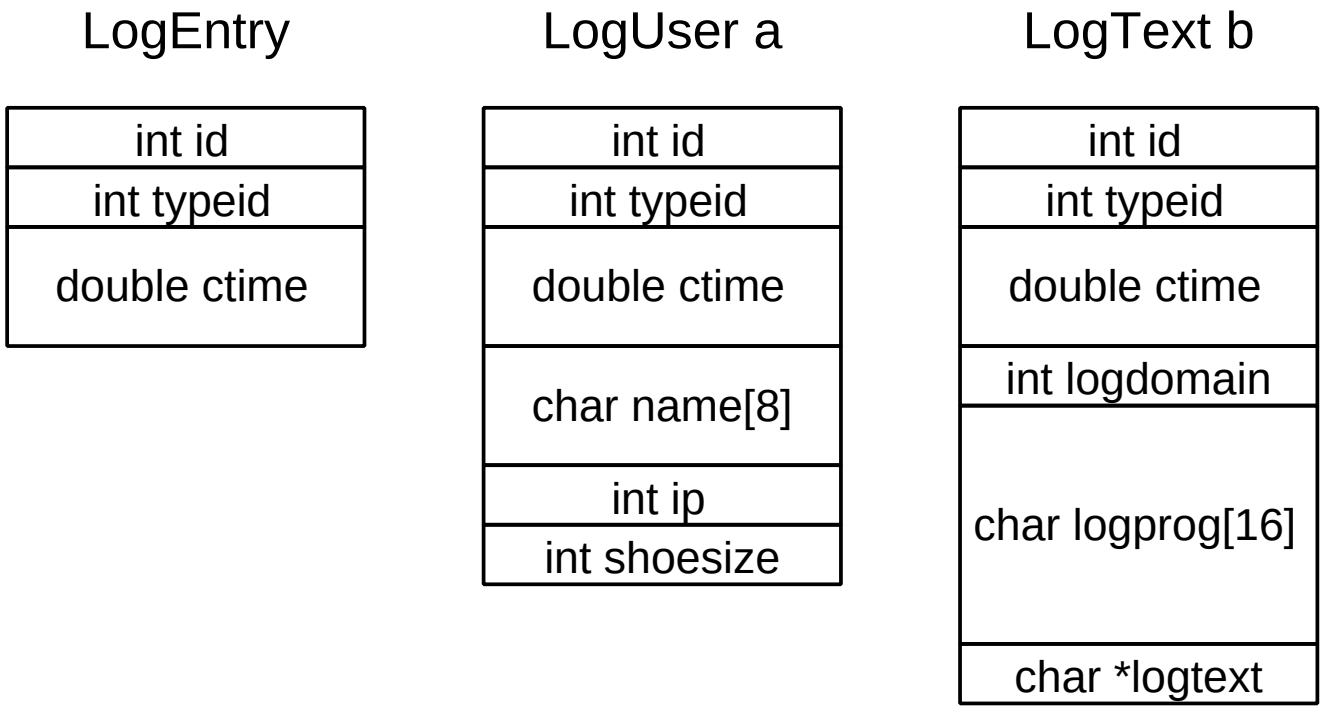
Ableitung in C

```
struct LogEntry {
  int id;
  int typeid;
  double ctime;
};
```

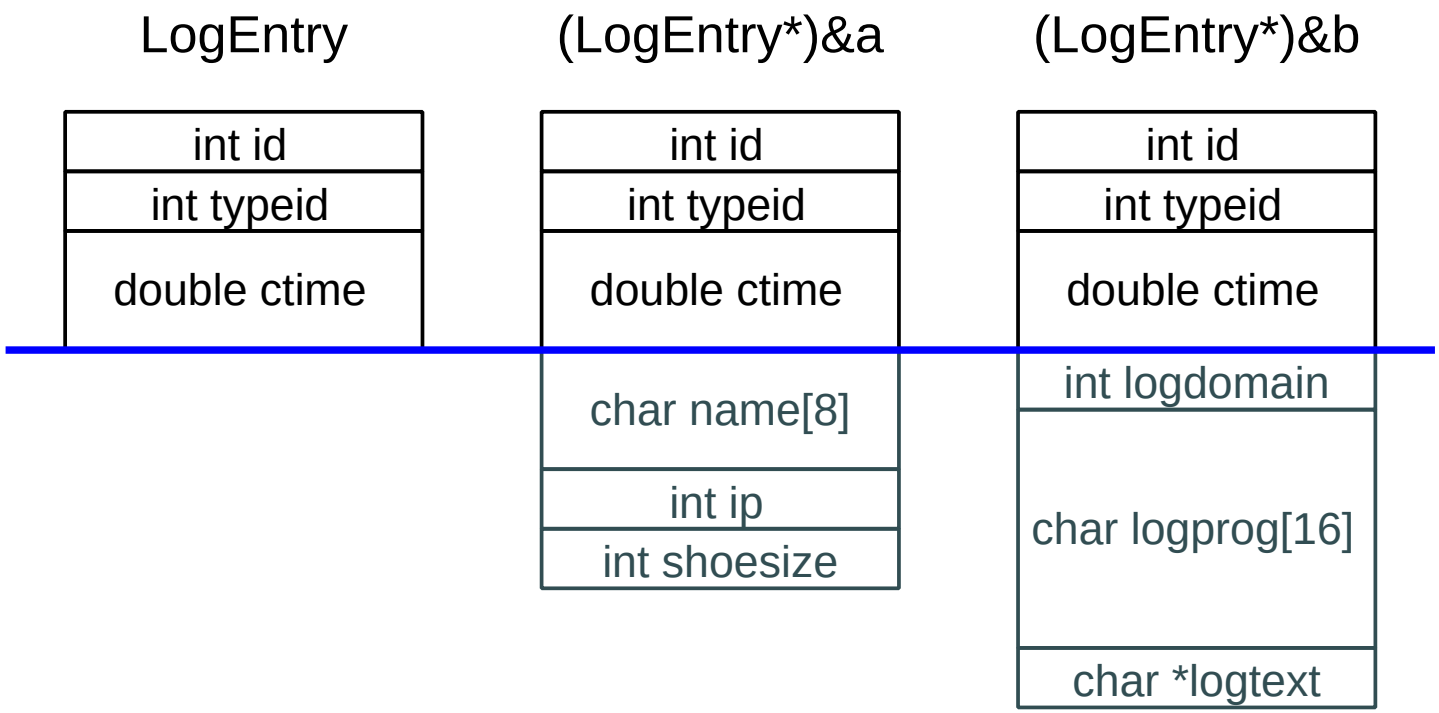
```
struct LogUser {
  int id;
  int typeid;
  double ctime;
  char name[8];
  int ip;
  int shoesize;
} a;
```

```
struct LogText {
  int id;
  int typeid;
  double ctime;
  int logdomain;
  char logprog[16];
  char *logtext;
} b;
```

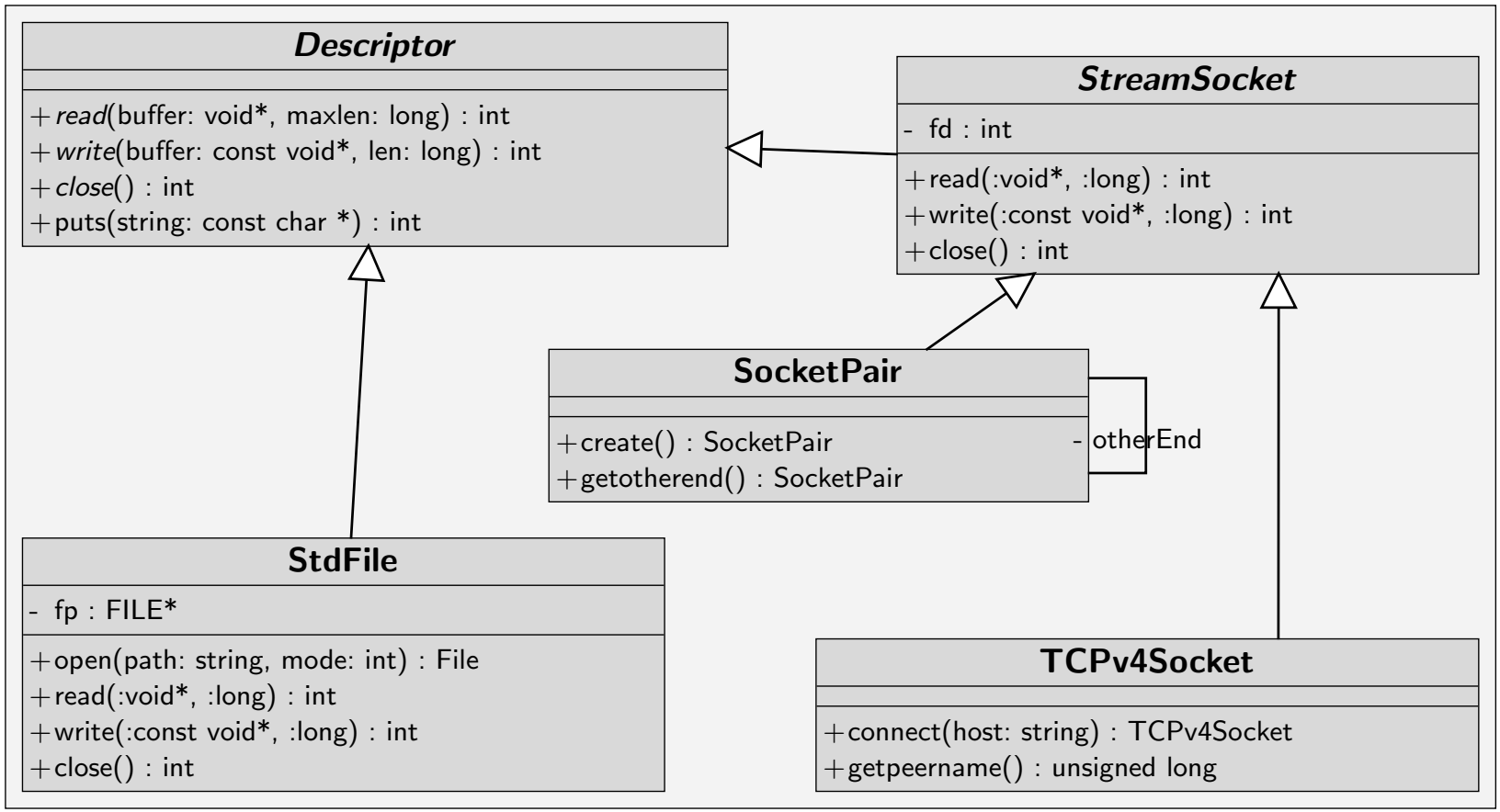
Ableitung in C



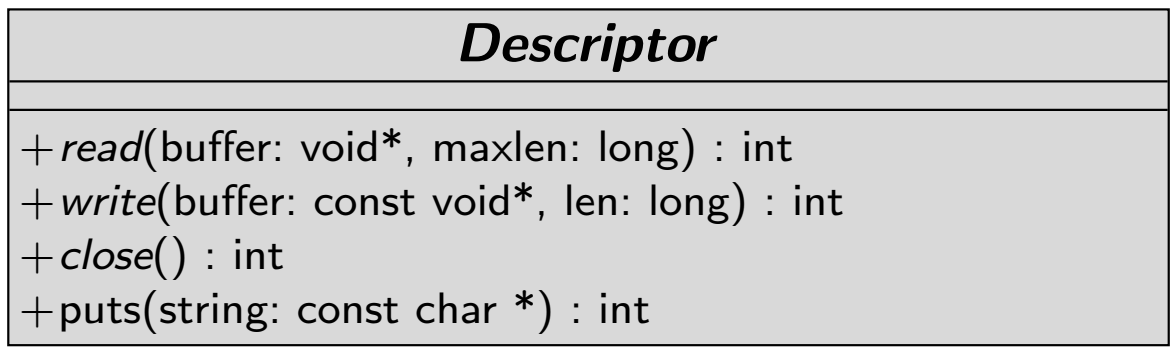
Ableitung in C



UML Diagramm der Descriptor-Hierarchie



Funktionspointer

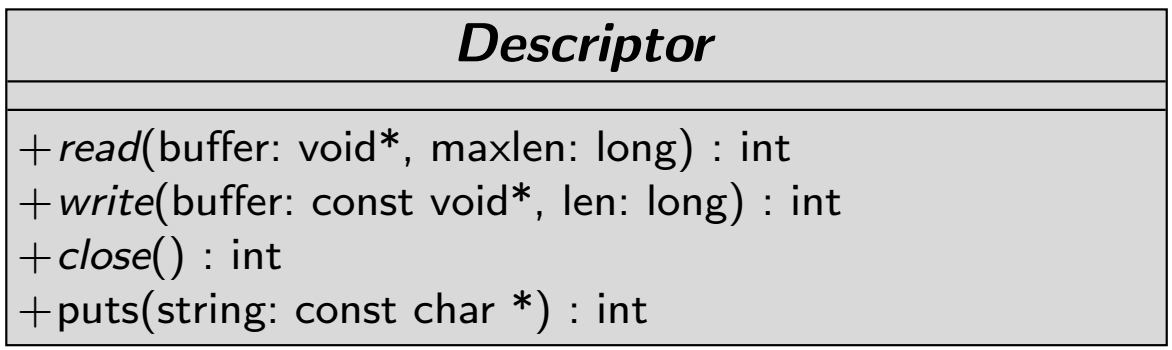


Funktionspointer

```
/* Descriptor.h */

typedef struct _Descriptor Descriptor;

struct _Descriptor {
     int (*read)(Descriptor *this, void* b, int ml);
     int (*write)(Descriptor *this, const void *b, int l);
     int (*close)(Descriptor *this);
};
```



Funktionspointer

```
/* Descriptor.h */

typedef struct _Descriptor Descriptor;

struct _Descriptor {
     int (*read)(Descriptor *this, void* b, int ml);
     int (*write)(Descriptor *this, const void *b, int l);
     int (*close)(Descriptor *this);
};
```

```
/* opaque objects */
typedef struct _StdFile StdFile;
/* constructor prototypes */
StdFile* stdfile_open(const char *path, const char* mode);
```

Funktionspointer

```
/* Descriptor.h */

typedef struct _Descriptor Descriptor;

struct _Descriptor {
     int (*read)(Descriptor *this, void* b, int ml);
     int (*write)(Descriptor *this, const void *b, int l);
     int (*close)(Descriptor *this);
};
```

```
/* opaque objects */
typedef struct _StdFile StdFile;
/* constructor prototypes */
StdFile* stdfile_open(const char *path, const char* mode);
```

```
Descriptor *f = (Descriptor*)stdfile_open("datei.txt","w");
f->write(f,"test",4);
f->close(f);
```

```
/* aus Descriptor.c */
struct _StdFile {
     int (*read)(StdFile *this, void* b, int ml);
     int (*write)(StdFile *this, void *b, int l);
     int (*close)(StdFile *this);
     FILE *fp;
};
static int stdfile_read(StdFile *this, void *buffer, int maxlen) {
     return fread(buffer, maxlen, 1, this->fp);
}
static int stdfile_write(StdFile *this, void *buffer, int len) {
     return fwrite(b, len, 1, this->fp);
}
static int stdfile_close(StdFile *this) {
     fclose(this->fp);
     free(this);
     return 1;
}
StdFile* stdfile_open(const char *path, const char* mode) {
     StdFile *n;
     FILE *fp = fopen(path, mode);
     if (fp == NULL) return NULL;
     n = malloc(sizeof *n);
     n->read = stdfile_read;
     n->write = stdfile_write;
     n->close = stdfile_close;
     n->fp = fp;
     return n;
}
```

```
/* Descriptor.h */

typedef struct _Descriptor Descriptor;
struct _Descriptor {
     int (*read)(Descriptor *this, void* b, int ml);
     int (*write)(Descriptor *this, const void *b, int l);
     int (*close)(Descriptor *this);
};

/* opaque objects */
typedef struct _StdFile StdFile;
typedef struct _SocketPair SocketPair;
typedef struct _TCPv4Socket TCPv4Socket;

/* constructor prototypes */
StdFile* stdfile_open(const char *path, const char* mode);
SocketPair* socketpair_create(void);
TCPv4Socket* tcpv4socket_connect(const char *host, int port);

/* object methods */
int descriptor_puts(Descriptor *this, const char *string);
SocketPair* socketpair_getother(SocketPair *this);
unsigned long tcpv4socket_getpeername(TCPv4Socket *this);
```

Generizität

In C zwei Möglichkeiten:

- mit einem Präprozessor-Makro
- mit `void*`

Generizität mit Makros

Mit einem Makro

- Definieren von template Makros mit Typnamen als Parameter.
- Makros definieren Funktionen für die angegebenen Typen.
- Total generisch. Aber: nicht praktikabel für größere Projekte.
- Dazu ein (Abschreckungs-)Beispiel

```
/* GenStack.h */
#define DefineGenStack(T)                                          \
struct GenStack_##T {                                              \
     int top, len;                                                 \
     T *data;                                                      \
};                                                                 \
typedef struct GenStack_##T GenStack_##T;                          \
static inline GenStack_##T *stack_##T##_create() {                 \
     GenStack_##T *n = malloc(sizeof *n);                          \
     n->top = n->len = 0;                                          \
     n->data = NULL;                                               \
     return n;                                                     \
}                                                                  \
static inline                                                      \
void stack_##T##_push(GenStack_##T *s, T e) {                      \
     if (s->top+1 > s->len) {                                      \
        s->len += 32;                                              \
        s->data = realloc(s->data, s->len * sizeof(T));            \
     }                                                             \
     s->data[s->top++] = e;                                        \
}                                                                  \
static inline T stack_##T##_pop(GenStack_##T *s) {                 \
     if (s->top > 0) return s->data[--s->top];                     \
     return 0;                                                     \
}
```

Generizität mit `void*`

Mit `void*`

- Bekanntes Vorgehen in C
- Stackfunktionen arbeiten auf `void*`
- Typinformation geht beim Einfügen verloren
- Bei Entfernen durch expliziten cast wieder dazugefügt

Genau das wollten wir verhindern.

$\Rightarrow$ Keine echte generische Programmierung.

Darstellung von C++ in Maschine

```
/* VTable.cc */
class Uni {
private:
     long long c;
public:
     Uni() { c = 0x00C0FFEE0000BABEll; }
     virtual int get() {
          return 42;
     }
     virtual void set(long long v) {
          c = v;
     }
};
class Earth : public Uni {
public:
     virtual const char *get_letters() {
          return "What is 6 times 9?";
     }
};
int main() {
     Uni u;
     fwrite(&u, sizeof(u), 1, stdout);
}
```

vtable - Analyse des Binary

```
./VTable | hexdump
```

```
Uni:   87e0 0804 babe 0000 ffee 00c0
Earth: 87c8 0804 babe 0000 ffee 00c0
```

vtable - Analyse des Binary

```
./VTable | hexdump
```

```
Uni:   87e0 0804 babe 0000 ffee 00c0
Earth: 87c8 0804 babe 0000 ffee 00c0
```

```
objdump -s VTable
```

```
Contents of section .rodata:
 80487a0 03000000 01000200 aaaaaaaa 00576861  .............Wha
 80487b0 74206973 20362074 696d6573 20393f00  t is 6 times 9?.
 80487c0 00000000 f0870408 4a860408 54860408  ........J...T...
 80487d0 78860408 00000000 00000000 e8870408  x...............
 80487e0 4a860408 54860408 b8990408 03880408  J...T...........
 80487f0 88990408 fc870408 e8870408 35456172  ............5Ear
 8048800 74680033 556e6900                    th.3Uni.
```

vtable - Analyse des Binary

`objdump -s VTable`

```
Contents of section .rodata:
 80487a0 03000000 01000200 aaaaaaaa 00576861  .............Wha
 80487b0 74206973 20362074 696d6573 20393f00  t is 6 times 9?.
 80487c0 00000000 f0870408 4a860408 54860408  ........J...T...
 80487d0 78860408 00000000 00000000 e8870408  x...............
 80487e0 4a860408 54860408 b8990408 03880408  J...T...........
 80487f0 88990408 fc870408 e8870408 35456172  ............5Ear
 8048800 74680033 556e6900                    th.3Uni.
```

`objdump -d VTable`

```
0804864a <_ZN3Uni3getEv>:
 804864a:       55                      push   %ebp
 804864b:       89 e5                   mov    %esp,%ebp
 804864d:       b8 2a 00 00 00          mov    $0x2a,%eax
 8048652:       5d                      pop    %ebp
 8048653:       c3                      ret
```

vtable - Analyse des Binary

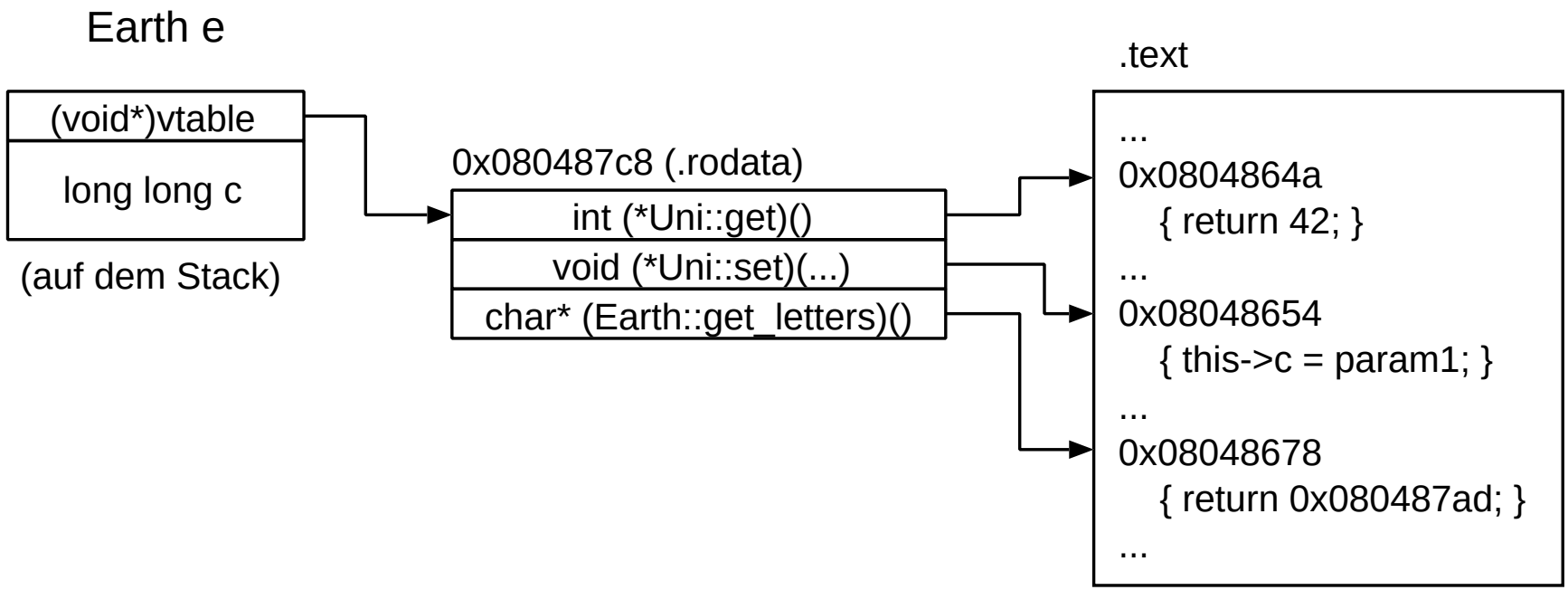
`objdump -s VTable`

```
Contents of section .rodata:
 80487a0 03000000 01000200 aaaaaaaa 00576861  .............Wha
 80487b0 74206973 20362074 696d6573 20393f00  t is 6 times 9?.
 80487c0 00000000 f0870408 4a860408 54860408  ........J...T...
 80487d0 78860408 00000000 00000000 e8870408  x...............
 80487e0 4a860408 54860408 b8990408 03880408  J...T...........
 80487f0 88990408 fc870408 e8870408 35456172  ............5Ear
 8048800 74680033 556e6900                    th.3Uni.
```

Adressen anderer Funktionen.

```
080485ee <_ZN3UniC1Ev>:
0804860e <_ZN5EarthC1Ev>:
0804862a <_ZN3UniC2Ev>:
0804864a <_ZN3Uni3getEv>:
08048654 <_ZN3Uni3setEx>:
08048678 <_ZN5Earth11get_lettersEv>:
```

vtable - Pointerdiagramm



Name-Mangling - Warum?

```
void Anzeigen(int  i)
{
    printf("%d\n",i);
}
void Anzeigen(double  f)
{
    printf("%f\n",f);
}
```

- In einer Binary kann jeder Funktionsname nur einmal vorkommen.
- Funktion wird nicht nur durch ihren Namen definiert!

$\Rightarrow$ Verändern (*Mangling*) des Namens in eine eindeutige Form.

- Jeder Compiler hat andere Konvention (sogar pro Version)

Name-Mangling - Vorgehensweise allgemein

Elemente die eine Funktion eindeutig beschreiben:

1. den Funktionsnamen
2. den Klassennamen und ggf. `namespace`-Namen
3. die Parameterliste

Name-Mangling - Vorgehensweise allgemein

Elemente die eine Funktion eindeutig beschreiben:

1. den Funktionsnamen
2. den Klassennamen und ggf. `namespace`-Namen
3. die Parameterliste
   1. Die C++ internen Datentypen int, long, short, char, long long werden durch `i`, `l`, `s`, `c`, `x` gemangelt.
   2. Unsigned Datentypen haben ein extra `U` vorangestellt. `unsigned char` wird `Uc`
   3. Pointer werden durch ein vorangestelltes `P` gekennzeichnet und Referenzen durch `R`

Name-Mangling - Beispiel

```
struct Point
{
    float x;
    float y;
};

namespace Graph
{
    class Test
    {
    public:
        float get_x(Point* p, int i);
        /* _ZN5Graph4Test5get_xEP5Pointi */
    };
}

void set_x(Point* p,int i);        /* _Z5set_xP5Pointi */
```

Name-Mangling - Beispiel

`_Z5set_xP5Pointi`

1. `_Z` für gemangeltes Symbole
2. `5set_x` ist der Funktionsname der 5 Zeichen enthält
3. `P` der erste Parameter ein Pointer ist
4. `5Point` der Namen des Typs
5. `i` der letzte Parameter: ein int

Name-Mangling - Beispiel

`_ZN5Graph4Test5get_xEP5Pointi`

1. `_Z` für gemangeltes Symbole
2. `N...E` schliesst eine Reihe von Namen ein
3. `5Graph4Test5get_x` Namespace, Klassenname, Funktionsname jeweils vom Typ (Länge, Name)
4. `P5Pointi` wie vorige Folie

RTTI

- RTTI - run time type information
- Zur Laufzeit direkt den Typ eines (dynamischen) Objekts abfragen.

RTTI - Wie?

- `#include <typeinfo>`
- Typeinfo im Compiler einschalten: `-frtti`
- typeid(object) bzw. typeid(typename) liefert const struct type_info&
- Vergleiche mit == und != benutzen.
- const char *type_info::name() const (Format ist compilerabhängig)

RTTI - Beispiel

```
class Shape {
public:
     virtual void Test() { printf("Shape\n"); }
};
class Rectangle : public Shape {
public:
     virtual void Test() { printf("Rectangle\n"); }
};
class Circle : public Shape {
public:
     virtual void Test() { printf("Circle\n"); }
};
class Box : public Shape {
public:
     virtual void Test() { printf("Box\n"); }
};
```

RTTI - Beispiel

```
void printType(Shape* sp)
{
     if (typeid(*sp) == typeid(Rectangle))
          printf("Es ist ein Rectangle\n");
     else if (typeid(*sp) == typeid(Circle))
          printf("Es ist ein Circle\n");
     else
          printf("Ja, was ist es denn...? %s\n",
                 typeid(*sp).name());
}
```

RTTI - Beispiel

```
int main()
{
     Rectangle r;
     Shape *sp = &r;
     printType(sp);

     Box b;
     sp = &b;
     printType(sp);

     return 0;
}
```

Ausgabe:

```
Es ist ein Rectangle
Ja, was ist es denn...? 3Box
```

RTTI - Compiler

Umsetzung im C++ Compiler:

1. Keine Pointer oder Referenzen
   - Typ zur Compilezeit bekannt
   - direkter Verweis auf das entsprechende `type_info`-Objekt
2. Nicht polymorphe Klasse
   - Typinformation bei Zeigern und Referenzen kann nur statisch sein.
   - wie oben.
3. Pointer auf Polymorphe Klassen
   - Im vtable ist ein Eintrag auf das passende `type_info`-Objekt
   - Nur zusätzlicher Speicher pro Typ!

vtable - Analyse des Binary

`objdump -s VTable`

```
Contents of section .rodata:
 80487a0 03000000 01000200 aaaaaaaa 00576861  .............Wha
 80487b0 74206973 20362074 696d6573 20393f00  t is 6 times 9?.
 80487c0 00000000 f0870408 4a860408 54860408  ........J...T...
 80487d0 78860408 00000000 00000000 e8870408  x...............
 80487e0 4a860408 54860408 b8990408 03880408  J...T...........
 80487f0 88990408 fc870408 e8870408 35456172  ............5Ear
 8048800 74680033 556e6900                    th.3Uni.
```

`objdump -d VTable`

```
0804864a <_ZN3Uni3getEv>:
 804864a:       55                      push   %ebp
 804864b:       89 e5                   mov    %esp,%ebp
 804864d:       b8 2a 00 00 00          mov    $0x2a,%eax
 8048652:       5d                      pop    %ebp
 8048653:       c3                      ret
```

Conclusion

Conclusion

Linie 1

Leberknödel mit Sauerkraut und Püree
Vegetarisches Gyros mit Tsatsiki und Baguette-Brötchen

Linie 2

Schweinebraten mit Biersoße
Semmelknödel

Linie 3

Spätzle-Pilz-Pfanne mit Gemüse

Linie 4 + 5

Zwiebelbraten vom Rind Kartoffel-Käsetaschen mit Curry-Dip
Kroketten

Schnitzel-Bar

Schweineschnitzel mit Beilagen