Objekt-Orientiertes Programmieren in C

Raphael Diziol, Timo Bingmann, Jörn Heusipp

14. Juni 2005

Inhaltsverzeichnis

1 Objekt-Orientierte Konzepte

Zuerst wollen wir eine kurze Übersicht über die Grundkonzepte des Objekt-Orientieren Programmierparadigmas geben, um daraufhin systematisch Umsetzungsmöglichkeiten dieser in plain C zu zeigen.

Im Objekt-Orientierten Grundmodell wird eine Programmausführung als ein *System kooperierender Objekte* angesehen. Die Objekte haben einen *lokalen Zustand*, der sich bei der Ausführung verändern kann, und eine gewisse *Lebensdauer*: Sie werden bei Programmstart oder während dem Ablauf erzeugt, und spätestens am Programmende gelöscht. Diese Objekte empfangen und verarbeiten ankommende *Nachrichten*, bei deren Bearbeitung ein Objekt seinen Zustand ändern kann, neue Objekte erzeugt oder existierenden gelöscht werden können.

Das Grundmodell der OO Programmierung kann auf verschiedene Weisen umgesetzt werden. Wir wollen in den folgenden Kapiteln dessen Umsetzbarkeit in der Sprache C untersuchen, wobei uns C++ als Vergleich dienen wird. Wir betrachten dann folgende allgemeine Prinzipien genauer:

Kapselung Durch die Strukturierung eines Programms in mehrere voneinander unabhängig operierende Objekte entsteht eine *Kapselung* der Objektdaten, welche eine Verallgemeinerung des Modulbegriffs des klassischen imperativen Programmierparadigmas darstellt.

Geheimnisprinzip Unterstützt durch die Kapselung sollen die internen Daten eines Objekts dem Benutzer verborgen werden. Zugriff auf das Objekt darf nur auf einer *wohldefinierten öffentlichen Schnittstelle* stattfinden.

Vererbung Eines der zentralen Ziele der OO ist die Wiederverwendung von Code durch *Erweiterung* einer bestehenden Objektklasse um speziellere Funktionalität. Andersherum können verschiedene spezielle Objekte unter einer abstrakteren Oberklasse in eine *Vererbungshierarchie* eingeordnet werden. Hierdurch wird die abgeleitete Klasse als Subtyp der Oberklasse in Funktionsaufrufen anstelle dieser einsetzbar. C++ trennt nicht die Konzepte Subtyping und Vererbung.

Polymorphie Durch Subtyping kann anstelle eines Basistyps eine beliebige davon abgeleitete Klasse verwendet werden. Durch dynamisches Binden der Methodenaufrufe des Objekts kommen die in der abgeleiteten Klasse *überschriebenen Methoden* zur Laufzeit anstelle derjenigen aus der Basisklasse zur Ausführung. Hierdurch wird es einfach einmal geschriebene Algorithmen auf neue Objekte mit passender Schnittstelle anzuwenden, ohne die bestehende Implementierung anzupassen.

Überladung Da Objektklassen auf Oberklassen implizit konvertiert werden, wird Überladung besonders nützlich. Diese erlaubt mehrere *sematisch ähnliche Funktionen mit unterschiedlicher Signatur* unter einen Namen zu gruppieren. Die Auswahl der konkret aufgerufenen Funktion wird *zur Compilezeit* durch die Typen der Parametervariablen getroffen.

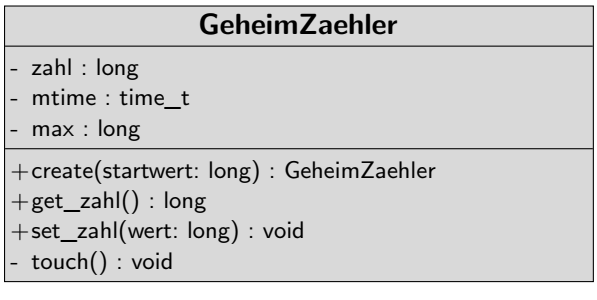
Generizität Oft will man ein Programmmodul auf Objekte beliebigen Typs anwenden können, ohne dass Typinformationen verloren gehen soll. Um typ-erhaltende Funktionen in beispielsweise einer Datenhaltungsklasse zu implementieren, kann diese *mit einem abstrakten Datentyp parametrisiert* werden. In C++ wird dieses Konzept *template* classes genannt.

2 OO geht in C!

Objekt-orientierte Programmierung in C (ohne plus plus) ist kein weisser Rappe, auch wenn C uns dabei nicht mit speziellen Sprachkonstrukten unterstützt.

2.1 Kapselung und Geheimnisprinzip

Ausgehend von einer Beispielklasse als UML Diagramm, wollen wir eine C Implementation erstellen, welche die beiden wesentliche OO Konzepte Kapselung und Geheimnisprinzip umsetzt.




```
/* 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);
};
```

Würde man diese Klasse in C++ implementieren, könnte die Headerdatei wie rechts aussehen. Auf eine Ausformulierung der Funktionsrümpfe werden wir verzichten.
Eine intuitive Implementierung in C kann durch Aufteilung des Codes in Include-Datei und Source-Datei eine Trennung von Interface und Implementierung erzwingen. Die Daten werden in eine struct verbunden, worauf dann verschiedene Funktionen angewandt werden können. Der implizite Objektkontext Parameter *this* muss in C explizit angegeben werden. Konstruktoren und *static* deklarierte Klassenmethoden haben keinen Objektkontext und daher keinen *this* Parameter. Da innerhalb eines ausführbaren Binary Symbole wie Funktionsnamen eindeutig sein müssen, wird hier durch Voranstellen des Klassennamen der Methodenname (hoffentlich) eindeutig gemacht. So wird der Konstruktor zu `geheimzaehler_create`, `get_zahl` zu `geheimzaehler_get_zahl`. Wie C++ dies umsetzt untersuchen wir später in Kap 3.3.

```
/* GeheimZaehler1.h */

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

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

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

Eine noch bessere Möglichkeit zum Verstecken der internen Datenstruktur bietet C in Form von unvollständigen Strukturdefinitionen: Man verwendet ein typedef auf eine nicht-definierte Struktur. So bleiben die Datenfelder und sogar die Speichergrösse der internen struct der Klasse vom Anwenderprogramm komplett verborgen. Dieses bekommt vom Konstruktor einen *opaque pointer* auf eine Speicheradresse, ohne zu wissen was dort steht.

```
/* GeheimZaehler.h */

typedef struct _GeheimZaehler GeheimZaehler;

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

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

Die tatsächliche Struktur wird dann in der C-Datei ausformuliert. Nur dieser Teil des Programms kann auch eine Instanz des Objekts erzeugen, denn nur hier ist die Speichergrösse bekannt.

```
/* GeheimZaehler.c */

#include "GeheimZaehler.h"

#include <stdlib.h>

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);
}
```

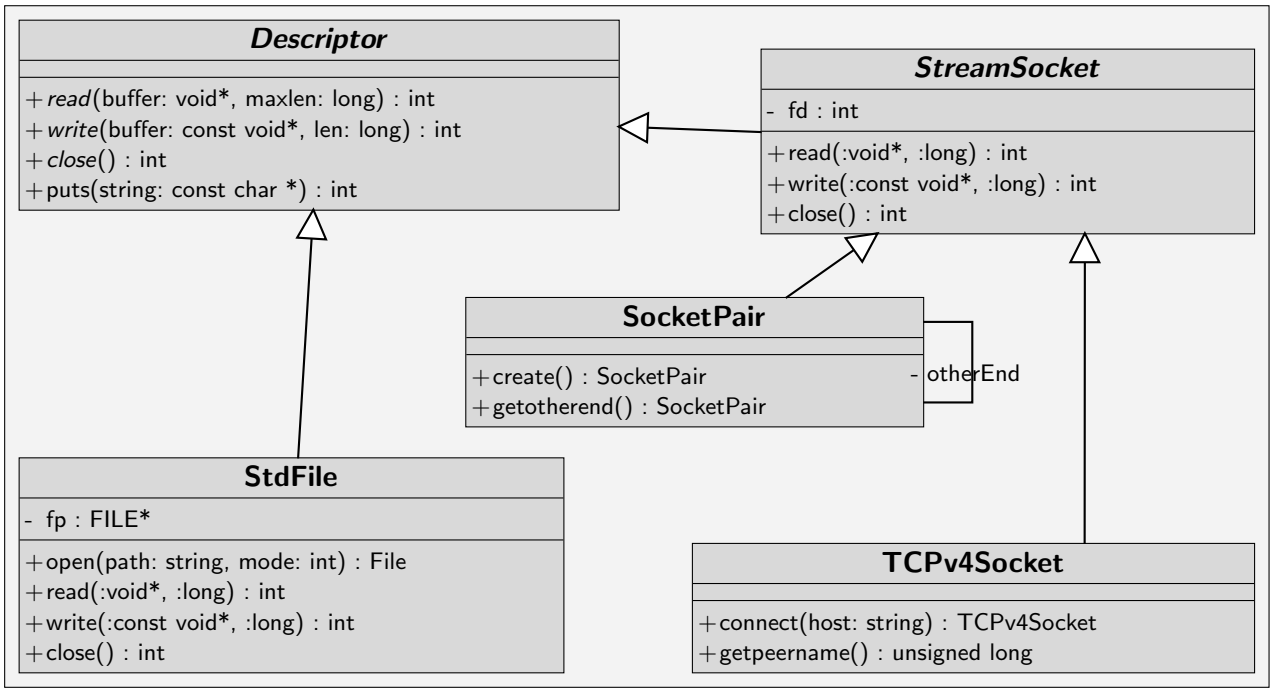
Unvollständige Strukturdefinitionen erlauben sogar eine bessere Umsetzung des Geheimnisprinzips, denn im Gegensatz zur C++ -Klassendefinition sind dem Benutzerprogramm nicht nur private Methoden, sondern auch die tatsächlichen Datenfelder nicht bekannt. Wie man oben erkennt, brauchen

private Methoden nicht außerhalb des C-Files bekannt sein, und können darum static deklariert werden.

2.2 Vererbung und Polymorphie

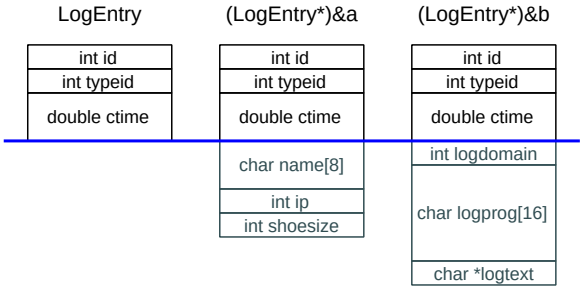
Zentral im OO Modell ist die Vererbung oder Spezialisierung von bestehenden Objektklassen, um so den Objektfunktionsumfang zu erweitern. Damit verknüpft ist die von OO Programmierern besonders geschätzte Polymorphie: die Verwendung vom abgeleiteten Objekten anstelle von Basistypen, wobei die in der abgeleiteten Klasse überschriebenen Funktionen anstelle derjenigen aus der Basisklasse zur Ausführung kommen.

Als Beispiel wollen wir eine abstrahierte Dateideskriptoren-Objekthierarchie modellieren. Die Hierarchie soll die Dateizugriffsfunktionen aus `stdio.h` mittels `FILE*` und aus `unistd.h` mittels `int fd` vereinigen.



Steht man vor der Aufgabe diese Klassen in C zu implementieren, stellt sich als erstes die einfache Frage wie Vererbung zu realisieren ist. Man will von den abgeleiteten structs auf die Oberstruct casten können, ohne die in beiden structs verfügbaren Felder zu verlieren. Darum liegt es nahe, am Anfang beider structs die gemeinsamen Felder zu definieren. Im Code kann dies durch Anlegen eines Datenfelds vom Typ der Oberstruct an erster Stelle in den abgeleiteten structs stattfinden, oder durch einfaches Kopieren aller Datenfelder der Oberklasse an den Anfang der abgeleiteten struct.

Das rechts stehende Diagramm soll verdeutlichen, wie man durch Kopieren der Datenstrukturen eine Quasi-Vererbung in C implementieren kann. Es sollen verschiedene Arten von Logeinträgen gespeichert werden: durch ein cast auf die Oberstruct `LogEntry` werden die Attribute der abgeleiteten structs verdeckt. Will man nachher den Logeintrag wieder umwandeln, kann man das typeid Feld untersuchen und durch einen (gefährlichen) downcast auf `LogUser` wieder die erweiterten Attribute verfügbar machen.



```
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;
```

Als nächste auffallende Implementierungsaufgabe in plain C stellen sich die abstrakte Methoden ("virtual" Funktionen in C++) in der Oberklasse *Descriptor*. Aufrufe dieser Methoden sollen, wenn man die Objektstruct hochcastet, nicht die Funktionen der Oberklasse aufrufen, sondern die der abgeleiteten Klassen. Im Beispiel vom Descriptor soll nach dem Öffnen des Streams, das abgeleitete Objekt nach *Descriptor* gecastet werden, wobei ein Aufruf von `read()` auf das Objekt dennoch die richtige Implementierung aufrufen soll.

Die Idee hierbei ist Funktionspointer in die struct der Oberklasse einzubetten. Diese Funktionspointer werden im Objektkonstruktor mit den richtigen Funktionsadressen belegt, und werden bei einem Cast auf die Oberstruct nicht verändert.

```
/* 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);
};
```

```
/* 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;
}
```

Oben sieht man eine Implementierung der von `Descriptor` abgeleiteten `StdFile` Klasse: die ausformulierte struct Definition ist wieder nur in der C-Datei vorhanden, wodurch das Geheimnisprinzip durchgesetzt wird. Von den obigen Funktionen wird nur `stdfile_open()` nach aussen exportiert, denn die "virtuelle" Methoden `stdfile_read()`, etc sollen nicht explizit durch ein nach aussenhin exportiertes Symbol aufgerufen werden können. Stattdessen erfolgt ein Methodenaufruf durch Aufruf des vom Konstruktor in die struct eingetragenen Funktionspointers, wobei wiederum der in C++ implizite *this*

Parameter mit übergeben werden muss. In dem folgenden Codesniplet wird die Implementierung von `StdFile` angewandt, wobei zuerst von der abgeleiteten Klasse auf die Oberklasse `Descriptor` gecastet wird.

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

Die genaue Implementierung der anderen Klassen wollen wir nicht mehr im Einzelnen untersuchen. Der vollständige Quellcode des Beispiels ist im Netz zusammen mit diesem Handout abrufbar. Dennoch schauen wir uns kurz die Headerdatei `Descriptor.h` als Ganzes an:

```
/* 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);
```

Die öffentlichen Informationen sind wirklich auf das aller Mindeste reduziert: die struct der abstrakten Oberklasse (hier in der Headerdatei wegen den public Funktionspointern), die Konstruktoren und die Prototypen der öffentlichen nicht-virtuellen Objekt- und Klassenmethoden. Interessant ist, dass in der Headerdatei die abstrakte Klasse `StreamSocket` *überhaupt nicht vorkommt*, obwohl sie die Oberklasse um das Attribut `fd` erweitert und Implementierungen der virtuellen Methoden angibt. Durch die starke Umsetzung des Geheimnisprinzips in diesem Beispiel werden alle privaten Attribute der Klassen durch die *opaque pointer* verdeckt, wodurch `fd` nicht nach außen bekannt wird. Die Implementierung der virtuellen Methoden werden ebenfalls nicht nach außen als Prototypen bekannt gegeben, sondern werden durch Füllen der Funktionspointer in `Descriptor` verfügbar.

2.3 Generizität

Um Instanzen beliebigen Typs in einer Datenhaltungsklasse zu speichern, kann man diese von einer gemeinsamen Vaterklasse ableiten, wodurch jedoch ein expliziter (und gefährlicher) downcast auf den ursprüngliche Klassentyp nach Entnehmen des Objekts aus der Datenhaltungsklasse nötig wird.

Eine bessere Möglichkeit bieten so genannte *generische* Klassen. Hierbei *parametrisiert* man Klassendefinitionen mit Typen: man erstellt “Schablonen” von Klassen, in denen an bestimmten Stellen bei Bedarf der verlangte Typ eingesetzt wird. Das Standardbeispiel ist der `Stack<T>` in dem Elemente von einem bei der Definition fest gewählten Typs `T` gepusht und gepopt werden können.

In plain C könnte man dieses Konzept mithilfe des Präprozessors durch Makros umsetzen. Diese Makros expandieren zu Funktionsdefintionen für die jeweils in den Argumenten angegebene Typen. Diese Methode funktioniert und ist total generisch, aber einfach nicht praktikabel für größere Projekte. Als Abschreckungsbeispiel ist rechts eine Implementation eines generischen Stacks in einem Präprozessormakro aufgeführt. Die ganzen \ am Rand und `##` sind natürlich notwendig.
Bekanntlich benutzt man stattdessen Stackfunktionen, die auf `void*` arbeiten. Dabei geht beim Einfügen eines Objekts seine Typinformation verloren, und sie muss beim Entfernen mit einem Cast wieder hinzugefügt werden. Echte generische Programmierung ist das nicht, es ähnelt mehr das in Java bis kürzlich üblichen Hochcasten auf `Object`, um die Objekte in Container einzufügen.

```
/* 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;                                                  \
}
```

3 Darstellung von C++ in Maschine

In diesem Abschnitt untersuchen wir wie C++ die oben in C ausgeführten Konzepte technisch umsetzt. Wir werden dabei auf einige andere Vorgehensweisen stoßen.

3.1 vtable

Um herauszukriegen wie C++ virtuelle Funktionen umsetzt, wollen wir ein ein kurzes C++ Programm schreiben und es dann untersuchen. Die links gezeigten Klassen sollen uns als Beispiel genügen: zwei virtuelle Funktionen und ein Attribut, das auf einen in einem Hexdump leicht erkennbaren Wert gesetzt wird. Wir drucken den von einer Instanz der Klassen belegten Speicher mit `fwrite()` aus.
Ein Speicherdump einer Instanz von `Uni` und `Earth` sieht folgendermaßen aus: (Adressen können sich ändern.)

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

Deutlich erkennt man, dass nicht wie erwartet die Variable `c` am Anfang der Speicherstruktur steht, sondern `0x080487e0`. Aus einem früheren Vortrag in RLP wissen wir das `0x08...` eine Adresse in unserem Code ist und wir gehen diese Stelle mit *objdump* suchen.

```
/* 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);
}
```

Mit `objdump -s VTable` finden wir die Stelle in der section `.rodata`:

```
Contents of section .rodata:
 80487a0 03000000 01000200 aaaaaaaa 00576861  .............Wha
```

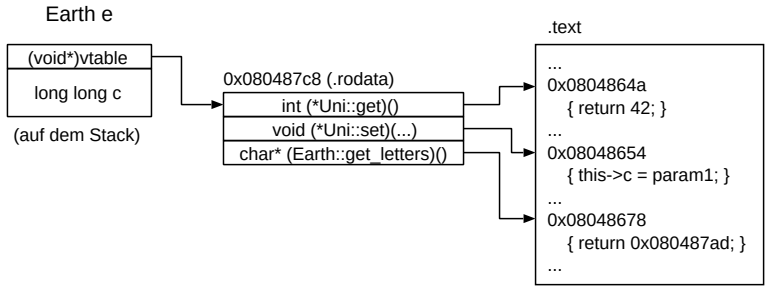
```
80487b0 74206973 20362074 696d6573 20393f00  t is 6 times 9?.
80487c0 00000000 f0870408 4a860408 54860408  ........J...T...
80487d0 78860408 00000000 00000000 e8870408  x...............
80487e0 4a860408 54860408 b8990408 03880408  J...T...........
80487f0 88990408 fc870408 e8870408 35456172  ............5Ear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

In dem obigen Hexdump sehen wir, dass an der Adresse `0x080487e0` die rot markierten Werte `0x0804864a`, `0x08048654` stehen. Wieder wissen wir, das sind Adressen im Codebereich. Ein weiterer `objdump -d VTable` führt uns zum Ziel:

```
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
```

An der Adresse `0x0804864a` steht also genau der Rumpf von `Uni::get`. Weiter unten finden wir bei `0x08048654` die Funktion `Uni::set`. Im Speicherabbild von `Earth` steht an erster Stelle die Adresse `0x040887e0`, die auf die blau markierten Codebereichsadressen zeigt. Hier erkennen wir mit `0x0804864a` und `0x8048654` die Funktionen der Oberklasse `Uni` wieder. Die in `Earth` neu hinzugekommene virtuelle Funktion `Earth::get_letters` steht mit `0x08048678` am Ende.

Man kann den *vtable* also im Diagramm mit Pfeilen als Zeiger dargestellt folgendermaßen aufzeichnen:



3.2 Überladung

In den klassischen Programmiersprachen wie C ist eine Funktion durch ihren Funktionsnamen eindeutig beschrieben. C++ bietet die Erweiterung, dass man seine Funktion nicht anders benennen muss, nur weil sie einen anderen Aufrufparametern hat. Funktionen werden anhand der Parametertypen, aber nicht anhand des Rückgabetyps unterschieden. Das folgende Beispiel zeigt zwei Funktionen, welche den gleichen Namen tragen, aber unterschiedliche Parameter haben. Der Compiler erkennt anhand der Parameter, welche Funktion mit welchem Aufruf zu verbinden ist.

```
void Anzeigen(int i) {
    printf("%d\n",i);
}
void Anzeigen(double f) {
    printf("%f\n",f);
}
int main()
{
    Anzeigen(12); // ruft die erste Funktion (12 ist int)
    Anzeigen(2.5); // ruft die zweite Funktion (2.5 ist float)
}
```

In einer Binary muss jedes Symbol eindeutig sein, daher kann auch jeder Funktionsname nur einmal verwendet werden. So muss der Compiler aus den Methodennamen und der Methodenparameterliste *global eindeutige Symbole* generieren. Dieser Vorgang wird Name-Mangling genannt.

Da der genaue Mangling-Vorgang von Compiler zu Compiler (MS, GNU) und sogar von Version zu Version verschieden ist, zeigen wir zuerst den prinzipiellen Vorgang:

3.3 Name-Mangling

Die folgenden Eigenschaften einer Funktion werden herangezogen, um diese eindeutig zu beschreiben:

1. der Funktionsname
2. der Klassennamen und ggf. `namespace`-Name
3. die Parameterliste
   (a) Die C++ internen Datentypen `int`, `long`, `short`, `char`, `long long` werden durch `i`, `l`, `s`, `c`, `x` gemangelt.
   (b) Unsigned Datentypen haben ein extra `U` vorangestellt.
   `unsigned char` wird z.B. als `Uc` abgebildet
   (c) Pointer werden durch ein vorangestelltes `P` gekennzeichnet und Referenzen durch `R`.
4. zusätzliche Informationen wie Callconventions (stdcall, fastcall, cdecl)

```
/* gcc compiler */

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 */
```

Alle gemangelten Symbole beginnen mit `_Z`. Mehrere aufeinanderfolgende Paare vom Typ (Länge, Name) werden in `N...E` eingeschlossen. Nach dem `_Z` stehen Namespace, Klassenname, Funktionsname jeweils im Format (Länge, Name). Anschliessend wird die Parameterliste angehängt.

`_Z5set_xP5Pointi`

1. `_Z` für gemangeltes Symbole
2. `5set_x` ist der Funktionsname der 5 Zeichen enthält
3. `P` gibt an das der erste Parameter ein Pointer ist
4. `5Point` gibt den Namen des Typs an, da dieser kein einfacher C++ Datentyp ist.
5. `i` ist der letzte Parameter, ein int.

`_ZN5Graph4Test5get_xEP5Pointi`

1. `_Z` für gemangeltes Symbole
2. `N...E` schliesst die folgende Reihe von Namen ein:
3. `5Graph4Test5get_x` Namespace, Klassenname, Funktionsname jeweils vom Typ (Länge, Name)
4. der Rest wie bei `P5Pointi`

3.4 Runtime-Type-Info

Mit dem Mechanismus der virtuellen Funktionen kann man sicherstellen, dass auch über Zeiger nur "objektgemässe" Funktionen aufgerufen werden. Die Typinformation über das Objekt wird dabei in Form des Zeigers auf die Sprungtabelle mitgeführt.

Mit Hilfe des Operators `typeid` kann man aber direkt den (dynamischen) Typ eines bestimmten Objekts abfragen (*RTTI* = run time type information). Die meisten Probleme, bei denen zur Laufzeit Typinformationen notwendig sind, kann und sollte man aber (implizit) eleganter durch virtuelle Funktionen lösen und RTTI nur einsetzen, wo diese nicht weiterhelfen.

Für RTTI ist vom Compiler ein ziemlicher Aufwand zu betreiben. Da zur Compilezeit nicht immer vorhergesehen werden kann, welche Typen in den `typeid`-Ausdrücken im Programm auftreten werden, muss er für jeden im Programm vorkommenden Typ eine Informations-Struktur anlegen. Deshalb ist meist eine spezielle Compiler-Option notwendig: bei älteren Versionen `-frtti` zum Aktivieren, bei neueren Versionen `-fno-rtti` zum Deaktivieren.

Ein Ausdruck der Form `typeid(object)` bzw. `typeid(typename)` ist vom Typ `type_info&`. Diese Klasse ist in der Header-Datei `typeinfo` definiert. Für jeden Typ wird ein Objekt dieser Klasse erzeugt und initialisiert. Mögliche Operationen sind Vergleiche mit `==` und `!=`, sowie das Zurückliefern des "Namens" des Typs mit `const char *type_info::name() const`. Das Format dieses Namens ist allerdings compilerabhängig.

Auf Konstantheit kann mit `typeid` nicht getestet werden. `==` und `!=` funktionieren auch mit den einfachen Typen. Allerdings liefert bei ihnen `name()` den Nullpointer zurück.

In der obigen Situation könnte man wie folgt feststellen, ob ein Shape-Pointer auf ein Rectangle-Objekt zeigt oder nicht:

```
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...?\n");
}
```

Die direkte Ausgabe mit `typeid(Rectangle).name()` liefert z.B. "9Rectangle". Um RTTI zur Verfügung zu stellen, muss ein Compiler je nach Art des Objekts verschiedene Schritte machen:

1. Wenn keine Pointer oder Referenzen beteiligt sind, ist der Typ zur Compilationszeit bekannt, und der Compiler setzt direkt einen Verweis auf das entsprechende `type_info`-Objekt ein.

2. Ist die Klasse nicht polymorph, kann die Typinformation auch bei Zeigern und Referenzen naturgemäss nur statisch sein, und der Compiler geht genauso vor wie bei 1. vor. Oben wäre dann `typeid(*sp) == typeid(Shape)`.

3. Es bleibt noch der Fall von Zeiger- oder Referenz-Zugriff auf polymorphe Klassen. In der Sprungtabelle *vtable* wird dazu als zusätzlicher Eintrag ein Zeiger *tptr* auf das passende `type_info`-Objekt angelegt. Es ist also kein zusätzlicher Speicheraufwand pro Objekt (nur pro Typ) erforderlich.

Wenn `typeid` auf den dereferenzierten Nullzeiger angewandt wird, wird eine `bad_typeid`-Exception ausgelöst.

Auch die (statische) Typinformation für Ausdrücke mit einfachen Typen ist abrufbar:

```
if (typeid(42L) == typeid(int)) cout << "int\n";
else if (typeid(42L) == typeid(long)) cout << "long\n";
else cout << "Der Compiler hats versaut.\n";
```