
Compilierung des Beispiels

- Letzte Stunde: Konzepte im Point-Beispiel
 - Erklärung für den Programmierer
 - Bedeutung zur Laufzeit
 - Definition in der JLS
- Heute: Implementierung in C
- Nächstes Mal: Realisierung im Compiler

Zwischenschritt: C als Ziel

- Wir verstehen nun den Quellsprache Java
- Es bleibt noch, auch die Zielsprache zu verstehen
- Erst danach: (Das ist ja wahrscheinlich euer erster Übersetzer!)
 - Vorstellung von Übersetzung entwickeln
 - Programm für Übersetzung schreiben

The Tools of Trade

- Wir brauchen nur einen kleinen Ausschnitt von C
 - int-Zahlen
 - Structs
 - Funktionen
 - Zeiger
 - Funktionszeiger
 - Sequenzausdrücke

⇒ Diese jetzt einmal einzeln kurz durchgehen

⇒ Erklärung nochmals bei Code-Erzeugung

Anweisungen und Ausdrücke

- Anweisungen und Ausdrücke i.w. wie in Java
- Arithmetische Berechnungen i.w. wie in Java
- Achtung: Lokale Variablen können nur zu Beginn einer Blockanweisung definiert werden.

```
{
  int i;
  i = 42 * 17;
  int j = i+1; // Syntaxfehler
  if (i > 0) {
    int k;    // ok
    ...
  }
}
```

Sequenzausdruck

- Ausdruck (e_1, \dots, e_n)
 - berechnet Ausdrücke e_1, \dots, e_n in der gegebenen Reihenfolge
 - verwirft die Ergebnisse von e_1, \dots, e_{n-1}
 - liefert das Ergebnis von e_n als Ergebnis
- Beispielanwendung: Funktionsaufruf $f(e_1, \dots, e_n)$, bei dem die Argumente von links nach rechts ausgewertet werden.
- Wähle temporäre Variablen t_1, \dots, t_n mit passendem Typ

$(t_1 = e_1, t_n = e_n, f(t_1, \dots, t_n))$

Structs

- Structs fassen mehrere Variablen zu einem Wert zusammen.
- Alle Felder eines Structs sind änderbar.
- Achtung: Nach der Struct-Deklaration muss ein Semikolon stehen!
- Achtung: Der Name allein ist **kein** Typ in C

```
struct Point { // Typdeklaration
    int x;
    int y;
}; // Semikolon hinter Deklaration
int main() { // Funktion
    struct Point p; // Typangabe mit struct Schlüsselwort
    ...
}
```

Funktionen und Variablen

- C-Funktionen haben Parameter, Rückgabe und lokale Variablen
- ⇒ Sehr ähnlich zu statischen (!) Methoden in Java

```
int fib(int n) {
    int prev = -1;
    int cur = 1;
    int i; // Laufvariable muss vor Schleife definiert werden
    for (i=1; i<=n; ++i) {
        int tmp = cur + prev;
        prev = cur;
        cur = tmp;
    }
    return cur;
}
```

Initialisierung von Structs

- Structs werden inialisiert, indem man in geschweiften Klammern Werte für ihre Felder angibt (in der Reihenfolge der Deklaration)

```
int f() {
    struct Point p = { 42, 127 };
    printf("x = %i, y = %i\n", p.x, p.y);
}
```

Ergibt: x = 42, y = 127

Structs sind Werte

- Achtung: Structs haben Wertsemantik
(Erinnerung: Java Objekte haben Referenzsemantik)

```
void move_nice_try(struct Point p, int dx, int dy) {
    p.x = p.x + dx;
    p.y = p.y + dy;
}
int main() {
    struct Point p = { 10, 20 };
    struct Point q;
    move_nice_try(p,1,2);
    // es bleibt p.x = 10; p.y = 20
    q = p;
    q.x = 0;
    // es bleibt p.x = 10; p.y = 20
}
```

Zeiger auf Structs

- Zugriff auf Struct über Zeiger: `.` hat höhere Präzedenz als `*`
- ⇒ `*p.x` wird geklammert als `*(p.x)`
- ⇒ Zuerst Feldzugriff auf `x`, dann Dereferenz → schreibe `(*p).x`
- Operator `->` ist Abkürzung

```
void move_by(struct Point *p, int dx, int dy) {
    p->x = p->x + dx;
    p->y = p->y + dy;
}
```

Zeiger

- Zeiger sind Werte, die auf Speicherstellen verweisen → Addr in Mini
- Typ `t*` ist Zeiger auf Speicherstelle vom Typ `t`
- Der Adressoperator `&` liefert Zeiger auf Variablen
- Der Dereferenzoperator `*` liefert den Wert, auf die ein Zeiger verweist
(→ ! in OCaml, load,store in Mini)
- 0 ist der Nullzeiger; er ist ungültig und kann nicht dereferenziert werden

```
void incr(int *i) {
    *i = *i + 1;
}
int main() {
    int i = 41;
    incr( &i);
    printf("i=%i\n", i); // ==> i=42
}
```

Der void-Zeiger

- Bisher sind alle Zeiger getypt: `t*` zeigt immer auf Speicherstellen, an denen ein `t` Wert gespeichert ist.
- Manchmal will man jedoch nur die reine Adresse
- Benutze `void *`
 - Keine Aussage über den Wert, der an der Adresse gespeichert ist
 - Dereferenz nicht erlaubt (Welchen Wert würde man erhalten?)

Casts

- Ausdruck $(T)e$ zwingt den Compiler, das Ergebnis von e als Wert mit Typ T zu betrachten
 - Konversionen auf primitiven Typen `float` ↔ `int` paßt Darstellung an
 - Zeiger `void*` ↔ `T*` für alle Typen T
 - Zusicherung, daß an einer Adresse ein bestimmter Wert steht
- Der C-Compiler ist gutgläubig bei Zeiger-Casts

```
float f;
int i = 42;
int *ip;
float *fp;
f = (float)i; // Anpassung der Darstellung
fp = &f;
ip = (int*)fp; // Zusicherung: ip zeigt auf einen int-Wert
printf("i=%i, *fp=%f, *ip=%i\n", i, *fp, *ip);
```

→ `i=42, *fp=42.000000, *ip=1109917696`

Funktionszeiger

- Funktionszeiger verweisen auf den Maschinencode von Funktionen.
- Funktionsaufruf funktioniert auch über Zeiger
- Adress- und Dereferenzoperator fügt der Compiler ein
- Syntax (Rückgabety) (*⟨Name: ⟩)(⟨Parametertypen⟩)

```
typedef void (*a_fun_ty)(int); // Syntax in Def. verstecken
void call_it(a_fun_ty a_f) {
    a_f(42);
}
void f(int i) { printf("f --> %i\n",i); }
void g(int i) { printf("g --> %i\n",i); }

int main() {
    call_it(f); // Funktionszeiger übergeben
    call_it(g);
}
```

Funktionszeiger als Rückgabe

- Funktionen könne auch Funktionszeiger als Ergebnis liefern

```
a_fun_ty get_one(int i) {
    if (i) return f;
    else return g;
}
int main() {
    call_it(get_one(0));
    call_it(get_one(3));
}
```

- Ohne Abkürzung ist die Syntax untragbar (Kernighan/Ritchie, §A8.6)

```
void (*get_me())(int) {
    return f;
}
```

Was ist der Unterschied zwischen
Funktionszeigern in C
und Funktionen in OCaml?

Definition und Deklaration

- Eine **Deklaration** gibt dem Compiler Informationen bekannt, beispielsweise
 - einen Typ (den neuen Typnamen und den Typ, für den er steht)
 - eine Variable (den Namen und Typ)
 - eine Funktion (den Funktionskopf)
- Eine **Definition** ist eine Deklaration, die Speicherplatz anlegt für
 - für eine Variable
 - für den Code einer Funktion
- Achtung: Variablendeklarationen sind immer auch -definitionen, es sei denn, sie werden mit `extern` eingeleitet.

```
extern int i;
```

Beispiel: point.h und point.c

point.h

```
#ifndef POINT_H
#define POINT_H
struct point {
    int x;
    int y;
};
void move_by(struct point *p, int dx, int dy);
#endif
```

point.c

```
#include "point.h"
void move_by(struct point *p, int dx, int dy) {
    p->x = p->x + dx;
    p->y = p->y + dy;
}
```

C-Dateien

- Eine C-Datei besteht aus einer Liste von Deklarationen und Definitionen
 - Typen
 - Variablen (sog. globale Variablen)
 - Funktionen
- Es gibt zwei Arten von Dateien
 - Headerdateien enthalten nur Deklarationen
 - Implementierungsdateien enthalten die Definitionen dazu
- Headerdateien werden mit `#include` eingelesen
- **Guards** verhindern mehrfaches Einlesen (und damit Compilerfehler)

```
#ifndef NAME_OF_HEADER_FILE
#define NAME_OF_HEADER_FILE
<Deklarationen der Headerdatei>
#endif
```

Compiler und Linker

- Die Übersetzung von C-Programmen erfolgt in zwei Schritten
 - Der Compiler übersetzt `.c`-Programme in Objektdateien (Endung: `.o`)
 - Der Linker löst Bezüge zwischen Objektdateien auf und erzeugt ein lauffähiges Programm
- ⇒ Die Header-Dateien sorgen nur dafür, dass der Compiler keine Fehlermeldungen bringt. Die erzeugten Objektdateien enthalten noch Referenzen auf undefinierte Namen.
- Der Linker sucht in allen übergebenen Objektdateien nach passenden Namen, er nimmt keine Rücksicht auf die Namen der Objektdateien selbst.
- ⇒ Die Aufteilung von Module M in $M.h$ und $M.c$ ist nur Konvention.

Beispiel: mainpoint.c und Übersetzung

```
mainpoint.c
#include "point.h"
int main() {
    struct Point p = { 10, 20 };
    move_by(&p,1,2);
}
```

- gcc -c point.c erzeugt point.o
- gcc -c mainpoint.c erzeugt mainpoint.o
- Beide Aufrufe lesen die Deklarationen in point.h ein
- gcc -o point point.o mainpoint.o erzeugt Programm point (-o steht für *output file*)

Übersetzung von Java-Konzepten nach C

Vorgehen für jedes Konzept

1. Zusammenfassung des Java-Konzeptes
 - a. Die Aspekte aus Sicht des Programmierers
 - b. Die Java Spezifikation
 - c. Mögliche Erweiterungen bedenken (z.B. Klasse → Vererbung)
2. Intuition: Was soll erreicht werden?
 - a. Wie sieht der C-Code aus, der wirklich abläuft?
 - b. Welchen Beitrag müssen die einzelnen Module leisten?
3. Implementierung in JC
 - a. Den Code für das Konzept aus den Compiler-Phasen durchgehen
 - b. Rückbezüge zum Konzept herstellen
 - c. Offene Detailfragen anhand der Implementierung klären

Abschluss C

- ✓ Wir haben den benötigten Sprachumfang von C in dieser einen Stunde durchgesprochen
- ✓ C ist eine sehr verhältnismäßig kleine Sprache
- ✓ Syntax ähnlich zu Java
- Neu sind nur
 - ✓ Sequenzausdrücke
 - ✓ Zeiger
 - ✓ Structs
 - ✓ Funktionszeiger

Konzept: Klasse

- Einführung eines neuen Typs
- Definition der Instanzen
 - Implementierung der Methoden
 - Definition der Instanzvariablen
- Objekte erzeugen durch Instanziierung
- ⇒ Konstruktion und Initialisierung
- Übersetzungseinheit: Eine Klasse wird bei javac eine .class-Datei

Grundidee Übersetzung von Klassen nach C

- Übersetzungseinheit → Pro Klasse eine Header- und eine C-Datei
- Instanzen → struct Typ mit einer Komponente pro Instanzvariable
- Implementierung von Methoden → Eine C-Funktion pro Methode
- Konstruktoren → Eine C-Funktion pro Konstruktor

⇒ Analog zum struct-Beispiel aus dem C-Teil

⇒ Zu Leisten:

- Rückbezug auf Konzept
- Details des Konzeptes betrachten

Beschreibung der Instanzen

- Alle Instanzen haben dieselben Variablen
- Fasse diese in C-Struct zusammen

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

Erinnerung: Point-Klasse

```
class Point {
    private int x;
    private int y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public void moveBy(int dx, int dy) {
        x = x + dx;
        y = y + dy;
    }
}
```

Variablen für Instanzen

- Objekte haben in Java Referenzsemantik
- In Variablen sind nur Zeiger auf Objekte gespeichert
- C besitzt Zeiger

Point p; → struct Point *p;

Feldzugriff

- Feldzugriff wird Zugriff auf struct-Komponente
 - this-Objekt: Brauchen Variable this in Methoden
 - Java-Konzept direkt durch C-Konzept ausgedrückt
- ⇒ Der C-Compiler kümmert sich um die Repräsentation im Speicher

```
this.x += dx → this->x += dx;
```

Methoden

- Methoden werden C-Funktionen
- Methoden können Instanzvariablen lesen und ändern
- Der Zugriff erfolgt über das this-Objekt
- Realisierung this-Objekt: Neuer erster Parameter

```
void moveBy(struct Point *this, int dx, int dy) {  
    this->x += dx;  
    this->y += dy;  
}
```

Methodenaufruf

- Methodenaufruf → C-Funktionsaufrufe
- Zielobjekt als erstes Argument übergeben → this
- $x.m(e_1, \dots, e_n)$ wird zu $m(x, e_1, \dots, e_n)$
- Hinweis: x kann i.a. ein komplexer Ausdruck sein
→ Ergebnis in temporärer Variable speichern
- Argumente von links nach rechts auswerten → Sequenz

```
p.moveBy(100-78, 89-67)  
→ (t=p, a1=100-78, a2=89-67, moveBy(t, a1, a2))
```

Konstruktoren

- Erinnerung: Instanziierung in 4 Schritten
 - Speicheranforderung für das neue Objekt
 - Auswertung der Konstruktorargumente
 - Aufruf des Konstruktors für das neue Objekt
 - Rückgabe des neuen Objekts
- ⇒ Der Konstruktor bekommt neues Objekt als this-Objekt

```
void ctor(struct Point *this, int x, int y) {  
    this->x = x;  
    this->y = y;  
}
```

Instanziierung

- Übersetze `new Point(42,18)`
- Die 4 Schritte nacheinander ausführen

⇒ Sequenzausdruck nutzen

```
( t = (struct Point*)malloc(sizeof(struct Point)),
  a1 = 42, a2 = 18,
  ctor(t,a1,a2),
  t )
```

C-Dateien für den ersten Versuch

```
----- point.h -----
#ifndef POINT_H
#define POINT_H
struct Point {
    int x;
    int y;
};
void ctor(struct Point *this, int x, int y);
void moveBy(struct Point *this, int dx, int dy);
#endif
```

```
----- point.c -----
#include "point.h"
void moveBy(struct Point *this, int dx, int dy) {
    this->x += dx;
    this->y += dy;
}
void ctor(struct Point *this, int x, int y) {
    this->x = x;
    this->y = y;
}
```

Die main-Methode

```
class Main {
    public static void main(String argv[]) {
        Point p = new Point(10,20);
        p.moveBy(1,2);
        System.out.println_int(p.getX());
        System.out.println_int(p.getY());
    }
}
```

- Programmstart bei C-Funktion `main()`
- Klasse `Main` wird in `Main.c` übersetzt

⇒ `main` muss `Main.main()` aufrufen

- `Main.main()` ist eine `static`-Methode

Einschub: Konzept Klassenmethode

§8.4.3.2 A method that is declared `static` is called a [class method](#). A class method is always invoked without reference to a particular object. An attempt to reference the current object using the keyword `this` or the keyword `super` in the body of a class method results in a compile-time error. [...]

- Klassenmethoden arbeiten nicht auf speziellen `this`-Objekt
- Das `this` kann im Rumpf gar nicht referenziert werden

⇒ Klassenmethoden brauchen keinen `this`-Parameter

- Methodendefinition wird einfacher
- Methodenaufruf wird einfacher

Die statische Methode Main.main()

- Parameter argv erst mal ignorieren, da Arrays unbehandelt
- Rumpf der Methode ist schon klar, bis auf System.out Zugriff

```
----- Main.c -----
void Main_main(void *argv) {
    struct Point *p = ...;
    moveBy(p,1,2);
    ...
}

----- main.c -----
int main(int argc, char **argv) {
    void *_argv = 0;
    // Verarbeitung von argv -> _argv
    ...
    // Aufruf der Main.main() Klassenmethode ohne this Parameter
    Main_main(_argv);
}
```

Auswertung: Erster Versuch für Konzept Klasse

- ✓ Einführung eines neuen Typs
- ✓ Definition der Instanzen
 - ✓ Implementierung der Methoden
 - ✓ Definition der Instanzvariablen
- ✓ Objekte erzeugen durch Instanziierung
- ✓ Konstruktion und Initialisierung
- ✓ Übersetzungseinheit: Eine Klasse wird eine .class-Datei
- ✓ Ansatz "Compiler hilft verstehen" erfolgreich: Details gesehen für
 - Die Schritte zur Instanziierung & Methodenaufruf
 - Klassenmethoden und -felder
 - this-Zugriff nur aus Nicht-Klassenmethoden

Einschub: Klassenvariablen

§8.3.1.1 If a field is declared static, there exists exactly one incarnation of the field, no matter how many instances (possibly zero) of the class may eventually be created. A static field, sometimes called a **class variable**, is incarnated when the class is initialized (§12.4).

- Eine Variable pro Klasse → brauchen kein konkretes Objekt für Zugriff
 - Wird beim Laden der Klasse angelegt → globale Variable in C
- ⇒ Zugriff einfach über den Namen der globalen Variable

```
----- System.c -----
struct PrintStream *System_out; // globale Variable

----- Main.c -----
// System.out.println_int(p.getX())
(th1 = System_out,
 a1 = (th2 = p, getX(th2)),
 println_int(th1,a1))
```

Auswertung: Erster Versuch für Konzept Klasse

- ✗ Bei Methodenaufruf
 - Laut JLS Bestimmung der aufgerufenen Methode zur Laufzeit
 - ⇒ Konzept: Objekte enthalten Methoden
- ⇒ Erweiterungen sind mit dieser Realisierung nicht möglich
 - Abstrakte Methoden: Welcher Code soll dabei ausgeführt werden?
 - Vererbung & Polymorphie: Das Laufzeit-Objekt entscheidet, welcher Code ausgeführt wird
 - (Erinnerung: Shape und ShapeEditor)
- Namensauflösung nicht präzise genug
 - Verschiedene Klassen haben getrennte Namensräumen
 - Überladene Konstruktoren & Methoden brauchen verschiedene Namen
- Überhaupt kein Typcheck durchgeführt