Namenlose Daten und dynamische Speicherbelegung

Zeiger und dynamische Speicherbelegung für **namenlose Daten** sind eng verwandte Konzepte in imperativen Programmiersprachen.

Bisher haben wir nur **deklarierte Daten** betrachtet:
In der Deklaration wird ein Name für das Datum angegeben und diesem Namen wird statisch eine (Relativ-)Adresse zugeordnet.

Mit Zeigern kann man auf namenlose Daten zugreifen und zur Laufzeit wachsende und schrumpfende verkettete Strukturen realisieren, wobei die einzelnen **Daten** nicht durch eine Deklaration, sondern **durch die Ausführung eines Befehls erzeugt** werden.

Namenlose Daten und dynamische Speicherbelegung

Die Semantik von C legt die **Lebensdauer** dynamisch erzeugter Daten nicht sehr präzise fest. Natürlich kann der durch ein solches Datum belegte Speicher schon vor dem Ende der Lebensdauer wieder freigegeben werden, wenn das laufende Programm auf das Datum nicht mehr zugreifen kann.

Der Prozess der Freigabe von Speicher, der von unerreichbaren Daten belegt ist, heißt **automatische Speicherbereinigung** (*garbage collection*).

Die kellerartige Speicherbelegung passt nicht zur Lebensdauer dynamisch erzeugter Daten.

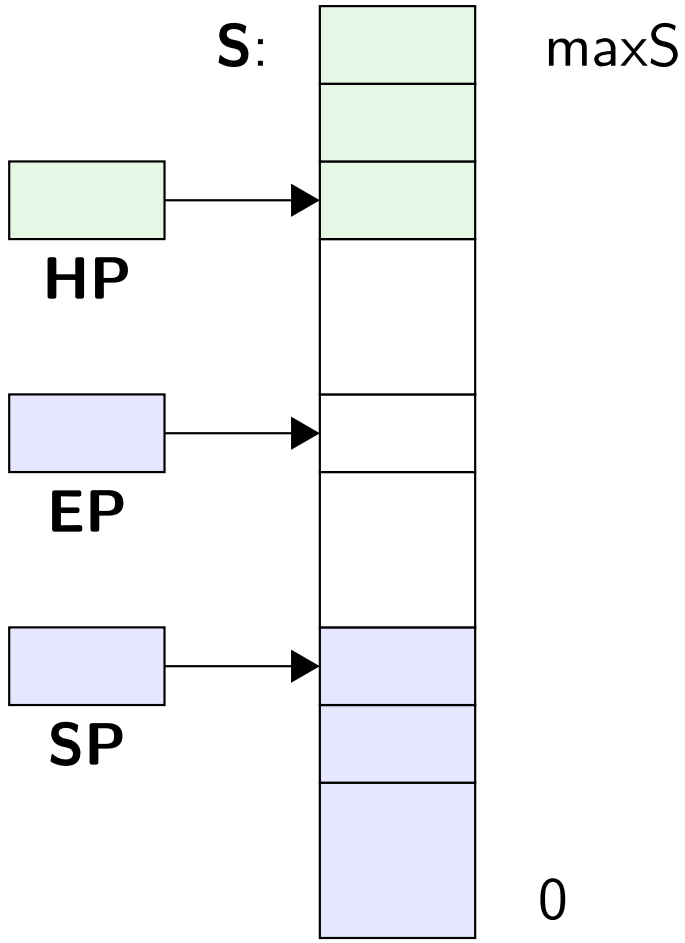
Deshalb werden dynamisch erzeugte Daten nicht im Keller, sondern in einem anderen Speicherbereich, der **Halde** (*heap*), untergebracht.
Ein Register der C-Maschine, der **Haldenzeiger HP** (*heap pointer*) zeigt stets auf die unterste belegte Zelle der Halde.

Namenlose Daten und dynamische Speicherbelegung

Um den physisch vorhandenen Speicher gut auszunutzen, definieren wir keine festen Teilbereiche für Keller und Halde, sondern belegen das untere Ende des Datenspeichers mit dem Keller und das obere Ende mit der Halde.

Keller und Halde wachsen aufeinander zu, dürfen sich aber nicht überlappen.

Eine Kollision kann durch Erhöhung von SP oder durch Verringerung von HP verursacht werden.



Der Datenspeicher der C-Maschine mit Halde (grün) und Keller (blau).

Namenlose Daten und dynamische Speicherbelegung

Um einen **Kellerüberlauf** (*stack overflow*) festzustellen, vergleichen wir *EP* jetzt nicht mehr mit *maxS*, sondern mit *HP*.

Die Abkürzung **testoverflow** wird daher endgültig festgelegt als:
testoverflow $\equiv$ **if** $EP \geq HP$ **then error**(„Stack Overflow“);

Zeigeroperationen

Was kann man mit Zeigern tun?

- Zeiger **erzeugen**, d.h. Zeiger auf Speicherzellen setzen;
- Zeiger **dereferenzieren**, d.h. durch Zeiger auf die Werte von Speicherzellen zugreifen.

Technisch ist ein Zeiger nichts anderes als eine Speicheradresse.

In C gibt es zwei Arten, Zeiger zu erzeugen:

- durch einen Aufruf der Bibliotheksfunktion **malloc** oder
- durch die Anwendung des **Adressoperator**s &.

Zeigeroperationen

Ein Aufruf **malloc**(e) berechnet den Wert m des Ausdrucks e und liefert einen Verweis auf die unterste Zelle eines neuen Speicherabschnitts der Größe m zurück.

$\text{code}^{\rho}_{W}(\textbf{malloc}(e)) = \text{code}^{\rho}_{W}\ e,\ \textbf{new}$

Ein Aufruf der Funktion **malloc** schlägt niemals fehl.
Selbst wenn nicht mehr genügend Platz zur Verfügung steht, liefert der Aufruf eine Adresse zurück: die Adresse 0.

Ein Programmierer muss deshalb stets diesen Rückgabewert auf 0 testen, um diese Fehlersituation zu erkennen.

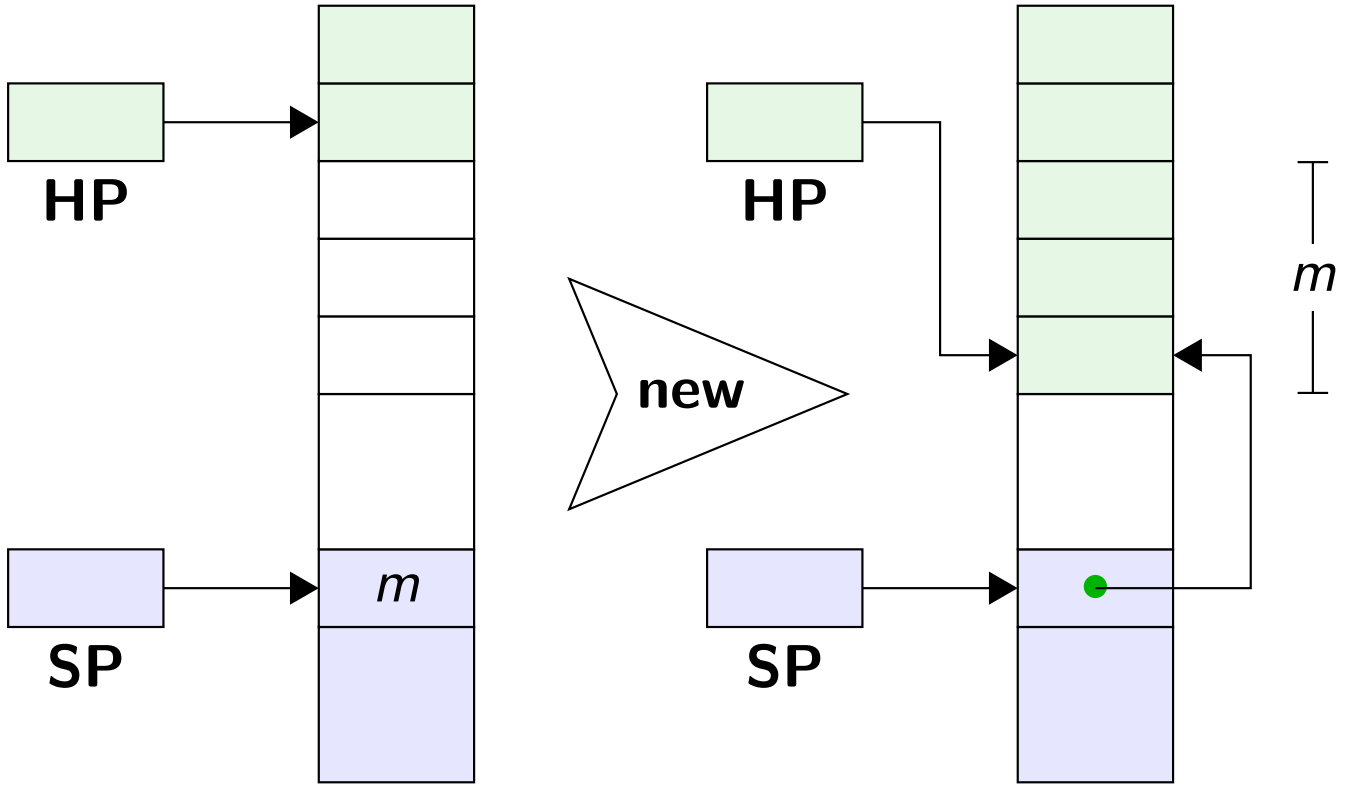
Befehl new der virtuellen Maschine

Ein neues Datum auf der Halde wird mit dem Befehl **new** erzeugt.

Der Befehl **new** erwartet oben auf dem Keller die Größe des neuen Datums und liefert die Anfangsadresse des dafür zur Verfügung gestellten Speicherbereichs zurück.

Vorher muss überprüft werden, ob dafür genügend freier Speicherplatz zur Verfügung steht.
Ist nicht genügend Platz vorhanden, liefert der Befehl **new** den Wert 0 zurück (besser wäre eine Fehlermeldung **Haldenüberlauf**).

Befehl new



if $HP{-}S[SP] > EP$ **then** $\{$
$HP \leftarrow HP{-}S[SP]; \quad S[SP] \leftarrow HP; \}$
else $S[SP] \leftarrow 0;$

Der Befehl **new**.

Zeigeroperationen

Die Anwendung $\&e$ des **Adressoperators** auf einen Ausdruck e liefert einen Zeiger auf das Speicherobjekt von e, d.h. auf dasjenige, das an der Anfangsadresse von e beginnt und den Typ von e besitzt.

Der Wert des Ausdrucks $\&e$ ist daher gerade die Adresse des Ausdrucks e:
$\text{code}_W^\rho(\&e) = \text{code}_A^\rho\ e$

Sei e ein Ausdruck, der zu dem Wert eines Zeigers p ausgewertet wird. Dieser Zeigerwert ist dann die Adresse des Datums, auf das der Zeiger zeigt.
Dieses Datum erhalten wir, wenn wir den Zeiger p **dereferenzieren**, d.h. den Präfix-Operator $*$ anwenden.

D.h., der Wert von e ist die Adresse von $*e$:
$\text{code}_A^\rho(*e) = \text{code}_W^\rho\ e$

Ein etwas größeres Beispiel

Beispiel: Für eine Deklaration:
struct t { **int** $a[7]$; **struct** t $*b$; };
int i, j;
struct t $*pt$;
wollen wir den Ausdruck $e' \equiv ((pt \to b) \to a)[i{+}1]$ übersetzen.

Der Operator $\to$ ist eine Abkürzung für eine Dereferenzierung, gefolgt von einer Selektion: $e \to c \;\equiv\; (*e).c$

Falls e den Typ $t*$ hat für eine Struktur t mit einer Komponente c, ergibt sich mit $m = \textbf{offsets}(t, c)$:
$\text{code}^{\rho}_{A}(e \to c) = \text{code}^{\rho}_{A}((*e).c) = \text{code}^{\rho}_{A}\ (*e)$, **loadc** m, **add**
$= \text{code}^{\rho}_{W}\ e$, **loadc** m, **add**

Im Beispiel gilt: $\textbf{offsets}(t, a) = 0$ und $\textbf{offsets}(t, b) = 7$.
Die Adressumgebung sei $\rho = \{(G, i) \mapsto 1,\ (G, j) \mapsto 2,\ (G, pt) \mapsto 3\}$.

Ein etwas größeres Beispiel

Im Beispiel ist **offsets**$(t, a) = 0$ und **offsets**$(t, b) = 7$.
Die Adressumgebung sei $\rho = \{(G, i) \mapsto 1,\ (G, j) \mapsto 2,\ (G, pt) \mapsto 3\}$.

Dann ergibt sich für den Ausdruck e':
$\text{code}_A^\rho\ ((pt \to b) \to a)[i{+}1]$
$= \text{code}_A^\rho((pt \to b) \to a), \text{code}_W^\rho(i{+}1),$ **loadc** 1, **mul**, **add**
$= \text{code}_A^\rho((pt \to b) \to a),$ **loada** 1, **loadc** 1, **add**, **loadc** 1, **mul**, **add**

Dabei ist:
$\text{code}_A^\rho((pt \to b) \to a) =$
$\text{code}_W^\rho(pt \to b),$ **loadc** 0, **add**
$=$ **loada** 3, **loadc** 7, **add**, **load**, **loadc** 0, **add**

Insgesamt erhalten wir:
loada 3, **loadc** 7, **add**, **load**, **loadc** 0, **add**, **loada** 1, **loadc** 1, **add**, **loadc** 1, **mul**, **add**

Ein etwas größeres Beispiel

Betrachten wir diese Befehlsfolge, fällt uns zweierlei auf.

- Ohne systematische Ableitung wären wir kaum so leicht auf diese Folge gekommen.
- Aber es bleibt Raum für diverse **Optimierungsmöglichkeiten**:
  Wir können die Addition von 0 einsparen und die Multiplikation mit 1.

  loada 3, **loadc** 7, **add**, **load**, ~~**loadc** 0, **add**~~, **loada** 1, **loadc** 1, **add**, ~~**loadc** 1, **mul**~~, **add**

  Das hätten wir schon bei der Codeerzeugung berücksichtigen können durch Fallunterscheidungen in den Übersetzungsschemata.

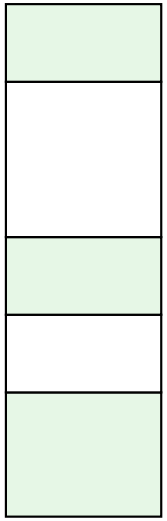
  Aus Übersichtlichkeit haben wir darauf verzichtet und vertrauen stattdessen auf einen Postpass-Optimierer, der solche lokalen Codeverbesserungen gesondert durchführt.

Speicherfreigabe

Jetzt bleibt nur noch die **explizite Freigabe** von Speicherbereichen zu regeln.

Die Freigabe eines Speicherblocks, auf den ein Zeiger zeigt, ist problematisch, weil es weitere Zeiger auf diesen Speicherbereich geben kann, die durch die Freigabe zu **hängenden Zeigern** (*dangling references*) würden.

Trotzdem könnte nach einigen (korrekten!) Freigaben die Halde so **fragmentiert** aussehen.
Die freien Speicherabschnitte können nach einiger Zeit sehr unregelmäßig über die Halde verstreut sein.



Die Halde nach Freigabe einiger Blöcke (grün = belegt, weiß = frei).

Speicherfreigabe

Es gibt eine Reihe von Algorithmen zur automatischen Speicherbereinigung.

In jedem Fall benötigt das Laufzeit-System weitere Datenstrukturen, deren Verwaltung Aufrufe der Funktionen **malloc** oder **free** verteuert.

In unserem minimalen Compiler tun wir bei einer Speicherfreigabe – nichts! Diese Implementierung ist korrekt, aber natürlich nicht speicheroptimal.

Wir übersetzen also:

$$\text{code}^{\rho}(\textbf{free}(e);) = \text{code}^{\rho}_{W}\ e,\ \textbf{alloc}\ -|e|$$

Software aus Standardbausteinen

Softwaresysteme werden immer größer und komplexer.
Entwicklung dieser Systeme soll effizienter und transparenter werden.

Eine Hoffnung besteht darin, auch Softwaresysteme
–wie Hardwaresysteme und Produkte wie Autos, Waschmaschinen–
aus vorgefertigten Standardbausteinen zusammenzusetzen.

Dieser Hoffnung versucht man u.a. durch folgende Ideen näherzukommen:

- Modularisierung,
- Wiederverwendbarkeit von Modulen,
- Erweiterbarkeit von Modulen und
- Abstraktion.

Objektorientierte Sprachen bieten hier neue Möglichkeiten.

Konzepte objektorientierter Sprachen

Die wichtigsten Konzepte objektorientierter Sprachen sind:

- Als weitere Modularisierungseinheit gibt es **Objektklassen**. Objektklassen kapseln Daten und Funktionen auf diesen Daten.
- Das **Vererbungskonzept** vereinfacht die Erweiterung oder Abänderung von Objektklassen.
- Das Typsystem nutzt das Vererbungskonzept:
  Erbende Klassen werden **Teiltypen** der Basisklassen;
  ihre Objekte können an (fast) allen Stellen benutzt werden, an denen Objekte der Basisklasse zulässig sind.
- **Vererbungshierarchien** führen unterschiedliche Abstraktionsebenen in Programme ein. Dies erlaubt, an verschiedenen Stellen innerhalb eines Programms oder Systems auf unterschiedlichen Abstraktionsstufen zu arbeiten.

Konzepte objektorientierter Sprachen

- **Abstrakte Klassen** können in Spezifikationen verwendet und durch schrittweise Vererbung verfeinert und dann implementiert werden. Der Übergang zwischen Spezifikation, Entwurf und Realisierung wird dadurch fließend.
- **Informationskapselung** trennt die abstrakte Sicht auf die Klassenbedeutung von der konkreten Sicht auf ihre Implementierung.
- **Generizität** erlaubt, Klassendefinitionen zu parametrisieren. Dadurch können Algorithmen und zugehörige Datenstrukturen wie Listen, Keller, Warteschlangen und Mengen unabhängig vom Datentyp ihrer Elemente implementiert werden.

Objekte

Imperative Programmiersprachen bieten **funktionale Abstraktion**:
Eine komplexe Berechnung wird in Funktion / Prozedur „verpackt“.

Für Aufgaben, die komplexe Datenstrukturen erfordern, sollte auch eine **Datenabstraktion** möglich sein, also die Kapselung der Datenstrukturen und der auf diesen Strukturen arbeitenden Funktionen zu einer Einheit.

Objekte

Grundkonzept objektorientierter Sprachen ist das **Objekt**.

Merkmale eines Objekts sind seine

- **Attribute** (lokale Daten) und
- **Objekt-Methoden** (darauf anwendbare Funktionen).

Die Werte der Attribute eines Objekts bestimmen seinen Objektzustand.

Die Werte der Attribute eines Objekts werden nur(!) durch den Aufruf von Methoden dieses Objekts geändert.

Die wichtigste Basisoperation objektorientierter Sprachen ist die **Aktivierung einer Methode** f für ein Objekt o (geschrieben $o.f$).

Innerhalb von f kann man mit dem Schlüsselwort **this** auf dieses Objekt o Bezug nehmen.

Objektklassen

Inkonsistenzen und Fehler in Programmen müssen frühzeitig und zuverlässig erkannt werden.
Übersetzer können (statische) Typinformation zur Überprüfung nutzen.

Ein **Typ** steht für eine Menge zulässiger Werte und bestimmt, welche Operationen auf diese Werte angewendet werden dürfen.

Übersetzer können Typinformation nutzen, um

- Inkonsistenzen zu erkennen,
- Mehrdeutigkeiten in der Operatorverwendung aufzulösen,
- automatisch Typkonversionen einzufügen und
- effizienteren Code zu erzeugen.

Nicht alle objektorientierten Sprachen benutzen statische Typisierung, z.B. nicht Smalltalk-80.
Aber Sprachen wie C++, Java oder C# erweitern die Typkonzepte von imperativen Sprachen wie Pascal oder C.
Ihre Typen heißen **Objektklassen**, kurz: **Klassen**.

Objektklassen

Eine Objektklasse legt Attribute und Methoden fest, die ein Objekt mindestens haben muss, um zu dieser Klasse zu gehören.

Für Attribute wird ihr Typ und für Methoden ihr Prototyp (Typen für Parameter und Rückgabewert) vorgegeben.
In einigen objektorientierten Sprachen (z.B. Eiffel) kann die Semantik der Methode genauer festgelegt werden, z.B. durch **Voraussetzung** (*precondition*) und **Effekt** (*postcondition*).

Häufig definiert die Klasse auch die Methoden; diese Definitionen können jedoch evtl. überschrieben werden.
Außerdem können u.U. Objekte der Klasse angehören, die neben den geforderten Merkmalen noch zusätzliche besitzen.

Die Objektklasse realisiert Datenabstraktion in objektorientierten Sprachen.
Sie fungiert als Generator für Objekte, für die **Instanzen** dieser Klasse.

Vererbung

Vererbung (*inheritance*) ist die Übernahme aller Merkmale einer Klasse K in eine neue Klasse U (ohne Code zu kopieren!).

Die Klasse U kann zusätzliche Merkmale definieren und
von K geerbte Methoden evtl. überschreiben.

Erbt U von K, dann heißt
U eine von K **abgeleitete Klasse** oder **Unterklasse** von K;
K heißt **Basisklasse** oder **Oberklasse** von U.

Die Vererbung vereinfacht Erweiterungen und Variationen entscheidend.
Durch Bildung von **Vererbungshierarchien** können Klassenbibliotheken strukturiert und Abstraktionsstufen gebildet werden.

Vererbung erleichtert, Teile einer bestehenden Implementierung zu nutzen, zu erweitern und bei Bedarf lokal –durch Überschreiben einzelner Methoden– anzupassen.

Abstrakte Klassen

Zusätzlich können wir **abstrakte Klassen** definieren.

Abstrakte Klassen sind Klassen, die undefinierte Methoden enthalten.

Sie besitzen keine eigenen Instanzen, das heißt alle ihre Objekte stammen aus echten Unterklassen.

Programmiersprachen erhalten so ein Äquivalent für abstrakte Begriffe und unterschiedliche Abstraktionsstufen in natürlichen Sprachen.

Was auf einer höheren Abstraktionsstufe formuliert werden kann, hat einen größeren Anwendungsbereich und ist öfter wiederverwendbar.

Teiltypen

Typisierte objektorientierte Sprachen besitzen daher eine Vererbungshierarchie in ihrem Typsystem.

Erbt eine Klasse U (öffentlich) von der Klasse K, dann ist der U zugeordnete Typ ein **Teiltyp** des Typs zu K.

Jedes Objekt eines Teiltyps ist automatisch auch Element der Oberklasse; eine erbende Klasse U wird **Unterklasse** der beerbten Klasse K.

Daher dürfen an einer Eingabeposition (Funktionsparameter, rechte Seiten von Zuweisungen) oder als Funktionsrückgabewert Objekte eines beliebigen Teiltyps des angegebenen Typs auftreten (**Teiltypregel**).

Objekte oder Verweise auf Objekte?

Dieses nützliche Prinzip hat aber Konsequenzen:

Objekte der Unterklassen von K können über zusätzliche Merkmale verfügen und brauchen dann mehr Speicherplatz als K-Objekte.

- Nutzt die Programmiersprache Verweise auf Objekte statt der Objekte (z.B. Eiffel, Java), so ist die Teiltypregel problemlos anwendbar: Verweise haben stets die gleiche Speichergröße.
- Sonst muss die Sprache genau zwischen Objekten und Verweisen auf Objekte unterscheiden:
  C++ ruft deshalb für Parameter, dessen Typ eine Unterklasse U ist, einen *copy*-Konstruktor der Klasse U auf, der den Typ anpasst, indem er die zusätzlichen Merkmale weglässt.

unmittelbare Objekte und unmittelbare Typen

Wir nennen die Objekte von K, die nicht auch Objekte einer echten Unterklasse sind, **unmittelbare Objekte** von K.
K heißt **unmittelbarer Typ** der unmittelbaren K-Objekte.

Damit besitzt jedes Objekt einen eindeutig bestimmten unmittelbaren Typ: den in der Hierarchie tiefsten Typ, zu dem das Objekt gehört.
Es ist darüber hinaus Element jeder Oberklasse seines unmittelbaren Typs.

Wegen der Teiltypregel akzeptieren Methoden und Funktionen in objektorientierten Sprachen auf einer Parameterposition (Verweise auf) Objekte von verschiedenen unmittelbaren Typen und damit von unterschiedlicher Struktur.
Dies ist eine Form von **Polymorphie**.

dynamische Auswahl von Methoden

Da Klassen eine geerbte Methode überschreiben dürfen,
richtet sich die bei einem Methodenaufruf $e.f(\ldots)$ aufgerufene Methode f nach dem unmittelbaren Typ des Objekts o, zu dem sich der Ausdruck e zur Laufzeit auswertet.

Damit muss der Übersetzer Code für die Aktivierung einer Methode erzeugen, die er zum Zeitpunkt der Übersetzung u.U. noch nicht kennt.
Diese Auswahl der Methoden zur Laufzeit (*dynamic lookup*) ist offenbar eine Form von **dynamischer Bindung**.

Diese **dynamische Auswahl der Methoden** bewirkt, dass das Objekt den auszuführenden Code (für eine Methode) bestimmt und damit, wie es auf einen Methodenaufruf („Nachricht“) reagiert.
Die gleiche Nachricht an unterschiedliche Objekte kann daher zu unterschiedlichen Ergebnissen („Antworten“) führen.

Generizität

Streng typisierte Sprachen zwingen häufig zu einer Reimplementierung der gleichen Funktion für verschiedene Typen, etwa von den Parametern. Diese mehrfachen Funktionsinstanzen erschweren die Implementierung, machen Programme unübersichtlicher und ihre Wartung aufwändig.

Das auf Vererbung beruhende Typkonzept objektorientierter Sprachen erspart oft eine Vervielfachung von Funktionsimplementierungen.

Für eine wichtige Problemklasse führt Vererbung allein aber nicht zu eleganten Lösungen.

Die Implementierung von **Datenstrukturen für Behälter** wie Listen, Keller oder Wartenschlangen haben einen natürlichen Parameter: den Typ ihres Inhalts.

Generizität

Durch **Generizität** (*genericity*) können wir für solche Datenstrukturen und ihre Methoden eine Mehrfachimplementierung vermeiden.
Funktions- und **Typdefinitionen** können **parametrisiert** werden;
als Parameter sind selbst wieder (evtl. eingeschränkte) Typen zugelassen.

Z.B. beschreibt *eine* Definition für die parametrisierte Klasse **list**<t>
Listen mit beliebigem Elementtyp t.
Listen mit spezifischem Typ werden durch **Instantiierung** der generischen Klasse erzeugt:
list<**int**> bezeichnet eine Liste, deren Elemente vom Typ **int** sind.

Generizität wird von einigen objektorientierten Sprachen (C++, Java) unterstützt; sie ist aber keine Erfindung objektorientierter Sprachen.
Generizität ist ein wesentliches Merkmal moderner funktionaler Programmiersprachen (OCaml, Haskell). Sogar die imperative Sprache Ada hatte bereits ein sehr ausgefeiltes Konzept von Generizität.

Informationskapselung

Die meisten objektorientierten Sprachen stellen Konstrukte zur Verfügung, mit denen die Merkmale einer Klasse als **privat** oder **öffentlich** klassifiziert werden können.

Private Merkmale sind in bestimmten Kontexten unsichtbar oder zumindest nicht zugreifbar.

Manche objektorientierte Sprachen unterscheiden verschiedene **Sichtbarkeit**skontexte, z.B.

- innerhalb der Klasse,
- in abgeleiteten Klassen,
- in fremden Klassen,
- in bestimmten Klassen.

Sprachkonstrukte oder Regeln können festlegen, in welchen Kontexten welche Merkmale sichtbar bzw. lesbar/schreibbar oder aufrufbar sind.

Informationskapselung

Diese Informationskapselung ermöglicht eine **Abstraktion** (*abstraction*) vom internen Aufbau der Objekte.

Üblicherweise existiert für Objekte eine **Schnittstelle** aus öffentlichen Funktionen, über die die versteckten internen Daten des Objekts gelesen und verändert werden können.

Der Übersetzer realisiert solche Konstrukte einfach, indem er unsichtbare Namen nicht in die Adressumgebung aufnimmt.

Wir gehen daher nicht weiter auf die Informationskapselung ein, obwohl sie wichtig ist für eine klare Trennung zwischen der abstrakten Sicht der Klassenbedeutung und der konkreten Sicht ihrer Implementierung.

Beispiel: Informationskapselung und Programmstruktur

Die Informationskapselung hat Auswirkungen auf die **Programmstruktur**.

Den Unterschied zwischen einer prozeduralen oder funktionalen Sprache und einer objektorientierten Sprache sieht man an folgendem Beispiel.

Beispiel

Man möchte für alle Angehörigen der Universität Informationen darstellen bzw. das Gehalt auszahlen können.

Beispiel: Informationskapselung und Programmstruktur

Beispiel (Funktionen für Uni-Angehörige: prozedural)

```
info(x) =
  case type(x) of
    Professor:   ["zeige Info über Professor"];
    Mitarbeiter: ["zeige Info über Mitarbeiter"];
    Hilfskraft:  ["zeige Info über Hilfskraft"];
  end;

bezahle(x) =
  case type(x) of
    Professor:   ["zahle Professor"];
    Mitarbeiter: ["zahle Mitarbeiter weniger"];
    Hilfskraft:  ["zahle Hilfskraft viel weniger"];
  end;
```

Beispiel: Informationskapselung und Programmstruktur

In einem objektorientierten Programm gibt es Klassen für Professoren, Mitarbeiter und Hilfskräfte und jede Klasse enthält die Methoden `info` und `bezahle`.

Beispiel (Funktionen für Uni-Angehörige: objekt-orientiert)

```
class Professor ≡
  info    = ["zeige Info über Professor"];
  bezahle = ["zahle Professor"];

class Mitarbeiter ≡
  info    = ["zeige Info über Mitarbeiter"];
  bezahle = ["zahle Mitarbeiter weniger"];

class Hilfskraft ≡
  info    = ["zeige Info über Hilfskraft"];
  bezahle = ["zahle Hilfskraft viel weniger"];
```

Beispiel: Informationskapselung und Programmstruktur

Beispiel (Funktionen für Uni-Angehörige: Vergleich)

Operation	Professor	Mitarbeiter	Hilfskraft
`info`	`Professor.info`	`Mitarbeiter.info`	`Hilfskraft.info`
`bezahle`	`Professor.bezahle`	`Mitarbeiter.bezahle`	`Hilfskraft.bezahle`

In konventionellen Programmiersprachen wird der Code zeilenweise in einer Funktion gruppiert, die auf allen Arten von Daten arbeitet.

In objektorientierten Programmiersprachen wird der Code spaltenweise gebündelt, in dem die einzelnen Funktionsteile mit den Daten, auf denen sie arbeiten sollen, gruppiert werden.

Beispiel: Informationskapselung und Programmstruktur

Beispiel (Funktionen für Uni-Angehörige: abstrakte Klasse)

In der funktionsorientierten Organisation konventioneller Programmiersprachen kann man leicht neue Operationen hinzufügen, z.B. `zahle_Weihnachtsgeld`, aber für einen neuen Datentyp (z.B. Junior-Professor) muss jede Funktion geändert werden.

In der datenorientierten Organisation objektorientierter Sprachen kann man einen neuen Datentyp leicht als neue Klasse hinzufügen, dagegen muss für eine neue Operation jede vorhandene Klasse geändert werden.

In objektorientierten Sprachen könnte man eine **abstrakte Oberklasse** Uni-Angehörige einführen und am Zahltag eine einfache Schleife über alle Elemente dieser Klasse (ohne Fallunterscheidung!) mit der Methode `bezahle` ausführen, da die Auswahl der Methode ja dynamisch nach dem unmittelbaren Typ des Objekts geschieht.

Eine objektorientierte Erweiterung von C

Wir erweitern jetzt unsere Implementierung von C auf Klassen, Objekte und ihre Merkmale.

Als Beispiel für eine objektorientierte Sprache wählen wir eine einfache Teilmenge von C++.

Klassen werden aufgefasst als Erweiterungen von Verbundtypen;
sie enthalten

- **Attribute**, das sind Datenfelder, und
- **Methoden**, die auch
  als **virtual** deklariert (und dann überschrieben) werden können, sowie
- **Konstruktoren**, die neu angelegte Objekte initialisieren.

Beispiel: Klasse **list**

```
class list {
        int info;
        list * next;
        list (int x) {info ← x; next ← null;}
        virtual int last() {
                if next = null then return info;
                else return next→last();
                }
};
```

Die Klasse **list** hat zwei Attribute *info* und *next* vom Typ **int** bzw. **list** $*$. Ein Konstruktor initialisiert neue Listenobjekte und eine Methode *last* liefert den Inhalt des Attributs *info* des letzten über *next*-Verweise erreichbaren **list**-Objekts zurück. Diese Methode ist als **virtual** gekennzeichnet; sie darf daher in Unterklassen redefiniert werden.

Eine objektorientierte Erweiterung von C

In Java liegen alle Objekte auf der Halde.

In C++ dürfen Objekte –wie Verbunde– auch direkt auf dem Keller angelegt werden.

Zur Vereinfachung verzichten wir auf Angaben zur Sichtbarkeit.
Um ein lauffähiges C++-Programm zu erhalten, könnten wir z.B. alle Merkmale der Klasse als **public** deklarieren.

Speicherorganisation für Objekte

Bei der Implementierung wollen wir nur Dinge im Objekt selbst anlegen, die sich von Objekt zu Objekt unterscheiden.

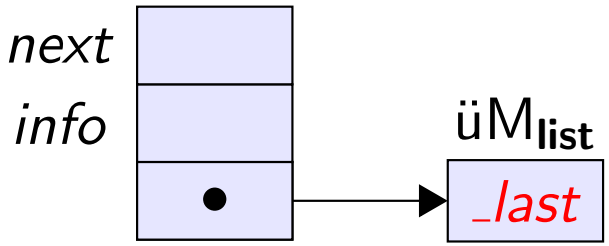
Nicht-überschreibbare Methoden ergeben sich aus dem Typ des Objekts und werden bei der Klasse (nicht bei jedem Objekt) abgespeichert.

Adressen von Attributen und überschreibbaren Methoden können aber nicht immer zur Übersetzungszeit ermittelt werden.

- Attribute sollen in allen Unterklassen gleich adressiert werden. Neue Attribute einer Unterklasse erhalten daher stets eine größere Adresse als die Attribute ihrer Oberklasse.
- Die zutreffende Implementierung von überschreibbaren Methoden ergibt sich erst zur Laufzeit aus dem unmittelbaren Typ des Objekts. Für jede Klasse K wird eine Tabelle üM_K angelegt, die alle **Anfangsadressen ihrer überschreibbaren Methoden** enthält. Alle unmittelbaren K-Objekte erhalten dann (an Relativadresse 0) einen Verweis auf diese Tabelle üM_K.

Ein Objekt der Klasse **list**

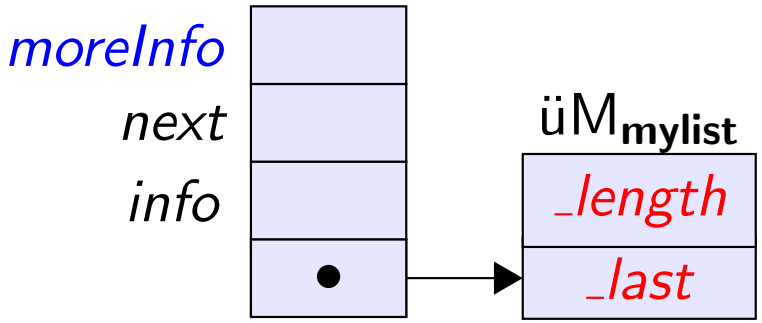
```
class list {
    int info;
    list * next;
    list (int x) { ... }
    virtual int last() { ... }
};
```



Ein Objekt der Klasse **list** mit Attributen *info* und *next* und einem Verweis auf die Tabelle der überschreibbaren Methoden der Klasse **list**.

Eine Unterklasse der Klasse **list**

```
class mylist : list {
    int moreInfo;
    virtual int length() {
        if next = null then return 1;
        else return 1+next→length();
        }
};
```



Ein Objekt der Unterklasse **mylist** erbt alle Merkmale der Oberklasse.
In der Unterklasse neu definierte Attribute oder überschreibbare Methoden liegen *über* den Merkmalen der Oberklasse.

Methoden

Eine Methode fassen wir als Funktion mit einem zusätzlichen impliziten ersten (Referenz-)Parameter auf:

- einem Verweis auf das aktuelle Objekt.
  Dieser Verweis wird innerhalb des Methodenrumpfes mit dem Schlüsselwort **this** bezeichnet.

Als erster Parameter steht er im Keller an der Adresse FP-3.

Wir übersetzen daher:

$\text{code}^{\rho}_{W}(\textbf{this}) = \textbf{loadr}\ -3$

Eine Adressumgebung für jede Klasse

Jede Klasse K erhält eine eigene Adressumgebung ρ_K.

Die Adressumgebung ρ_K bildet jeden in der Klasse K sichtbaren Namen x auf eine –um ein Etikett erweiterte– Relativadresse a ab.

Wir unterscheiden durch Etiketten folgende Arten von Namen:

globale Variable	(G, a)
lokale Variable, formale Parameter	(L, a)
Attribut	(A, a)
nicht-überschreibbare Methode	(N, a)
überschreibbare Methode	(V, a)

Die Adressumgebung ρ_K enthält für eine

- nicht-überschreibbare Methode x die Anfangsadresse des Codes für x.
- überschreibbare Methode x die Relativadresse in der Tabelle üM_K, wo die Anfangsadresse von x abgelegt ist.

Adressierung von Attributen

Sonst wird Code zur Berechnung der Adresse wie folgt erzeugt:

$$\text{code}_A^\rho\ x = \begin{cases} \textbf{loadc}\ a & \text{, falls } \rho(x) = (G, a) \\ \textbf{loadr}\ a & \text{, falls } \rho(x) = (L, a) \\ \textbf{loadr}\ -3,\ \textbf{loadc}\ a,\ \textbf{add} & \text{, falls } \rho(x) = (A, a) \end{cases}$$

Attribute des aktuellen Objekts werden über den Verweis **this** adressiert.

Zur Adressierung von Attributen können wir Kurzbefehle einführen:

loadmc $q \equiv$ **loadr** -3, **loadc** q, **add**
loadm $q\ m \equiv$ **loadmc** q, **load** m
storem $q\ m \equiv$ **loadmc** q, **store** m

wobei wir ein Argument m=1 weglassen.

Methodenaufruf

Einen Methodenaufruf $e_1.f(e_2, \ldots, e_n)$ behandeln wir als Funktionsaufruf, dem als zusätzlicher erster Parameter der Wert des Ausdrucks e_1 übergeben wird, der ein Verweis auf ein Objekt o sein muss.

Eine überschreibbare Methode f wird indirekt über das Objekt o und deren Tabelle der überschreibbaren Methoden aufgerufen.

Ein Methodenaufruf $f(e_2, \ldots, e_n)$ ohne Angabe eines Objekts steht für: **this** $\rightarrow f(e_2, \ldots, e_n)$ bzw. $(*\textbf{this}).f(e_2, \ldots, e_n)$

Zur Vereinfachung der Übersetzungsschemata nehmen wir an, dass

- keine optionalen Parameter vorkommen und
- der Platz für die aktuellen Parameter (incl. des übergebenen Objektverweises) stets für den Rückgabewert ausreicht.

$$\text{code}^{\rho}_{W}\ f(e_1, \ldots, e_n) =$$
$$\cancel{\textbf{alloc}\ q,}\ \text{code}^{\rho}_{W}\ e_n, \ldots, \text{code}^{\rho}_{W}\ e_1,\ \textbf{mark}, \text{code}^{\rho}_{W}\ f, \textbf{call}\ \cancel{, \textbf{slide}\ d\ |t|}$$

Übersetzung eines Methodenaufrufs

Sei K die (statisch bekannte) Klasse des Ausdrucks e_1.

Für eine **nicht-überschreibbare Methode** f mit $\rho_K(f) = (N, \_f)$ gilt:

$$\text{code}^{\rho}_{W}\ e_1.f(e_2, \ldots, e_n) = \text{code}^{\rho}_{W}\ e_n, \ \ldots, \ \text{code}^{\rho}_{W}\ e_2, \ \text{code}^{\rho}_{A}\ e_1, \ \textbf{mark}, \ \textbf{loadc}\ \_f, \ \textbf{call}$$

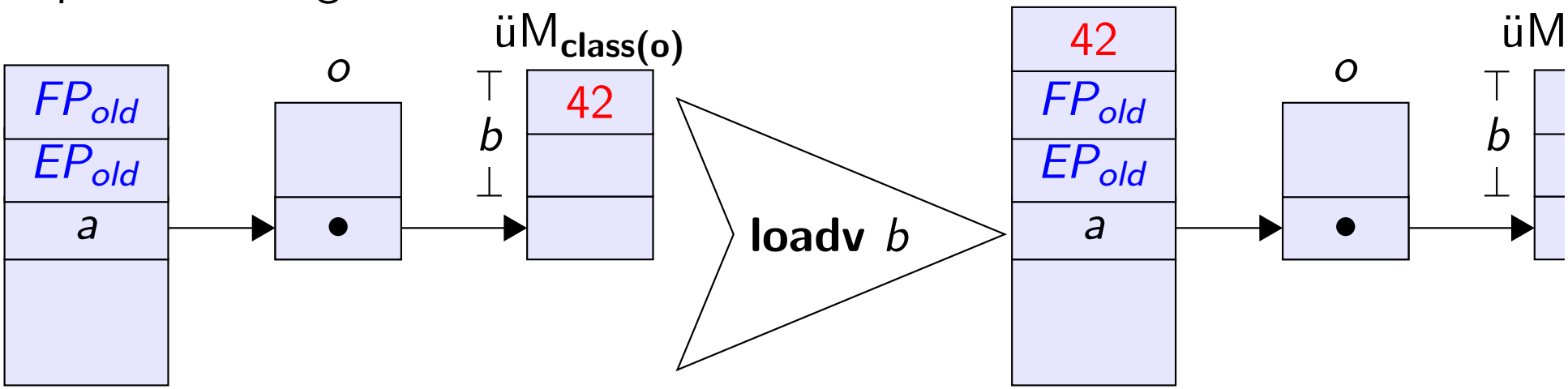
Das Objekt, zu dem sich e_1 auswertet, wird **by reference** übergeben.
Daher wird für den Parameter e_1 Code zur Berechnung der Adresse erzeugt und nicht zur Berechnung des Wertes wie für die übrigen Parameter.

Für eine **überschreibbare Methode** f kann die anzuspringende Codeadresse erst zur Laufzeit ermittelt werden.
Aber bei allen Unterklassen U_i von K steht die Adresse für f an der gleichen Stelle in deren Tabelle üM$_{U_i}$!
Mit $\rho_K(f) = (V, b)$ erhalten wir daher:

$$\text{code}^{\rho}_{W}\ e_1.f(e_2, \ldots, e_n) = \text{code}^{\rho}_{W}\ e_n, \ \ldots, \ \text{code}^{\rho}_{W}\ e_2, \ \text{code}^{\rho}_{A}\ e_1, \ \textbf{mark}, \ \textbf{loadv}\ b, \ \textbf{call}$$

Der Befehl **loadv**

Der Befehl **loadv** b berechnet aus der Relativadresse b von f innerhalb des Objekts o, zu dem sich e_1 auswertet, die Anfangsadresse der gewünschten Implementierung von f.



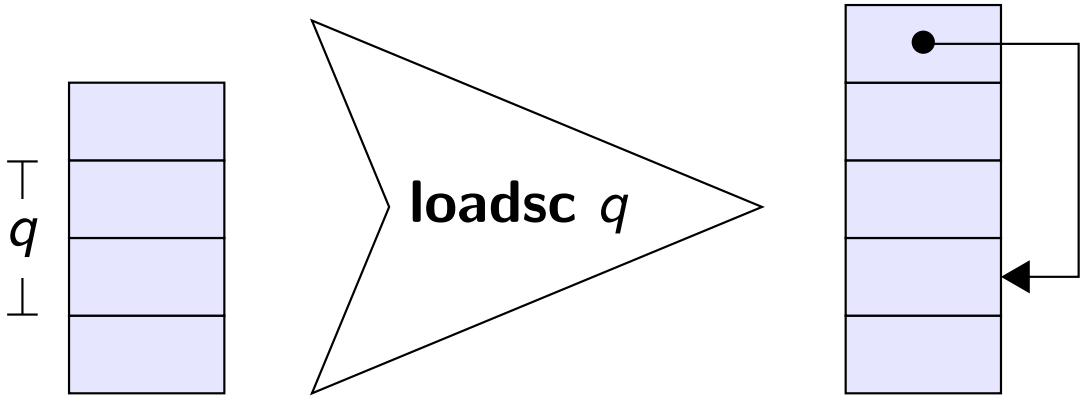
$S[SP+1] \leftarrow S[S[S[SP-2]]+b]; \quad SP++;$

Der Befehl **loadv** b.

Die Anfangsadresse a des Objekts o wird als erster Parameter übergeben; sie steht im Keller unter den organisatorischen Zellen an Adresse SP−2.
Vom Objekt o gelangt man zur Tabelle üM seiner virtuellen Methoden.
In dieser Tabelle steht die Anfangsadresse der Methode f an Adresse b.

Implementierung von **loadv**

Der Befehl **loadv** b greift auf das Objekt o relativ zum SP zu.
Der Befehl **loadsc** q lädt die *Adresse* $-q$ relativ zum Kellerzeiger SP.



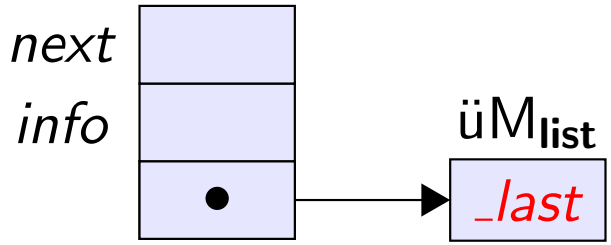
$S[SP+1] \leftarrow SP-q; \quad SP++;$

Der Befehl **loadsc** q.

Der Befehl **loads** q lädt dann den *Inhalt* der Speicherzelle $-q$ relativ zum Kellerzeiger SP:
loads $q \equiv$ **loadsc** q, **load**

Der Befehl **loadv** b wird dann realisiert durch die Folge:
loadv $b \equiv$ **loads** 2, **load**, **loadc** b, **add**, **load**

Methodenaufruf



Beispiel: Die Adressumgebung für die Klasse **list** ist gegeben durch:
$\rho_{list} = \{info \mapsto (A,1),\ next \mapsto (A,2),\ list \mapsto (N, \_list),\ last \mapsto (V,0)\}$

Der rekursive Aufruf $next \rightarrow last() \equiv (*next).last()$ im Rumpf der überschreibbaren Methode *last* wird damit übersetzt in die Folge:
loadm 2, **mark**, **loadv** 0, **call**

Wir erzeugen Code, um die Adresse des Ausdrucks $(*next)$, also den Wert von *next* zu ermitteln. Dieser besteht aus dem Befehl **loadm** 2.

Hier bietet sich ein neuer Spezialbefehl an: **callv** $b \equiv$ **mark**, **loadv** b, **call**

Definition von Methoden

Die Definition einer Methode f einer Klasse K hat die Form:
$t\ f(t_2\ x_2, \ldots, t_n\ x_n)\ \{\ ss\ \}$

Außer dem zusätzlichen ersten Parameter (Verweis auf das aktuelle Objekt) und der Möglichkeit auf diesen Verweis mit dem Schlüsselwort **this** zuzugreifen, ändert sich nichts am Übersetzungsschema für Methoden in C++ gegenüber Funktionen in C.

Definition von Methoden

```
virtual int last() {
    if next = null then return info;
    else return next→last();
    }
```

Beispiel: Die Implementierung der Methode *last* der Klasse **list** liefert:
_*last*: **alloc** 0, **enter** 6, **loadm** 2, **loadc** 0, **eq**, **jumpz** A, **loadm** 1, **storer** −3, **return** 3, A: **loadm** 2, **callv** 0, **storer** −3, **return** 3

Im Vergleich zur Übersetzung von C haben wir nur spezialisierte Befehle benutzt für

- den Zugriff auf die Attribute des aktuellen Objekts: **loadm**, **storem**
- die Anfangsadresse der überschreibbaren Methode: in **callv**

Konstruktoren initialisieren neue Objekte

Konstruktoren initialisieren neue Objekte.

Ein Konstruktor trägt denselben Namen wie die Klasse.

Wir unterscheiden zwei Arten der **Objekterzeugung**:

- indirekt auf der Halde: **new** $K(e_2, \ldots, e_n)$
  Der Wert ist ein *Verweis* auf das neu angelegte Objekt.
- direkt auf dem Keller: $K(e_2, \ldots, e_n)$, z.B. rechtsseitig in Zuweisung
  Als Wert erhalten wir das (initialisierte) Objekt selbst.

In jedem Fall muss zuerst Platz für das neue Objekt reserviert werden.
Den Konstruktoraufruf behandeln wir wie einen Funktionsaufruf in C.
Der Verweis auf das neue Objekt muss dabei als erster (impliziter) Parameter an den Konstruktor übergeben und nach Beendigung des Aufrufs auch oben auf dem Keller hinterlassen werden.

Objekterzeugung auf der Halde bzw. dem Keller

Übersetzungsschema für die indirekte Objekterzeugung:

$$\text{code}^{\rho}_{W}\ (\textbf{new}\ K(e_2,\ldots,e_n)) =$$
$$\textbf{loadc}\ |K|,\ \textbf{new},\ \text{code}^{\rho}_{W}\ e_n,\ \ldots,\ \text{code}^{\rho}_{W}\ e_2,\ \textbf{loads}\ |par|,\ \textbf{calld}\ \_K$$

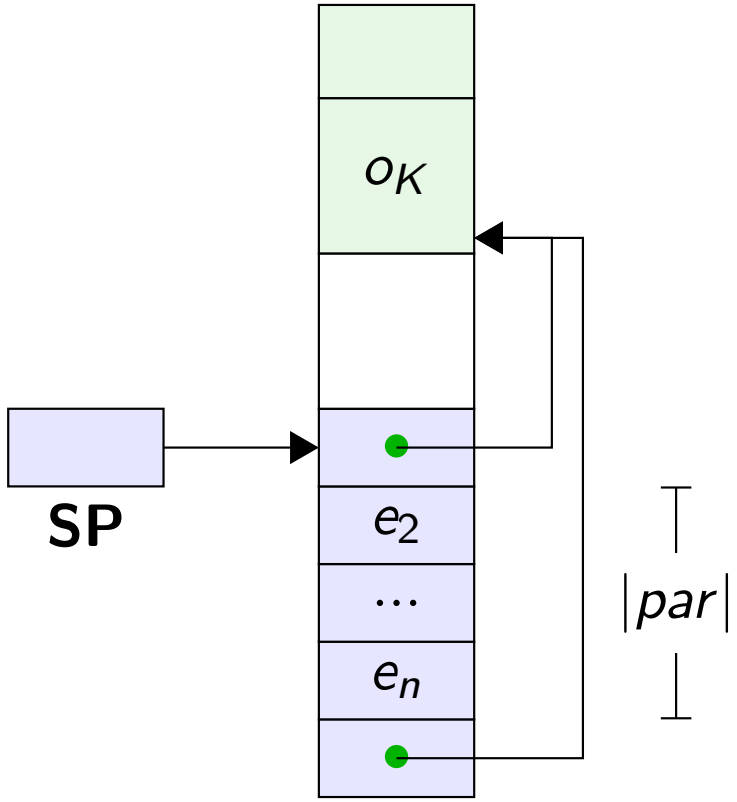
Übersetzungsschema für die direkte Objekterzeugung:

$$\text{code}^{\rho}_{W}\ (K(e_2,\ldots,e_n)) =$$
$$\textbf{alloc}\ |K|,\ \text{code}^{\rho}_{W}\ e_n,\ \ldots,\ \text{code}^{\rho}_{W}\ e_2,\ \textbf{loadsc}\ q,\ \textbf{calld}\ \_K$$

wobei $|K|$ der Platzbedarf für eine Instanz von K,
$|par|$ der Platzbedarf für die aktuellen Parameter $e_2,\ldots,e_n$ und
$\_K$ die Anfangsadresse des Codes für den aufzurufenden Konstruktor der Klasse K ist.

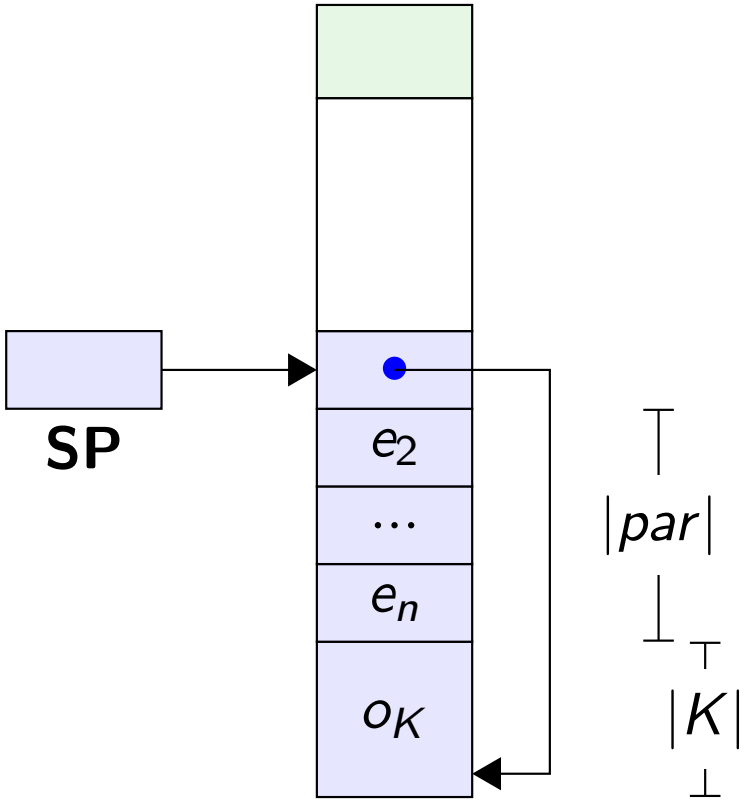
Für direkt erzeugte Objekte lässt sich die Anfangsadresse des neuen Objekts am einfachsten relativ zum Kellerzeiger ermitteln.
Der Abstand zum Kellerzeiger betragt: $q = |par|+|K|-1$

Objekterzeugung auf der Halde



Objekterzeugung auf der Halde
(nach Ausführung von **loads** $|par|$).

Objekterzeugung auf dem Keller



Objekterzeugung im Keller
(nach Ausführung von **loadsc** q).

Die Verwendung von Konstruktoren

Beim Anlegen eines neuen Objekts wurde nur der nötige Platz reserviert.

Den Verweis auf die Tabelle der überschreibbaren Methoden soll dann der Konstruktor anlegen.