Sprachkonzepte

Teil 4: Namen

Bindungen, Scopes, Lebensdauern

Namen

Namen sind ein sprachliches Abstraktionsmittel.

Gibt man einem Gegenstand einen Namen, kann man über den Namen den Gegenstand jederzeit in einfacher Weise referenzieren, unabhängig davon, wie komplex der Gegenstand ist.

in einer Programmiersprache wird von Registern, Speicheradressen usw. abstrahiert
relevante Gegenstände sind dort Variablen, Typen, Operationen usw.

Wichtige Fragen im Zusammenhang mit Namen:

- Bindungszeitpunkte:
  Zu welchen Zeitpunkten können Namen an Gegenstände gebunden werden?
- Scopes:
  In welchen Programmbereichen gelten Bindungen *(bindings)* zwischen Namen und Gegenständen?

  Wie hängen die Geltungsdauern von Bindungen und die Lebensdauern von Gegenständen zusammen?

Bindungszeitpunkte (1)

Zeitpunkte bei der Programmierung, zu denen Namen an Gegenstände gebunden werden können *(aufsteigend von den frühesten zu den spätesten Zeitpunkten)*:

- beim Entwerfen und Implementieren der Programmiersprache

  das Vokabular der Sprache an Programmierkonzepte binden
- beim Schreiben eines Programms

  Namen an benutzerdefinierte Typen, Anweisungsfolgen usw. binden
- beim Übersetzen *(compile time),* Binden *(link time),* Laden *(load time)* und spätestens beim Ausführen *(run time)* eines Programms

  z.B. Variablen und Funktionen an Speicheradressen binden

Die Wahl der Bindungszeitpunkte hat einen großen Einfluss auf einerseits die Effizienz und andererseits die Flexibilität von Programmen.

frühere Bindung fördert die Effizienz, spätere Bindung die Flexibilität

Bindungszeitpunkte (2)

Oft wird nur grob zwischen statischer und dynamischer Bindung unterschieden:

- statische Bindung *(frühe Bindung)*:
  alle Bindungen, die vor der Laufzeit (run time) passieren
- dynamische Bindung *(späte Bindung)*:
  Bindung zur Laufzeit

Geltungsbereiche von Bindungen (Scopes)

Der Textbereich eines Programms, in dem die Bindung zwischen einem Namen und einem Objekt aktiv ist, wird als der **Scope** der Bindung bezeichnet.

umgekehrt werden auch Textbereiche selbst als Scopes bezeichnet, wenn sie einen Geltungsbereich von Bindungen definieren

Objekt wieder im allgemeinen Sinn: etwas, das Speicher belegt

- statische Scope-Ermittlung *(static scoping, lexical scoping)*

  zur Compile-Zeit wird aus der Schachtelung der Blöcke im Programmtext ein Baum von Scopes bestimmt und darin die geltende Bindung von innen nach außen gesucht, d.h. ausgehend vom aktuellen Scope in Richtung Wurzel

  die übliche Lösung in modernen Programmiersprachen

- dynamische Scope-Ermittlung *(dynamic scoping)*

  die geltende Bindung wird entlang der Stack-Frames der Aufrufsequenz gesucht

  Programme sind dadurch schwieriger zu verstehen und Zugriffe auf nicht-lokale Variablen brauchen mehr Laufzeit

Scopes: Beispiel (1)

- Programm in einer fiktiven Programmiersprache:

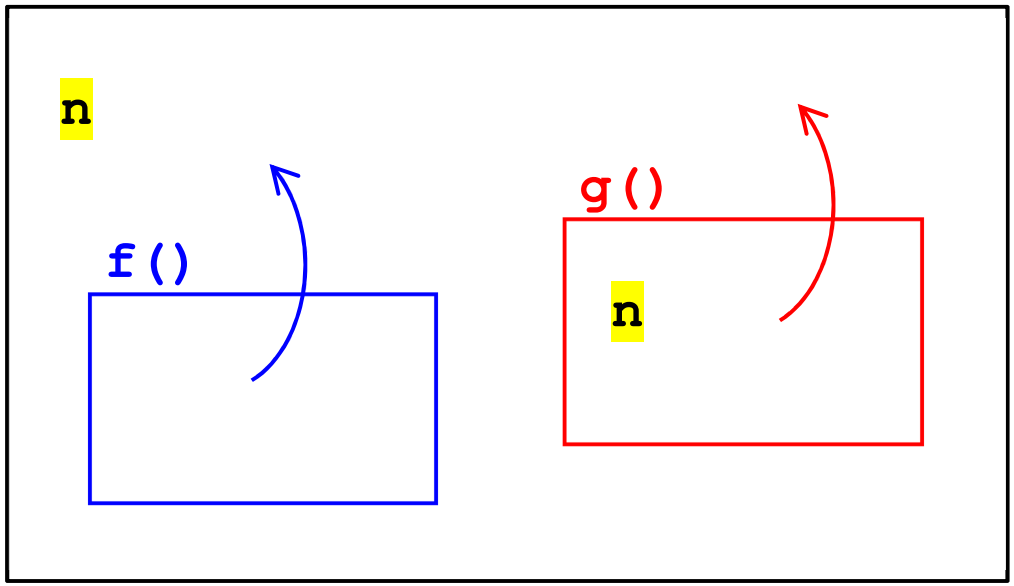
```
n : integer                    -- globale Variable
procedure f()
    n := 1
procedure g()
    n : integer                -- lokale Variable
    f()
n := 2
f()
print n
n := 3
g()
print n
```

- Ausgaben des Programms:
  - `11` *bei statischer Scope-Ermittlung*
  - `13` *bei dynamischer Scope-Ermittlung*

Scopes: Beispiel (2)

- statische Scope-Ermittlung:

  Programme sind in geschachtelte Textblöcke gegliedert.
  Bindungen, die im aktuellen Block fehlen, werden aus dem umgebenden Block bezogen, usw.



*in **f** wird **n** an die globale Variable gebunden*

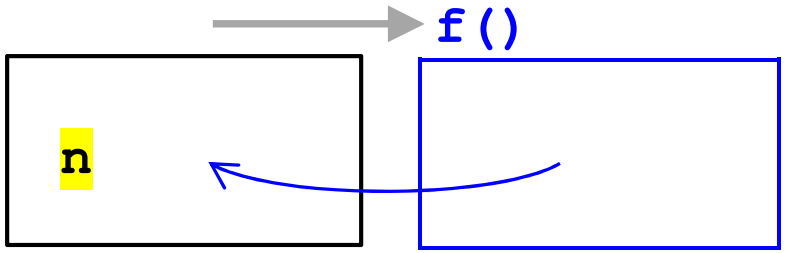
*in **g** wird **n** an die lokale Variable gebunden, die die globale Variable verdeckt (die Bindung von **n** an die globale Variable ist innerhalb von **g** inaktiv)*

Scopes: Beispiel (2)

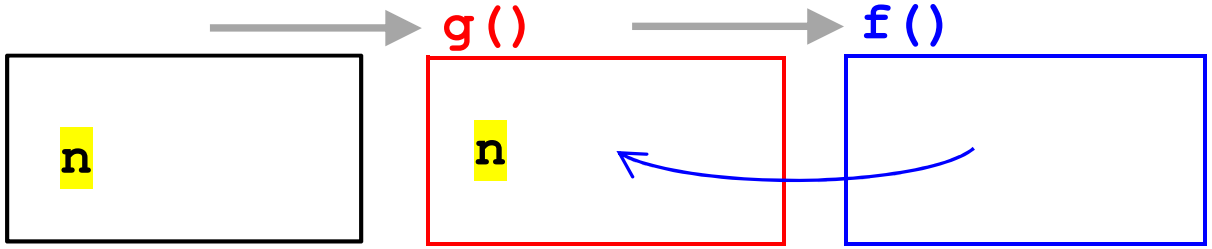
- dynamisch Scope-Ermittlung:

  Bindungen, die im aktuellen Frame fehlen, werden aus dem vorhergehenden Frame (= dem Frame des Aufrufers) bezogen, usw.

bei Aufruf von `f` aus dem Hauptprogramm Bindung von `n` in `f` an die globale Variable



bei Aufruf von `f` aus `g` Bindung von `n` in `f` an die lokale Variable in `g`



Lebensdauer von Objekten (object lifetime)

Mit Objekten sind hier Gegenstände gemeint, die zur Laufzeit Speicher belegen.
das ist ein allgemeinerer Objektbegriff als der der objektorientierten Programmierung

Die Objektlebensdauer hängt von der Art der Speicherverwaltung ab:

- statische Allokierung

  während der gesamten Programmlaufzeit feste absolute Adressen
  verwendet für globale Variablen, Programmcode, unterstützende Daten für Debugging und dynamische Typprüfung usw.

- Stack-basierte Allokierung

  Allokierung und Deallokierung in Last-in- / First-out-Reihenfolge
  im Zusammenhang mit Funktionsaufrufen
  verwendet für Parameter und lokale Variablen von Funktionen

- Heap-basierte Allokierung

  Allokierungen und Deallokierungen zu beliebigen Zeitpunkten
  verwendet z.B. für Strings, Arrays, dynamische Datenstrukturen wie Listen usw.

Stack-basierte Allokierung (1)

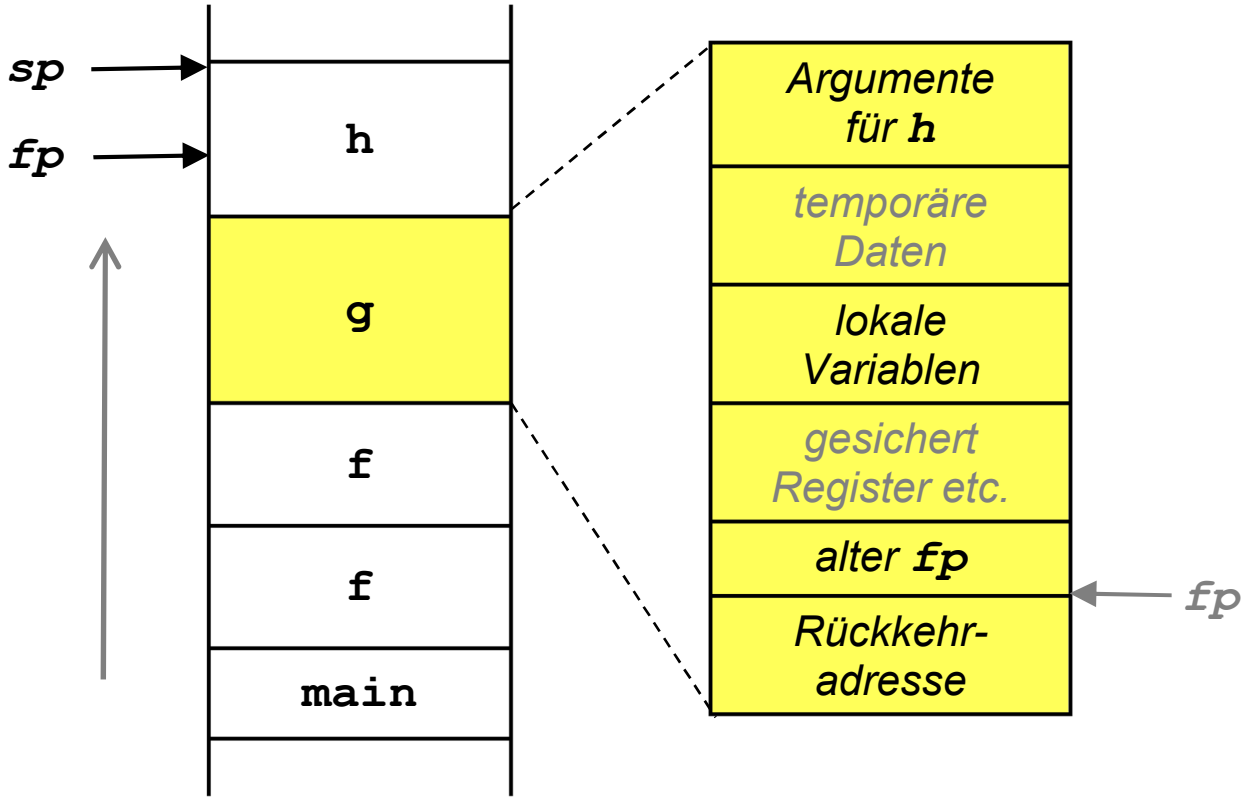
Bei jedem Aufruf einer Funktion wird ein **Frame** auf dem Stack allokiert:

```
void h(…) {
   …
}

void g() {
   h(…);
}

void f() {
   if (…)
      f();
   else
      g();
}

void main() {
   f();
}
```



sp = stack pointer, fp = frame pointer

Stack-basierte Allokierung (2)

Aufrufsequenz beim Aufrufer:

1. Register etc. sichern
2. Aufrufargumente auf den Stack legen
3. Unterprogrammsprung *(legt Rückkehradresse auf den Strack)*
4. Rückgabewert entgegennehmen
5. Register etc. wiederherstellen

Aufgerufene Funktion:

1. Frame allokieren *(sp −= frame size)*
2. Frame-Pointer sichern und dann aktualisieren
3. Register etc. sichern

Prolog

4. Rumpf ausführen

5. Rückgabewert bereitstellen
6. Register etc. wiederherstellen
7. Frame-Pointer wiederherstellen
8. Frame deallokieren *(sp += frame size)*
9. Rücksprung

Epilog

Stack-basierte Allokierung (3)

Nutzen und Vorteile der Stack-basierten Allokierung:

- ermöglicht rekursive Funktionen
  weil pro Frame jeweils eine Instanz der lokalen Variablen und Parametern
- einfach und effizient
  durch Register (fp, sp, ...) und Maschinenbefehle unterstützt
  keine Fragmentierung des Speichers

Heap-basierte Allokierung (1)

Die Flexibilität von Allokierungen und Deallokierungen zu beliebigen Zeitpunkten macht die **Heap-Verwaltung** komplex.

schwieriger Kompromiss zwischen Laufzeit, Platzbedarf und Fehlerrisiko

- beim Allokieren muss ein passendes freies Stück gesucht und gegebenenfalls in ein belegtes Stück und ein freies Reststück zerlegt werden

  First-fit- oder Best-fit-Suche in einer Liste freier Speicherstücke (Aufwand $O(n)$ bei n freien Speicherstücken)

  oder Verwaltung getrennter Pools für vorgegebene Stückelungen mit entsprechender Aufrundung der Speicheranforderungen (z.B. beim schnellen Buddy-Verfahren nur Zweierpotenzen als Stückelung)

- beim Deallokieren sollte versucht werden, benachbarte freie Stücke wieder zusammenzufassen

  dafür gegebenenfalls Nachbarschaftslisten erforderlich (beim Buddy-Verfahren sind Nachbarschaften per Adressrechnung bestimmbar)

Heap-basierte Allokierung (2)

- sehr unterschiedliche Größen und Lebensdauern von Speicherstücken führen zu einer Fragmentierung des Heaps:

  externe Fragmentierung durch die verstreute Lage allokierter Speicherstücke

  interne Fragmentierung durch Aufrundung von Speicheranforderungen auf vorgegebene Stückelungen

  dadurch können große Speicheranforderungen eventuell nicht befriedigt werden, obwohl in Summe eigentlich genug Speicher frei wäre

- vergessenes Deallokieren führt zu Speicherlecks, verfrühtes Deallokieren zu dangling Pointers

  ein richtiger Zeitpunkt kann je nach Programmiersprache eventuell nur automatisch mittels Garbage-Collection bestimmt werden

Garbage-Collection: Idee

Ein **Garbage-Collector** sorgt auf dem Heap für die Wiederverwendung allokierter Speicherstücke, sobald diese nicht mehr verwendet werden.

Das automatisierte Deallokieren verhindert Speicherlecks und dangling Pointers.

Das Zusammenschieben verbleibender belegter Speicherstücke wirkt zusätzlich der Fragmentierung des Heaps entgegen.

Technische Voraussetzungen:

- der Garbage-Collector muss zwischen belegten und freien Speicherstücken im Heap unterscheiden können
- der Garbage-Collector muss erkennen können, welche belegten Speicherstücke im Programm aktuell noch verwendet werden

  dazu muss er Referenz- / Zeiger-Variablen im globalen Datensegment, in den aktiven Stack-Frames und in belegten Heap-Stücken als solche erkennen können (bei C und C++ ist das z.B. nicht exakt machbar)

Garbage-Collection: Mark-and-Sweep

Das **Mark-and-Sweep-Verfahren** arbeitet in zwei Phasen:

- Markierungsphase *(mark)*:

  Alle vom Root-Set (= globale plus lokale Variablen) aus direkt oder indirekt erreichbaren Speicherstücke markieren.

- Bereinigungsphase *(sweep)*:

  Alle nicht markierten Speicherstücke deallokieren (dabei gegebenenfalls Zusammenfassung mit benachbarten bereits deallokierten Speicherstücken).

  Bei allen markierten Speicherstücken die Markierung entfernen.

Bewertung:

- nur begrenzt wirksam gegen externe Fragmentierung,
  weil erreichbare Speicherstücke niemals verschoben werden
- Sweep-Phase braucht viel Laufzeit, weil sie alle allokierten Speicherstücke bearbeitet, nicht nur die erreichbaren

Garbage-Collection: Stop-and-Copy

Das **Stop-and-Copy-Verfahren** teilt den Heap in zwei gleich große Bereiche:

- zwischen zwei Bereinigungen immer nur aus einem der Bereiche allokieren
- bei einer Bereinigung alle vom Root-Set aus erreichbaren Speicherstücke in den zweiten Bereich umkopieren

  dabei jeweils die neue Adresse in der referenzierenden Variablen speichern und zusätzlich im ursprünglichen Speicherstück hinterlegen
- in allen umkopierten Speicherstücken die enthaltenen Zeiger auf die hinterlegten neuen Adressen umsetzen
- zuletzt die Rollen der beiden Bereiche tauschen

Bewertung:

- sehr wirksam gegen externe Fragmentierung, weil Speicherstücke beim Kopieren ohne Lücken abgelegt werden
- relativ laufzeiteffizient, weil nur erreichbare Speicherstücke betrachtet werden (typischerweise die deutliche Minderheit der allokierten Speicherstücke)

Garbage-Collection: Mehrgenerationen-Heap

Ein **Mehrgenerationen-Heap** verwaltet Speicherstücke, die seit der letzten Bereinigung neu allokiert wurden *(junge Generation)* und solche, die schon eine oder mehrere Bereinigungen überlebt haben *(alte Generation)*, in getrennten Speicherbereichen:

- junge Generation mit Stop-and-Copy bereinigen

  Speicherstücke, die eine bestimmte Anzahl Bereinigungen überlebt haben, in die alte Generation verlagern

- gelegentlich alte Generation mit Mark-and-Sweep bereinigen

Bewertung:

- vermeidet gegenüber reinem Stop-and-Copy das permanente Hin- und Her-Kopieren langlebiger Speicherstücke
- effizienter Umgang mit generationsübergreifenden Referenzen etwas aufwendig

  die Speicherstücke der alten Generation erweitern im Prinzip das Root-Set für die Erreichbarkeitsprüfung in der jungen Generation

Bindung von lokalen Variablen (1)

Bei lokalen Variablen und Parametern einer Funktion stack-basierte Allokierung und Bindung der Namen zur Compile-Zeit:

- die Namen werden an Adressen relativ zum Framepointer gebunden

  der Compiler bestimmt für jede Funktion das Layout ihres Frames,
  daraus ergibt sich für jeden Parameter und jede Variable ein bestimmter Offset,
  also bei einem Stack, der in Richtung der kleinen Adressen wächst:

  Parameter p: $fp + offset_p$

  lokale Variable v: $fp - offset_v$

- bei Programmiersprachen mit geschachtelten Funktionsdefinitionen wird zusätzlich eine Verkettung der Frames auf dem Stack genutzt *(static links)*

  über den static link im Frame werden die Parameter und lokalen Variablen
  der umgebenden Funktion adressiert (usw. bei tieferer Verschachtelung):

  Parameter q: $*(fp + offset_{static\ link}) + offset_q$

  lokale Variable w: $*(fp + offset_{static\ link}) - offset_w$

Bindung von lokalen Variablen (2)

Der Scope der Bindung einer lokalen Variablen ist maximal der Funktionsrumpf, weil der die Lebensdauer des Stack-Frames beschränkt.

Je nach Programmiersprachen kann der Scope aber auch kleiner sein:

- der Scope beginnt eventuell erst ab der Zeile der Variablendefinition

  z.B. bei C (ab C99) und Java, aber nicht bei Python

- der Scope kann auf einen umschließenden Anweisungsblock beschränkt sein

  z.B. bei C und Java definieren geschweifte Klammern einen Scope

- der Scope kann Unterbrechungen haben

  bei geschachtelten Scopes kann eine Bindung in einem eingebetteten Scope eine Bindung des umgebenden Scopes mit gleichem Namen verdecken

Lokale Variablen mit nicht überlappenden Scopes kann der Compiler an dieselbe Adresse im Frame binden, um Speicherplatz zu sparen.

Bindung von Funktionen

Bei Funktionen (und globalen Variablen) statische Allokierung und Bindung der Namen zur Link-Zeit.

je nach Implementierung Bindung an eine absolute oder relative Adresse

Der Scope der Bindung ist die gesamte Programmlaufzeit, kann aber je nach Programmiersprache Unterbrechungen haben:

- globale Namen können in einigen Programmteilen verdeckt sein
- die Bindung modul-privater Funktionen ist außerhalb des Moduls inaktiv

  z.B. static markierte Funktionen in C, Funktionen in anonymem namespace in C++
- Bindungen aus anderen Übersetzungseinheiten müssen explizit aktiviert werden

  z.B. Prototypen in C

Bindung an Heap-Objekte

Adressen von Objekten auf dem Heap werden immer erst zur Laufzeit bekannt. Deshalb sind Zeiger bzw. Referenzen (je nach Programmiersprache) erforderlich, die die Heapadresse des Objekts aufnehmen.

Die Namen werden zur Compile-Zeit an die Zeiger bzw. Referenzen gebunden. Diese Indirektion kann die bereits genannten Probleme verursachen:

- dangling pointer / reference

  Zeiger / Referenz existiert noch, das referenzierte Objekt aber nicht mehr

- memory leak

  Heap-Objekt existiert noch, aber es gibt dazu keine Zeiger / Referenzen mehr

  beide Probleme lassen sich durch die automatisierte Freigabe von Heapspeicher per Garbage-Collector vermeiden, oder wie in C++ per intelligente Zeiger deutlich reduzieren

Aliase und Überladung

Die Bindung zwischen Namen und Objekten ist nicht immer eine 1:1-Beziehung.

- Aliase:

  es können mehrere Namen an dasselbe Objekt gebunden werden (n:1)

  immer der Fall bei indirekter Bindung über Zeiger / Referenzen

- Überladung *(overloading)*:

  ein Name wird kontextabhängig an verschiedene Objekte gebunden (1:m)

  der Aufrufname einer Funktion wird abhängig von Anzahl und Typ der Aufrufargumente an verschiedene Implementierungen gebunden

Bindungsumgebungen *(referencing environments)*

Die **Bindungsumgebung** *(auch: der Kontext)* einer Anweisung ist die Menge aller bei der Ausführung der Anweisung aktiven Bindungen.

- bei der funktionalen Programmierung wird die Kombination aus einer Funktion und einer Bindungsumgebung als Closure bezeichnet

  Closures treten auf, wenn Funktionen als Aufrufargumente an Funktionen übergeben oder als Rückgabewerte von Funktionsaufrufen geliefert werden

  die Funktionen sind dabei oft Lambdas, d.h. sie sind anonyme Funktionen

- die von der Bindungsumgebung einer Closure zugelieferten Variablen werden als freie Variablen der zugehörigen Funktion bezeichnet

  im Gegensatz zu den Aufrufparametern und den lokalen Variablen

Closures: Beispiele

Beispiele für Closures in Python:

```
def f(c):
    print(c(1))

x = 2
f(lambda y: x + y)   # Closure als Aufrufargument, x ist freie Variable


def g(x):
    def h(y):
        return x + y   # x ist freie Variable von h
    return h   # Closure als Rückgabewert

c = g(3)
print(c(4))
```

Closure wird mit Wert 2 für x und Wert 1 für y ausgeführt

Closure wird mit Wert 3 für x und Wert 4 für y ausgeführt (Achtung: der Wert 3 des Parameters x darf nicht im Stackframe von g gespeichert sein, weil der Frame hier bereits nicht mehr exisitiert!)