universitätbonn

Dynamische Speicherverwaltung

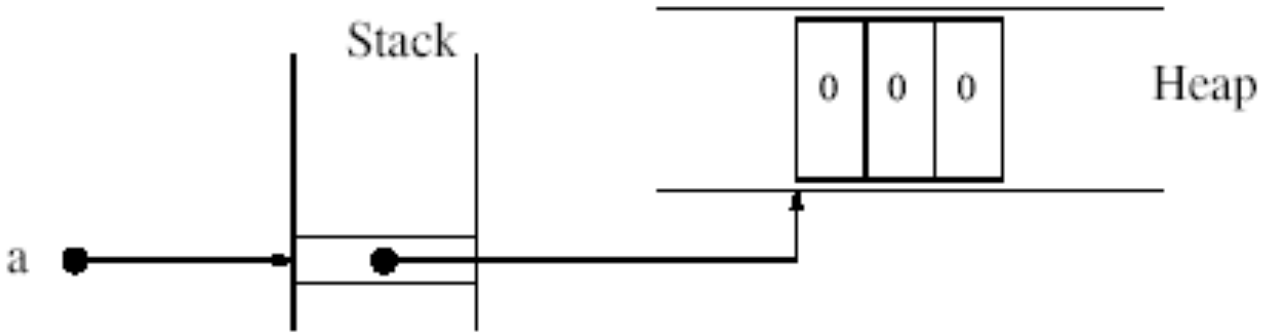
Eine sehr rudimentäre Skizze

Motivation für dynamische Speicherverwaltung

- Oftmals Größe von Arrays nicht zur Compilierzeit bekannt
  - Bisheriges Schema nur für konstante Größe zulässig
- Neues Konzept
  - Funktion für dynamische Allokation
  - malloc
    - Memory allokations
    - Funktionsprototyp
      - **void * malloc(unsigned int length);**
        - void * bezeichnet Referenz auf beliebigen Datentyp
  - Beispiel für array
    - **double * array = malloc(n * sizeof(double));**

Halde und Stapel

- Die Speicherbereiche Stapel und Halde nach der Anweisung **int * a = malloc(3 * sizeof(int));**



Speicherallokation auf dem Heap (Haldenspeicher)

- Durch malloc wird Speicher nicht auf dem Stapelspeicher, sondern einem anderen Bereich (dem Haldenspeicher (Heap)) allokiert
- Und muss dort explizit freigegeben werden, wenn er nicht mehr gebraucht wird
  - In C und C++
  - Keine automatische Freigabe
    - In Java erfolgt eine automatische “garbage collection”
  - Befehl zur Freigabe:
    - **void free(void array);**

Verwaltung von Objekten im Speicher

- Tote Objekte auf der Halde bezeichnet man als **Abfall** (garbage)
- Um den Haldenspeicher wieder verwenden zu können, kennt das Java-Laufzeitsystem eine **automatische Speicherbereinigung** oder „Abfallsammlung“ (garbage collection)
  - Die es z.B. anstößt, wenn der Haldenspeicher zur Neige geht

Explizite Verwaltung von Objekten im Speicher

- In C und C++ hingegen muss (bzw. darf) die Speicherrückgabe explizit programmiert werden
- Der Programmierer muss für jede von ihm entworfene Klasse eine spezielle Funktion schreiben, einen sogenannten **Destruktor** (destructor), in der die Rückgabe des Speichers von einem nicht mehr benötigten Objekt beschrieben wird
  - Dieser ruft i.A. einen Operator delete auf
- Dies ist in vielen Fällen effizienter als eine automatische Speicherbereinigung
  - Ist aber für den Programmierer sehr viel umständlicher und kann leicht zu Programmierfehlern führen

Explizte Verwaltung von Objekten im Speicher

- Insbesondere können dabei folgende Programmierfehler vorkommen
  - Speicher für ein Objekt wird irrtümlicherweise nicht zurückgegeben, obwohl es tot ist
  - Speicher für ein Objekt wird fälschlicherweise zurückgegeben wird, obwohl es noch nicht tot ist

Immer mehr Speicher wird verbraucht „Speicherleck“ (memory leak)

Solche Fehler sind extrem boshaft, da sie erst dann bemerkt werden, wenn der Speicherplatz erneut zugeteilt und überschrieben wurde

Können in C++ vorkommen, nicht in Java

universitätbonn

Sichtbarkeit von lokalen und globalen Variablen in Blöcken und Unterprogrammen

Blöcke

- Ein Block (block) ist eine Anweisung, die aus einer Folge von Anweisungen besteht, die durch Begrenzer zusammengefasst werden
  - In C, C++ und Java sind die Begrenzer geschweifte Klammern { ... }
  - In der ALGOL-Familie (incl. Pascal) sind die Begrenzer ein Paar begin ... end
- Der Block { } stellt die leere Anweisung dar
- Geschachtelte Blöcke (nested blocks) sind möglich, also z. B. {...{...}...}
  - Wir sprechen von äußeren und inneren Blöcken (outer / inner blocks) und von der Schachtelungstiefe eines Blocks (nesting depth)

Blöcke mit lokalen Deklarationen

- In einem Block enthaltene Deklarationen werden als lokale Deklarationen dieses Blocks bezeichnet
  - Jeder Block kann lokale Deklarationen (von Variablen oder Typen) enthalten
  - In einem Block deklarierte Variablen sind lokale Variablen des Blocks.
- In einem Block deklarierte Namen sind nur in diesem Block und seinen geschachtelten Unterblöcken sichtbar
  - Dies dient vor allem der Kapselung von Information
    - ð Deklaration und Gebrauch von Namen sollen nahe beisammen liegen, und die Deklaration eines Namens soll nicht den ganzen globalen Namensraum belasten
  - Somit können verschiedene Blöcke jeweils ihre eigene Variable i oder x haben, ohne sich gegenseitig zu stören

Blöcke

- Grundsätzlich können wir auch den Rumpf einer Klasse als Block ansehen
- In diesem Block sind sowohl die Felder (Daten) der Klasse deklariert, als auch ihre Methoden (Funktionen)
- Auch ein Funktionsrumpf ist ein Block:
  - Dort sind lokale Variablen deklariert
- Jede Programmiersprache trifft Einschränkungen, welche Sprachobjekte in welcher Art von Blöcken deklariert werden dürfen
  - In Pascal kann z. B. ein Funktionsrumpf wieder eine Funktionsdeklaration enthalten
  - In C++ und Java geht dies nur bei Klassenrümpfen
  - In C können Funktionen nur im globalen Block auf der obersten Programmebene (entsprechend unserer Klasse Program) definiert werden

Blöcke

- Nun stellen sich zwei Hauptfragen
  - Nach dem **Gültigkeitsbereich** für einen Namen
  - Nach der **Lebensdauer** des bezeichneten Objekts

Blöcke

- Ein Name für ein Sprachobjekt ist an einem Ort **gültig** (valid), wenn er dort tatsächlich dieses Objekt bezeichnet
  - und kein anderes
- In der Folge konzentrieren wir uns auf den Fall von Variablen
- Eine Variable, die in einem Block deklariert wurde, ist dort eine **lokale Variable**

Blöcke

- In C/C++ und Pascal ist der **Gültigkeitsbereich** (scope) eines zu Beginn eines Blockes deklarierten Namens der ganze Block
  - Mit der Ausnahme etwaiger inneren Blöcke, die eine erneute Deklaration des Namens (für ein anderes Objekt) enthalten
  - In den verschiedenen Programmiersprachen kann es unterschiedliche Ausnahmen geben, wenn man die Objekte am Typ unterscheiden kann
- In Java ist eine erneute Deklaration eines Namens in einem inneren Block verboten, da dies manchmal zu schwer auffindbaren Fehlern führt
  - Wenn man den Überblick verliert, welche Deklaration gerade gilt
- Allerdings ist in Java die erneute Deklaration eines Namens erlaubt, der schon ein Feld der umgebenden Klasse bezeichnet

Blöcke

- Folgender Code ist in C/C++ gültig, aber nicht in Java

```
{ int x = 1; // Deklaration 1
int y;
  { int x=2; // Deklaration 2
  // OK in C/C++, Fehler in Java
  y=x; // Stelle 1
  }
y=x; // Stelle 2
}
```

- Es gibt zwei verschiedene Blöcke mit zwei verschiedenen Variablen mit dem Bezeichner x
- In C/C++ gilt folgendes:
  - Das x im äußeren Block hat durchgängig den Wert 1 (z.B. an Stelle2), das im inneren Block hat den Wert 2 (z.B. an Stelle 1)
  - Das x aus Deklaration 1 ist im inneren Block nicht gültig
  - An Stelle 1 bezeichnet x die Variable aus Deklaration 2, an Stelle 2 bezeichnet x die Variable aus Deklaration 1.
- In Java ist der Code nicht korrekt, da die Deklaration 2 das x aus Deklaration 1 verdeckt

Blöcke

- Folgender Code ist sowohl in C/C++ als auch in Java gültig

```
{ int x; // Block 1
  {int y=1; // Block 1.1
    x=y;
    // ...
  }
  {int y=2; // Block 1.2
    // ...
  }
}
```

- In Block 1.1 und Block 1.2 existieren (zu verschiedenen Zeiten) zwei verschiedene Variablen, die beide den Bezeichner y haben

Blöcke

- Bei Variablen ist nicht nur der Name lokal zum Block, in dem sie deklariert sind, sondern auch der Speicherplatz
- Eine Variable lebt genau so lange, wie für sie Speicher reserviert ist
- Die **Lebensdauer** (lifetime) einer in einem Block deklarierten Variablen ist die Zeit vom Eintreten des Kontrollflusses in den Block bis zum Austreten aus dem Block
  - Einschließlich des Verweilens in inneren Blöcken

Blöcke

- Jedem Block entspricht ein Speicherbereich (**Rahmen**, frame) auf dem **Laufzeitstapel** (run-time stack)
  - Dort sind die Werte der lokalen Variablen abgelegt
- Bei jedem Eintreten in den Block wird der zugehörige Rahmen zuoberst auf dem Laufzeitstapel **angelegt** (allocate)
  - Er verbleibt dort solange, bis der Kontrollfluss den Block verlässt
    - ð Worauf der Rahmen wieder entfernt wird
- Jeder lokale Variablenname wird mit einer Adresse in diesem Speicherblock verbunden
  - Damit dies einfach geht ist jede Referenz einer lokalen Variablen bereits als **Versatz** (offset) relativ zu einer zur Übersetzungszeit noch unbestimmten Anfangsadresse angegeben
  - Für jeden neuen Block wird diese **Basisadresse** neu bestimmt und oft in einem **Basisadressregister** gespeichert

Blöcke

- Tritt der Kontrollfluss in einen inneren Block ein, so kommt der Rahmen des inneren Blockes im Stapel auf den Rahmen des äußeren zu liegen
- Bei einem rekursiven Eintreten in denselben Block wird erneut ein Rahmen zuoberst auf den Stapel gelegt und jede (lokale) Variable erfährt eine neue **Inkarnation**
  - indem ihr Name an einen neuen Speicherplatz gebunden wird (vgl. rekursive Prozeduren später).
- Das **Binden** (linking) einer Referenz an einen Namen kommt auch nach der Übersetzung vor, wenn die Startadressen von Bibliotheksfunktionen an die Namen von externen Funktionsaufrufen gebunden werden (sog. external linking)

Blöcke

- Eine Variable, die auf dem Laufzeitstapel angelegt wird, heißt auch **dynamische Variable**
  - da ihre Speicherstelle erst zur Laufzeit bestimmt wird
  - Ihre **absolute Adresse** erhält man nur über eine **Adressrechnung** zur Laufzeit
    - ð Die Adressrechnung steht in C/C++ dem Programmierer auch explizit zur Verfügung, in Java kann er eine solche nicht direkt durchführen
  - In C gibt es auch **statische Variablen**, die zur Übersetzungszeit in einem festen globalen Speicherbereich angelegt werden und die daher keine Adressrechnung brauchen
  - Diese haben eine unbegrenzte Lebensdauer und behalten ihre Werte zwischen Prozeduraufrufen

Blöcke

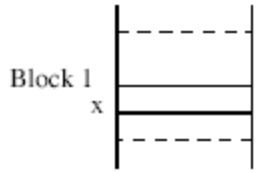
- Der **Stapelspeicher** (stack) pulsiert also im Takt des Abarbeitens von Blöcken
  - Werte, die in einem inneren Block gespeichert sind, gehen nach Ende des Blocks verloren
  - Sie können gerettet werden, indem sie noch im inneren Block an eine Variable eines äußeren Blocks zugewiesen werden
  - **Bemerkung:** Das Prinzip des Stapelspeichers und Prinzipien der Werteübergabe werden wir am Beispiel von Unterprogrammen (Prozeduren, Funktionen) noch ausführlich diskutieren

Blöcke

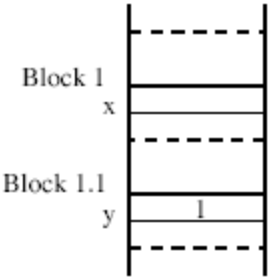
- Bild des Stapelspeichers beim Durchlaufen des Codes aus obigem Beispiel (nochmals unten angegeben)
  - Es hat sich im Bereich der Systemprogrammierung eingebürgert, Stapelspeicher als *nach unten wachsend* zu zeichnen
  - Die Rahmen der inneren Blöcke werden unter den Rahmen des äußeren Blocks gezeichnet

```
{ int x; // Block 1
  {int y=1; // Block 1.1
    x=y;
    // ...
  }
  {int y=2; // Block 1.2
    // ...
  }
}
```

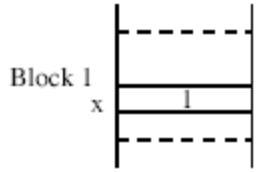
Zu Beginn von Block 1:



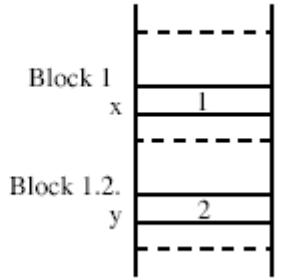
Zu Beginn von Block 1.1:



Nach Ende von Block 1.1. und vor Beginn von Block 1.2:



Zu Beginn von Block 1.2:

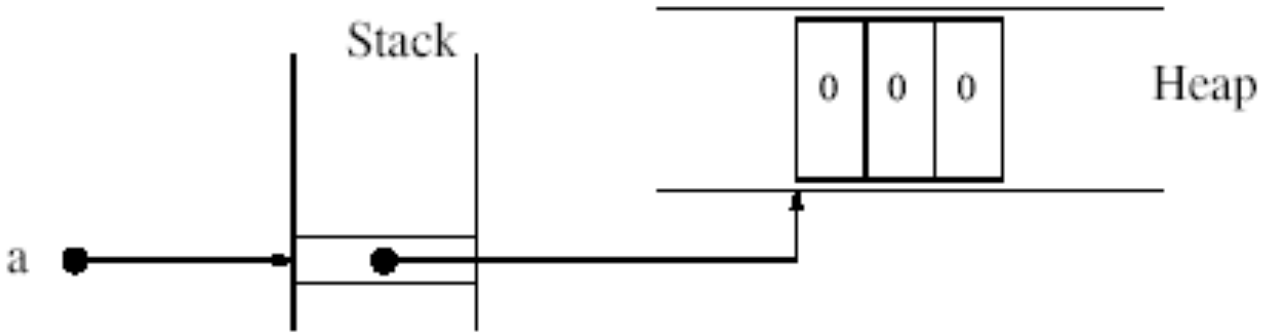


Blöcke

- Der Haldenspeicher (Halde, heap) auf dem alle mit new erzeugten Klassenobjekte gespeichert werden, kennt dagegen keine Rahmen und pulsiert *nicht* im Takt der Abarbeitung von Blöcken
- Das Objekt auf der Halde lebt weiter, auch wenn die entsprechende Referenzvariable vom Stapel verschwindet
  - Gibt es auf das Objekt keine gültige Referenz mehr, so kann es vom garbage collector gelegentlich eingesammelt und sein Speicherbereich recycled werden
- Es ist insbesondere möglich, die Referenz auf das Haldenobjekt an eine Variable in einem äußeren Block zuzuweisen und damit nach draußen (außerhalb des Blocks) weiterzugeben

Blöcke: Halde und Stapel

- Zur Erinnerung: Die Speicherbereiche Stapel und Halde nach der Anweisung **int * a = malloc(3 * sizeof(int));**



Binky Pointer Fun