Imperative Sprachen

Rücksprung zum Binding: Speicher-Bindung

1. bei statischen Variablen
2. bei Stack-dynamischen Variablen
3. bei explizit Heap-dynamischen Variablen
4. bei implizit Heap-dynamischen Variablen

Imperative Sprachen

Speicher-Bindung bei statischen Variablen:

- werden vor Programmausführung an Speicherzelle gebunden
- Bindung endet bei Programmende
- Vorteil: Variablen könne im Programm direkt adressiert werden → effizienter Zugriff
- Nachteil: Speicherzelle ist für gesamte Laufzeit belegt, kann nicht von anderen Variablen genutzt werden

Imperative Sprachen

Stack-dynamische Speicher-Bindung :

- bei Variablen mit „festem“ Typ
- Bindung am Speicherzelle im Moment der Deklaration / des Eintritts in den die Deklaration umgebenden Block
- Aufheben der Bindung bei Verlassen der/des umgebenden Funktion/Blocks
- Speicherzelle gehört zum sog. „Runtime-Stack“
- Vorteil: Variablen, die nicht gleichzeitig leben, können gleiche Zelle benutzen
- Nachteil: Allokation/Deallokation kostet (etwas), Speicher i.d.R. indirekt adressiert (Zugriff langsamer)

Imperative Sprachen

Explizit Heap-dynamische Speicher-Bindung :

- Bindung mit “namenlosen“ Speicherzellen
- Zellen können explizit reserviert/freigegeben werden
- Zellen gehören zum Heap
- Beispiel: alle Java-Objekte; aber wegen Garbage Collection keine explizite Freigabe nötig
- Beispiel: C++

  `new` liefert Pointer auf Speicherzelle

  `delete` deallokiert Speicher wieder
- Beispiel: C#

  stack-dynamische Objekte und explizit Heap-dyn. Objekte

  Garbage Collection → automatische Freigabe

  Pointer, können auf bereits freigegebenen Bereich zeigen →

  Methoden mit Pointer müssen `unsafe` im Header aufführen

Imperative Sprachen

Explizit Heap-dynamische Speicher-Bindung :

- Vorteil: explizit Heap-dyn. Variablen gut für dynamische Strukturen wie verkettete Listen, Bäume
- Nachteil: Heap-Management schwieriger zu implementieren, kostet Zeit

Imperative Sprachen

implizit Heap-dynamische Speicher-Bindung :

- Bindung der Variablen bei Wertzuweisung
- Speicherzellen vom Heap
- in typlosen Sprachen (z.B. JavaScript)
- alle Variablenattribute werden gleichzeitig gebunden

- Vorteil: extrem flexibel
- Nachteil kostet Zeit, fehleranfällig

Imperative Sprachen

Einfache Datentypen (Basistypen)

- Numerische:
  - Integer (byte, short, int long, unsigned,..;
    Einer-, Zweierkomplement)
  - Fließkomma-Zahlen: float, double (real; Mantisse, Exponent)
  - Komplexe Zahlen
  - Dezimalzahl
- Boolescher Typ (true/false; in C: ≠/= 0)
- Character-Typen (8-bit ASCII, Unicode, UTF-32)

Imperative Sprachen

Einfache Datentypen (Basistypen)

- Strings:
  - Grundsatzentscheidungen:

    Als Basistyp oder als Array vom Typ „character“?

    Mit fester oder dynamischer Länge?
  - Basisoperationen: Zuweisung, Konkatenation, Vergleich, Substring, Pattern matching
  - Kein Basistyp: realisiert über char-Arrays mit Längenangabe oder mit Ende-Kennung
  - Problem: manche Stringfunktionen berücksichtigen Länge/Ende-Kennung nicht: `strcpy(destin, source);`

Imperative Sprachen

Einfache Datentypen (Basistypen)

- Strings (2):

  Implementierungsformen:

  – bei statischer Länge: Name des Typs, Länge, Startadresse
  – bei begrenzter dyn. Stringlänge: Name, maximale Länge, aktuelle Länge, Startadresse
  – bei dyn. Länge:
    1. als verkettete Liste
    2. als Felder von Pointern, die auf Charakter im Heap zeigen
    3. in direkt aufeinander folgenden Speicherzellen -> wächst String evtl. neuen Speicherbereich auf Heap suchen, dorthin verschieben